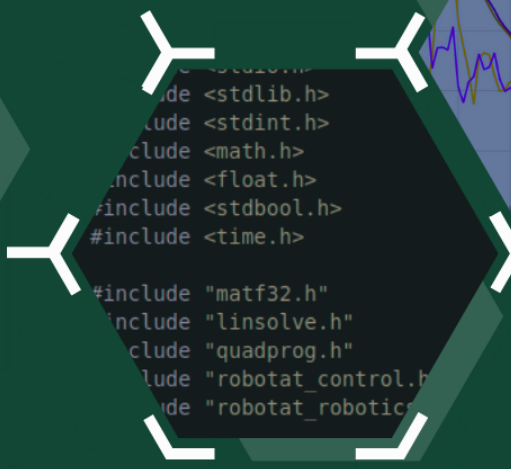
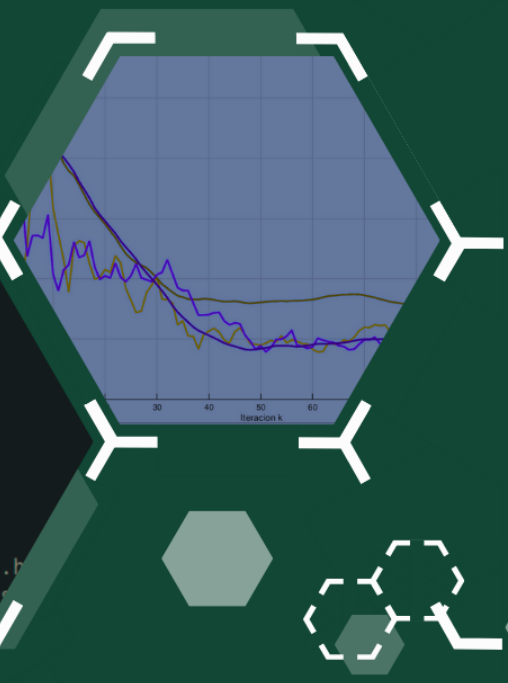
Implementación y validación de algoritmos de control y robótica empleando la librería Robotat Linalg

Laura Andrea Barrientos Pineda





UVG | UNIVERSIDAD DEL VALLE DE GUATEMALA

UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería

Implementación y validación de algoritmos de control y robótica empleando la librería Robotat Linalg

Trabajo de graduación presentado por Laura Andrea Barrientos Pineda
para optar al grado académico de Licenciada en Ingeniería Mecatrónica

Guatemala,

2025

UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería

Implementación y validación de algoritmos de control y robótica empleando la librería Robotat Linalg

Trabajo de graduación presentado por Laura Andrea Barrientos Pineda
para optar al grado académico de Licenciada en Ingeniería Mecatrónica

Guatemala,

2025

Vo.Bo.:

(f) _____

MSc. Miguel Enrique Zea Arenales

Tribunal Examinador:

(f) _____

MSc. Miguel Enrique Zea Arenales

(f) _____

PhD. Luis Alberto Rivera

(f) _____

MSc. Carlos Esquit

Fecha de aprobación: Guatemala, 7 de enero de 2026.

Prefacio

Escoger este proyecto como mi tesis no estaba originalmente en mis planes. Aunque, durante varios años me imaginé que, por una razón u otra, yo le daría continuidad. Parcialmente, porque mi hermano lo trabajó en su momento, pero también por mi afinidad a los temas de matemática y programación. Sin embargo, todo ese tiempo tuve la preocupación y temor de que si llegaba a trabajar en este proyecto, de alguna manera me marcaría como un reflejo de mi hermano en lugar de verme como mi propia persona, de la misma forma que me ha sucedido en otras ocasiones. Agradezco mucho a Dios que no fue el caso y que, en cambio, esta fue una buena experiencia y que disfruté bastante realizar este proyecto.

Parte de esto fue gracias a mi asesor, el M.Sc. Miguel Zea, quien me ha dado su apoyo desde el momento que le dije que estaba buscando un nuevo proyecto para mi tesis y que ha expresado continuamente su confianza en mi capacidad de llevarlo a cabo. Me dio el ánimo que yo no tenía inicialmente para hacer otra tesis, buscando que diera lo mejor de mí en el proceso. Le agradezco por dedicar el tiempo para explicarme los temas que no entendía o que eran nuevos para mí, por resolver mis dudas y por la paciencia de escucharme cada semana, en especial cuando yo estaba tan nerviosa que no podía explicar bien.

Agradezco además a mi hermano Daniel, quien me ayudó a resolver dudas acerca del código que desarrolló previamente para este proyecto y quien, al igual que mi asesor, tuvo que escuchar todas mis dudas y delirios matemáticos en estos meses. Asímismo, agradezco a mis amigos que me apoyaron durante este proyecto, especialmente a Estuardo Menendez, el matemático de confianza, quien me ayudó a resolver dudas y aclarar conceptos de distintos métodos numéricos, a Estuardo Mancio Ruballos, quien ha sido un gran apoyo moral para mí a lo largo de este año, y a Daniel Cortez, por la ayuda con el formato de la tesis.

Adicionalmente, agradezco a las personas que me han dado apoyo moral o emocional y mostrado interés en mi bienestar en estos meses, incluyendo al Dr. Luis Rivera, el Ing. Kurt Kellner, el Ing. Adrián Martínez, la Ing. Magda Moscoso, el M.Sc. José Bagur y el Ing. Estuardo Mancio. También, al Ing. Andrés Viau, a quien además del apoyo moral, debo agradecer por la difícil (pero positiva) experiencia de aprendizaje que tuve en uno de sus cursos porque, sin eso, probablemente me habría cambiado de carrera en tercer año. Finalmente, gracias a las personas que me convencieron de seguir adelante y no abandonar mi carrera, en el breve tiempo que lo consideré antes de escoger una nueva tesis.

Índice general

Índice de figuras

Índice de cuadros

Resumen

En este trabajo se completó y validó la librería de computación numérica Robotat Linalg, previamente desarrollada en la Universidad del Valle. Se completaron las rutinas principales para álgebra lineal, y los solucionadores lineal y cuadrático, validando que pueden generar resultados con cinco decimales de exactitud al comparar su rendimiento con MATLAB, con tiempos de operación de milisegundos o menos para la mayoría de pruebas. Se completó la implementación de la librería Robotat Control para algoritmos de control, validando las rutinas para controladores PID, sistemas lineales y no lineales, filtros de Kalman, y añadiendo rutinas para expandir la librería, incluyendo el algoritmo de un tipo de control de modelo predictivo. Se desarrolló y validó la librería Robotat Robotics para algoritmos de robótica, con las rutinas esenciales para cálculos de pose. Todas las librerías se validaron en las siguientes plataformas: una computadora con arquitectura amd-64, un ESP32, un Arduino MEGA y una STM NUCLEO F446RE. En la validación de todas las librerías, el uso de memoria RAM del ESP32 y la NUCLEO, fue menor al 15 %. La memoria FLASH se mantuvo alrededor del 20 % en el ESP32 y del 6 % en la NUCLEO. En el Arduino MEGA, dos métodos del solucionador cuadrático y el control de modelo predictivo superaron la capacidad de la RAM, por lo que no se lograron evaluar en este dispositivo. Se concluyó que el ESP32 y la NUCLEO son adecuados para aplicaciones complejas empleando estas librerías, mientras que el Arduino MEGA en operaciones de menor escala.

Palabras clave: computación numérica, optimización convexa, control, robótica, dispositivos embebidos.

Abstract

This work focused on finishing the implementation of the numerical computation library Robotat Linalg, developed previously in Universidad del Valle de Guatemala. The routines for linear algebra, and for the linear and quadratic solvers were completed, validating they are capable of achieving a numerical precision of five decimals when compared with MATLAB, with operation times of miliseconds or smaller in most cases. The implementation of the library Robotat Control for control algorithms was completed, validating the routines for the PID controllers, linear and non linear systems, and the Kalman Filter, also adding new routines, including a type of control of predictive model. The library Robotat Robotics for robotics algorithms was developed, including the most important routines for robot pose calculations. All libraries were validated in the following platforms: a computer with amd-64 architecture, an ESP32, an Arduino MEGA and a NUCLEO F446RE. The RAM memory usage of the ESP32 and NUCLEO during the validations remained below 15 % with all of the libraries, while the FLASH memory was measured around 20 % for the ESP32 and around 6 % for the NUCLEO. For the Arduino MEGA, two methods for the quadratic solver and the predictive control model surpassed the RAM capacity, so these were not evaluated in this device. The ESP32 and NUCLEO were determined to be adequate for complex applications with these libraries, while the Arduino MEGA is recommended to use in smaller scale operations.

Keywords: numerical computation, convex optimization, control, robotics, embedded devices.

CAPÍTULO 1

Introducción

En este trabajo se continúa con el desarrollo de la librería Robotat Linalg desarollada previamente en la Universidad del Valle de Guatemala (UVG), que está compuesta por tres librerías: `matf32` para álgebra lineal, `linsolve` para solución de sistemas lineales y `quadprog` para solución de problemas cuadráticos convexos. Este trabajo se enfocó en validar las tres librerías de Robotat Linalg para luego implementar y validar dos librerías adicionales, para algoritmos de control y robótica, respectivamente.

Para esto, primero se describen los antecedentes relevantes para este trabajo (Capítulo 2), y la importancia de realizar el mismo (Capítulos 3). Seguido de los objetivos (Capítulo 4), el alcance del proyecto (Capítulos 5), así como los conceptos de teoría más importantes (Capítulo 6). Luego, se describen los cambios que se realizaron a las librerías, con respecto de la versión anterior de Robotat Linalg, y los resultados de la evaluación del rendimiento de las rutinas principales de `matf32`, `linsolve` y `quadprog` (Capítulo 7). Seguido de esto, se presenta la implementación de las librerías Robotat Control y Robotat Robotics, para control y robótica, respectivamente, junto a los resultados de la evaluación de sus rutinas (Capítulo 8). Posteriormente, se presentan las conclusiones obtenidas durante este trabajo y las recomendaciones para su continuación en trabajos futuros (Capítulos 9 y 10, respectivamente). Asímismo, se incluye la bibliografía empleada para la realización de este trabajo (Capítulo 11). Por último, se incluyen en los anexos de este documento las gráficas y cuadros de evaluación empleados para distintas rutinas (Capítulo 12).

Por medio de este trabajo, se validó el correcto funcionamiento de las tres librerías que conforman Robotat Linalg, así como de las librerías Robotat Control y Robotat Robotics. Las plataformas de evaluación empleadas fueron las siguientes: una computadora con arquitectura amd-64, un ESP32, un Arduino MEGA y una STM NUCLEO F446RE, empleando MATLAB como referencia para el rendimiento de cada rutina. Tanto el ESP32 y la NUCLEO se consideraron adecuadas para emplear estas librerías por su mayor capacidad de memoria, mientras que el Arduino MEGA mostró limitación con algunas rutinas por lo que es más apropiado emplearlo en aplicaciones menores.

CAPÍTULO 2

Antecedentes

La optimización numérica es ampliamente utilizada en múltiples ramas de la ingeniería para solucionar problemas de forma más eficiente [1]. Esto incluye ingeniería electrónica, civil, química, mecánica y aeroespacial, así como en otras aplicaciones como sistemas de control, diseño de redes, finanzas, planificación, entre otras [1]. Algunos tipos de optimización se ajustan mejor a ciertas aplicaciones, como es el caso de la optimización convexa que tiene mayor utilidad para control óptimo y aprendizaje automático, en donde los problemas se resuelven repetidas veces [2].

La optimización convexa presenta algunas ventajas a comparación de otros tipos de optimización no lineal, ya que estos problemas son más predecibles y fáciles de resolver ya que no requieren tantos ajustes y configuraciones [2]. Aparte de eso, la optimización convexa se divide en distintas categorías (programación lineal, cuadrática, cónica, entre otras), enfocadas a resolver diferentes tipos de problemas [2]. Entre estas, la programación cuadrática es cada vez más utilizada para aplicaciones de robótica ya que muchos problemas en esta área pueden representarse como programas cuadráticos [3].

Por otro lado, implementar algoritmos de optimización convexa en dispositivos embebidos tiene algunas ventajas, principalmente, que no se requiere tanta precisión y que se resuelve múltiples veces el mismo tipo o familia de problemas [4]. Por lo mismo, el desarrollo de los algoritmos se puede enfocar en reducir el tiempo de resolución, sabiendo las características específicas de los problemas a resolver [4].

Sin embargo, para resolver problemas convexos en dispositivos embebidos es necesario cumplir con varios requisitos, principalmente: el algoritmo debe ser robusto, rápido (alcanzar una solución en milisegundos o microsegundos), debe ser sencillo e independiente de librerías externas. [4] Adicionalmente, el lenguaje utilizado debe ser adecuado para aplicaciones donde la seguridad es crítica, como el lenguaje C [5]. A partir de esto, se han desarrollado múltiples herramientas para resolver problemas de optimización.

Una de estas es CVXGEN, un generador de código para resolver problemas de optimización [4]. Sin embargo, tiene la desventaja de que el tamaño del código incrementa conforme

aumentan las dimensiones del problema a resolver [6]. Por eso mismo, se han desarrollado alternativas, incluyendo un generador de código basado en el solucionador de problemas cuadráticos OSQP, el cual se comparó con CVXGEN y otros generadores de código (GUROBI, qpOASES, FiOrdOs), demostrando ser más rápido que todos estos al resolver problemas [6].

El solucionador OSQP (Operator Splitting Quadratic Program) fue desarrollado en C, es de código abierto y tiene interfaz para otros lenguajes de programación, como Matlab y Python. Está enfocado en resolver problemas de álgebra lineal dispersa, implementando un método de multiplicadores de dirección alternante (ADMM), el cual forma parte de los métodos iterativos de primer orden. [7].

Otro solucionador es DAQP, en el cual se implementaron dos versiones del algoritmo, distinguidas como: DAQP, que implementa un método iterativo de conjunto activo dual y DAQP PROX, que añade al método el uso de iteraciones de punto proximal [8]. Tanto DAQP como DARQ PROX se compararon con otros solucionadores (OSQP, HPIPM, QPKWIK, qpOASES_c) en una aplicación de modelo de control predictivo en MATLAB. Estos fueron más eficientes que los otros solucionadores en problemas relativamente pequeños o medianos, por lo que se consideraron adecuados para uso en sistemas embebidos, aunque se mencionó que otros solucionadores podrían ser más adecuados para problemas más grandes [8].

De forma similar, el solucionador embQP, implementa un método de conjunto activo dual. Este se evaluó en una aplicación de control para seguimiento de rutas, enfocado en vehículos autónomos, comparándolo con el solucionador QL desarrollado en Fortran, que de por sí es rápido y eficiente. El funcionamiento de embQP fue bastante similar a QL, por lo que se concluyó que es adecuado para implementar en sistemas embebidos [5].

Para aplicaciones específicamente de robótica, existe el solucionador qpSWIFT [3]. Este implementa un método de punto interior para resolver problemas cuadráticos y ha sido evaluado en distintas aplicaciones de software como optimización de portafolios, máquinas de soporte de vectores (SVMs) y un algoritmo de modelo de control predictivo (MPC), en donde resultó ser más eficiente que otros algoritmos (quadprog, ECOS, Gurobi, Mosek y OSQP); así como para el control de pose y funciones de caminar de un robot cuadrúpedo, los cuales se lograron ejecutar [3].

Por otro lado, también esta la librería Robotat Linalg, desarrollada anteriormente en la Universidad del Valle de Guatemala (UVG) para aplicaciones de control y robótica. Fue desarrollada en el lenguaje C, utilizando únicamente librerías propias del lenguaje (por ejemplo: `math.h`, `stdio.h`, `stdlib.h`, `float.h`, entre otras). Está organizada en sublibrerías para funciones de álgebra matricial, métodos directos para resolver sistemas lineales (descomposiciones LU, Cholesky, QR) y para resolver problemas cuadráticos convexos. Se evaluaron algunas de sus rutinas en una computadora con arquitectura amd-64 y sistema operativo Linux Mint, así como en un ESP32S y una Tiva C, comparando el tiempo de operación de las rutinas contra el tamaño de las matrices empleadas.

CAPÍTULO 3

Justificación

La optimización numérica es de gran importancia para distintas áreas de la ingeniería. Dependiendo del problema o situación a resolver, existen igualmente distintos tipos de optimización y métodos que se pueden emplear. Dentro de estos se encuentra la optimización convexa, y específicamente, la programación cuadrática, que es bastante utilizada para aplicaciones relacionadas al análisis de datos, inteligencia artificial y robótica. En el caso de la robótica, actualmente hay interés en desarrollar aplicaciones en sistemas embebidos. Sin embargo, para esto es necesario desarrollar algoritmos de computación numérica que sean eficientes de utilizar en dispositivos embebidos cuya capacidad de memoria y procesamiento es menor comparada con otros dispositivos.

Una de las soluciones para esta problemática es el desarrollo de herramientas, adaptadas a dispositivos embebidos para la solución de problemas cuadráticos convexos. Dentro de esta categoría se incluyen generadores de código y librerías con funciones especializadas. A partir de eso, en la Universidad del Valle de Guatemala (UVG) se ha desarrollado la librería de computación numérica Robotat Linalg, enfocada en el desarrollo de aplicaciones empleando dispositivos embebidos. Un aspecto clave en su desarrollo es que pueda utilizarse en diversos dispositivos embebidos, en lugar de estar especializada o limitada a un dispositivo en específico, logrando esto a partir de la estructura del código desarrollado en lenguaje C.

No obstante, aún es necesario evaluar el funcionamiento de esta librería en algoritmos de control y robótica. A partir de eso, el presente trabajo busca continuar el desarrollo de esta librería, evaluando su rendimiento en diferentes algoritmos relevantes a control y robótica. Para esto, se utilizan diferentes dispositivos embebidos para determinar cuáles son más eficientes para la utilización de de esta librería. A largo plazo, se busca emplear Robotat Linalg, y las librerías correspondientes para control y robótica, con el propósito de expandir las capacidades del laboratorio de robótica Robotat, de la UVG, además de ser una herramienta didáctica para los estudiantes en los cursos de control y robótica.

CAPÍTULO 4

Objetivos

4.1. Objetivo general

Continuar el desarrollo de la librería Robotat Linalg y utilizarla para implementar y evaluar algoritmos representativos de control y robótica.

4.2. Objetivos específicos

- Evaluar y validar la implementación de las rutinas actuales de álgebra lineal de la librería Robotat Linalg.
- Completar la implementación de las rutinas de solución de sistemas lineales y de programas cuadráticos convexos.
- Emplear las librerías de computación numérica de Robotat Linalg para implementar y evaluar algoritmos representativos de control y robótica en microcontroladores.

CAPÍTULO 5

Alcance

Este trabajo de graduación se enfoca en evaluar el rendimiento de las librerías de computación numérica de Robotat Linalg, siendo estas: `matf32` para álgebra lineal, `linsolve` para el solucionador lineal y `quadprog` para el solucionador cuadrático. Para esto, las métricas a emplear son las siguientes: tiempo de operación y exactitud numérica con respecto de un valor definido como resultado teórico o referencia, y el uso de memoria de los dispositivos embebidos empleados durante la evaluación. Asímismo, en cada librería se añaden las rutinas que se consideren necesarias para completar su implementación. Finalmente, se evalúa el rendimiento de distintos algoritmos de control y robótica representativos, empleando las rutinas disponibles en Robotat Linalg.

CAPÍTULO 6

Marco teórico

La optimización se subdivide en distintos tipos, dependiendo del problema a resolver, que puede tener restricciones (o no restricción alguna), puede ser lineal o no lineal, entre otras categorías más específicas como optimización discreta y convexa [2]. Para resolver un problema de optimización, es necesario realizar operaciones numéricas para solucionar sistemas de álgebra lineal, para lo cual también existen diversos métodos. A partir de eso, a continuación se presenta un resumen de los temas más relevantes para este trabajo de graduación (especialmente, sistemas lineales y programas cuadráticos convexos), así como diferentes métodos para resolver cada uno de estos.

6.1. Sistemas lineales

Los sistemas lineales son fundamentales para la computación científica, y la rapidez y eficiencia computacional son clave para resolverlos. Para ello, existen distintos métodos: directos, que se refiere a calcular y manipulan directamente la matriz que representa al sistema por medio de distintas operaciones; e iterativos, cuando se aproxima la solución con métodos numéricos sin utilizar directamente la matriz. Muchos sistemas de interés se pueden escribir como $\mathbf{Ax} = \mathbf{b}$, en donde $\mathbf{A}$ es una matriz, mientras que $\mathbf{x}$ y $\mathbf{b}$ son vectores [9].

Tanto los métodos directos como los iterativos tienen ventajas y desventajas que se deben tomar en cuenta a la hora de escoger cuál utilizar. Los métodos directos tienden a ser más eficientes y robustos, pero la respuesta se obtiene hasta el final del procedimiento, por lo que no se puede controlar el costo computacional resultante [2]. Por el contrario, los métodos iterativos pueden detenerse en cualquier punto del proceso y aún así obtener una aproximación de la respuesta, permitiendo un mayor control sobre la precisión y costo computacional [2]. Los métodos iterativos también presentan una ventaja cuando se conoce un estimado del valor de la solución para introducir un valor inicial que acelere la convergencia del mismo y son más eficientes cuando se aplican a problemas grandes y dispersos, precisamente porque no necesitan manipular las matrices directamente [2].

6.1.1. Factorizaciones de matrices

6.1.1.1. Factorización LU

En esta factorización, la matriz $\mathbf{A}$ se descompone en dos matrices, denominadas superior ($\mathbf{U}$) e inferior ($\mathbf{L}$). Para definir las matrices $\mathbf{U}$ y $\mathbf{L}$, es necesario aplicar eliminación gaussiana. A partir de eso, se construyen ecuaciones que permiten despejar las variables y se sustituyen los coeficientes de las matrices. La matriz se define ahora como $\mathbf{A} = \mathbf{LU}$ [9].

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ m_{21} & 1 & 0 \\ m_{31} & m_{32} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}. \tag{1}$$

Dado que $\mathbf{Ax} = \mathbf{b}$ y $\mathbf{A} = \mathbf{LU}$, se obtiene $\mathbf{LUx} = \mathbf{b}$. Una vez establecidas las matrices, se redefine el nuevo sistema como $\mathbf{Ly} = \mathbf{b}$, en donde $\mathbf{y} = \mathbf{Ux}$. [9]. Entonces, para solucionar un sistema lineal $\mathbf{Ax} = \mathbf{b}$ con métodos numéricos, se realiza lo siguiente:

1. Calcular la factorización $\mathbf{A} = \mathbf{LU}$.
2. Solucionar $\mathbf{Ly} = \mathbf{b}$ usando sustitución hacia adelante.
3. Solucionar $\mathbf{Ux} = \mathbf{y}$ usando sustitución hacia atrás.

Para calcular la factorización LU existen diferentes algoritmos, algunos de los cuales incluyen pivoteo o permutaciones para mayor estabilidad [10].

6.1.1.2. Factorización Cholesky

La factorización Cholesky se basa en la descomposición LU, pero utilizando únicamente la matriz inferior $\mathbf{L}$ [1]. Este método es aplicable únicamente a matrices simétricas positivas definidas, por lo que sus aplicaciones son más limitadas comparadas con otros métodos [10]. Se define como $\mathbf{A} = \mathbf{LL}^\top$, en donde $\mathbf{L}$ se conoce como el factor Cholesky y es una matriz triangular inferior.

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} m_{11} & 0 & 0 \\ m_{21} & m_{22} & 0 \\ m_{31} & m_{32} & m_{33} \end{bmatrix} \begin{bmatrix} m_{11} & m_{21} & m_{31} \\ 0 & m_{22} & m_{32} \\ 0 & 0 & m_{33} \end{bmatrix}. \tag{2}$$

En computación numérica, el procedimiento para resolver un sistema lineal con la factorización Cholesky es el siguiente [10]:

1. Calcular la factorización $\mathbf{A} = \mathbf{LL}^\top$.
2. Resolver $\mathbf{Ly} = \mathbf{b}$ con sustitución hacia adelante.
3. Resolver $\mathbf{L}^\top\mathbf{x} = \mathbf{y}$ con sustitución hacia atrás.

6.1.1.3. Factorización QR

La factorización QR se puede aplicar tanto a matrices cuadradas como rectangulares (no-cuadradas) por lo que es un método versatil, además de ser robusto y estable en general. Esta se define como: $\mathbf{A} = \mathbf{QR}$ [11],

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} q_{11} & q_{12} & q_{13} \\ q_{21} & q_{22} & q_{23} \\ q_{31} & q_{32} & q_{33} \end{bmatrix} \begin{bmatrix} r_{11} & r_{21} & r_{31} \\ 0 & r_{22} & r_{32} \\ 0 & 0 & r_{33} \end{bmatrix}. \tag{3}$$

A partir de esto, para solucionarla por medio de computación numérica, se realizan los siguientes pasos:

1. Calcular la factorización $\mathbf{A} = \mathbf{QR}$
2. Calcular el vector $\mathbf{y} = \mathbf{Q}^\top \mathbf{b}$
3. Resolver $\mathbf{Rx} = \mathbf{y}$ con sustitución hacia atrás

6.1.1.4. Descomposición de Valores Singulares (SVD)

Este método está definido de la siguiente manera: $\mathbf{A} = \mathbf{U\Sigma V}^\top$, en donde $\mathbf{U}$ y $\mathbf{V}$ son matrices cuadradas ortonormales que representan rotaciones aplicadas a la matriz $\mathbf{A}$, mientras que $\mathbf{S}$ es una matriz diagonal que contiene los valores singulares de $\mathbf{A}$. Cada valor singular corresponde a la raíz cuadrada de cada eigenvalor de $\mathbf{A}$, por lo que una de las estrategias para generar SVD es calculando primero los eigenvalores. Sin embargo, estos métodos pueden ser costosos ya que implican construir explícitamente las matrices $\mathbf{AA}^\top$ y $\mathbf{A}^\top\mathbf{A}$ [12] [10].

Sin embargo, también existen métodos para calcular la SVD sin tener que calcular todos los eigenvalores de $\mathbf{A}$. Cabe destacar que estos son iterativos, porque se busca aproximar los valores singulares de la descomposición para evitar calcular directamente $\mathbf{AA}^\top$ y $\mathbf{A}^\top\mathbf{A}$, ahorrando así memoria y construyendo $\mathbf{U}$ y $\mathbf{V}$ en el proceso. Uno de estos es la SVD de Golub-Kahan, que implementa una reducción bidiagonal y luego el algoritmo QR iterativo con desfases (*QR Algorithm with Shifts*), sin embargo, calcula algunos eigenvalores como estimación para la iteración inicial [10] [12].

El método más sencillo para implementar la SVD, aunque es más lento que otros, es el método de Jacobi unilateral (*One-Sided Jacobi)* [12] [13], en el cual se aplican rotaciones de la siguiente manera:

$$\begin{aligned} \mathbf{A} &= \mathbf{AJ}, \\ \mathbf{V} &= \mathbf{VJ}, \end{aligned} \tag{4}$$

en donde $\mathbf{J}$ es la matriz de rotación de Jacobi. Eventualmente, después de suficientes iteraciones, se genera la matriz $\mathbf{AV} = \mathbf{U\Sigma}$, siendo $\mathbf{V}$ una matriz ortogonal en la que se acumularon las mismas rotaciones aplicadas a $\mathbf{A}$ [13].

Cada valor singular se calcula como la norma de cada columna de $\mathbf{AV}$ y se almacena en $\mathbf{\Sigma}$. Finalmente, cada columna de $\mathbf{U}$ se genera normalizando la respectiva columna de $\mathbf{AV}$ al dividirla entre el valor singular correspondiente [12].

La solución a un sistema $\mathbf{Ax} = \mathbf{b}$ por medio de SVD, es la siguiente:

$$\mathbf{x} = \mathbf{V}\mathbf{S}^{-1}\mathbf{U}^{\top}\mathbf{b}. \tag{5}$$

6.1.1.5. Factorización $\mathbf{LDL}^{\top}$

La factorización $\mathrm{LDL}^{\top}$ es similar a la factorización Cholesky, pero tiene una matriz diagonal adicional entre los factores triangulares. Está definida de la siguiente manera:

$$\mathbf{PAP}^{\top} = \mathbf{LDL}^{\top}, \tag{6}$$

en donde $\mathbf{A}$ es la matriz a factorizar, $\mathbf{P}$ es una matriz de permutación, $\mathbf{L}$ es una matriz triangular inferior y $\mathbf{D}$ es una matriz diagonal [12].

Esta factorización suele emplearse en matrices simétricas definidas o semidefinidas. Sin embargo, también puede aplicarse a matrices indefinidas con ciertas modificaciones, por lo cual es útil para solucionar algunos problemas de optimización cuadrática [12] [11]. Para cuidar la estabilidad y simetría de la matriz, suelen implementarse estrategias de pivoteo o permutación de filas, por ejemplo, pivoteo simétrico. No obstante, para matrices simétricas indefinidas, este método de pivoteo no es útil y, en cambio, se recomienda utilizar el método de Aasen o, preferentemente, el método de Bunch y Parlett [12] [11].

Para solucionar un sistema lineal $\mathbf{Ax} = \mathbf{b}$ empleando $\mathrm{LDL}^{\top}$, se deben realizar los siguientes pasos [11]:

1. Calcular factorización $\mathbf{PAP}^{\top} = \mathbf{LDL}^{\top}$.
2. Resolver $\mathbf{Lz}_1 = \mathbf{Pb}$ con sustitución hacia adelante.
3. Resolver $\mathbf{Dz}_2 = \mathbf{z}_1$ con algún método válido.
4. Resolver $\mathbf{L}^{\top}\mathbf{z}_3 = \mathbf{z}_2$ con sustitución hacia atrás.
5. Calcular $\mathbf{x} = \mathbf{P}^{\top}\mathbf{z}_3$.

6.1.2. Métodos iterativos

6.1.2.1. Método de Jacobi

A partir de un sistema lineal:

$$\left[\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \end{array}\right] \quad \rightarrow \quad \begin{array}{l} a_{11}x + a_{12}y + a_{13}z = b_1 \\ a_{21}x + a_{22}y + a_{23}z = b_2 \\ a_{31}x + a_{32}y + a_{33}z = b_3 \end{array}, \tag{7}$$

se despeja una ecuación para cada una de las variables (x, y & z):

$$\begin{aligned} x_{k+1} &= \frac{1}{a_{11}}(b_1 - a_{12}y_k - a_{13}z_k), \\ y_{k+1} &= \frac{1}{a_{22}}(b_2 - a_{21}x_k - a_{23}z_k), \\ z_{k+1} &= \frac{1}{a_{33}}(b_3 - a_{31}x_k - a_{32}y_k). \end{aligned} \tag{8}$$

Estas ecuaciones se utilizan para aproximar los valores de la solución, proceso que se itera constantemente hasta llegar a un resultado satisfactorio [9]. El procedimiento general es el siguiente:

1. Escoger valores iniciales para las variables x_0, y_0 y z_0.
2. Iterar para calcular el siguiente valor de las variables.
3. Verificar si hay convergencia en el rango deseado, por ejemplo: $|x_{k+1} - x_k| <$ tolerancia

El método de Jacobi está garantizado que converge para matrices con diagonal estrictamente dominante. Es decir, que para cada fila de la matriz $\mathbf{A}$, el valor absoluto de su pivote sea mayor a la suma de los valores absolutos de los otros elementos de la fila [9].

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \quad \rightarrow \quad \begin{aligned} &|a_{11}| > |a_{12}| + |a_{13}| \\ &|a_{21}| > |a_{22}| + |a_{23}|. \\ &|a_{31}| > |a_{32}| + |a_{33}| \end{aligned} \tag{9}$$

Cambiar el orden de las filas puede alterar esto y, por consecuencia, puede haber divergencia en el resultado con el método de Jacobi [9]. Adicionalmente, existen diferentes implementaciones del método de Jacobi, por lo que el método más adecuado debe escogerse según el problema a resolver [13] [12].

6.1.2.2. Método de Gauss-Seidel

Se basa en el método de Jacobi, únicamente modificando las ecuaciones para las variables. En la ecuación de y_{k+1} se sustituye x_k por x_{k+1}. De forma similar, para z_{k+1} se utiliza x_{k+1} y y_{k+1}, en lugar de x_k y y_k [9].

$$\begin{aligned} x_{k+1} &= \frac{1}{a_{11}}(b_1 - a_{12}y_k - a_{13}z_k), \\ y_{k+1} &= \frac{1}{a_{22}}(b_2 - a_{21}x_{k+1} - a_{23}z_k), \\ z_{k+1} &= \frac{1}{a_{33}}(b_3 - a_{31}x_{k+1} - a_{32}y_{k+1}). \end{aligned} \tag{10}$$

Estos cambios pueden reducir el número de iteraciones necesarias para converger y, al igual que el método de Jacobi, está garantizado que converge cuando las matrices tienen diagonal estrictamente dominante [9].

6.1.2.3. Método del gradiente

Este método es una alternativa a los métodos de Jacobi y Gauss-Seidel ya que tiene un menor costo computacional [9]. Este método empieza con la función cuadrática:

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{A}\mathbf{x} - \mathbf{b}^\top \mathbf{x} + c, \tag{11}$$

en donde $\frac{1}{2}\mathbf{x}^\top \mathbf{A}\mathbf{x}$ es el término cuadrático, $-\mathbf{b}^\top \mathbf{x}$ es el término lineal y c es una constante. El gradiente de esta función es:

$$\nabla f(\mathbf{x}) = \frac{1}{2}\mathbf{A}^\top \mathbf{x} + \frac{1}{2}\mathbf{A}\mathbf{x} - \mathbf{b}. \tag{12}$$

La forma en la que está escrito el gradiente asume que la matriz $\mathbf{A}$ no es simétrica (es decir, tiene dimensiones $m \times n$ [9]. Por lo mismo, cuando la matriz es simétrica (dimensiones $n \times n$), la ecuación del gradiente se reduce a:

$$\nabla f(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b}. \tag{13}$$

El punto crítico de $f(\mathbf{x})$ cuando $\nabla f(\mathbf{x}) = 0$ y $\mathbf{A}\mathbf{x} = \mathbf{b}$, además, si la matriz $\mathbf{A}$ es simétrica positiva definida entonces la solución obtenida es un mínimo de la función $f(\mathbf{x})$ [9]. Por el contrario, si $\mathbf{A}$ no es simétrica positiva definida, puede ser uno de los siguientes casos: matriz negativa definida, matriz indefinida (resultando en un punto silla) o matriz singular [9]. En estos casos, el método del gradiente puede no funcionar [9].

Para reducir el consumo de recursos computacionales, se busca el mínimo de $f(\mathbf{x})$. Para eso, el método del gradiente esencialmente busca la ruta más corta hacia la solución [9]. El procedimiento general para el método del gradiente es el siguiente:

1 Calcular un valor inicial para el gradiente $\nabla f(\mathbf{x})$.

2 Cacular el punto de la siguiente iteración con la ecuación: $\xi(\tau) = \mathbf{x} - \tau \nabla f(\mathbf{x})$, en donde τ indica cuánto se mueve la función en dirección del gradiente, de manera que tanto $\nabla f(\mathbf{x})$ y $\nabla f(\xi)$ sean ortogonales.

6.1.2.4. Métodos del subespacio de Krylov

Un subespacio de Krylov se define de la siguiente manera

$$\mathcal{K}_k(A; r_0) = \text{span}\{r_0, Ar_0, A^2 r_0, ..., A^{k-1} r_0\}, \tag{14}$$

en el cual, una solución se representa como una combinación lineal de todos los términos dentro de la definición: $\{r_0, Ar_0, A^2 r_0, ..., A^{k-1} r_0\}$ [2].

Estos métodos generalmente son más eficientes que los métodos iterativos estacionarios o de punto fijo (Jacobi, Gauss-Seidel y método del gradiente). Una de las diferencias es que

implementan lo que se conoce como precondicionamiento [2]. Esta operación modifica la escala del sistema lineal $\mathbf{Ax} = \mathbf{b}$, multiplicando una nueva matriz $\mathbf{M}$:

$$(\mathbf{M}^{-1}\mathbf{A})\mathbf{x} = \mathbf{M}^{-1}\mathbf{b}. \tag{15}$$

Adicionalmente, $\mathbf{M}$ se modifica en cada iteración, conforme a los datos calculados. Sin embargo, igual que en otros métodos iterativos, no se manipula la matriz $\mathbf{A}$ ni $\mathbf{M}$ directamente. En el caso de $\mathbf{M}$, solo se necesita estimar su inversa $\mathbf{M}^{-1}$ [2].

Un solucionador que implementa métodos del subespacio de Krylov, debe tener los siguientes requerimientos:

1 Una base ortonormal para el subespacio de Krylov.

2 Una propiedad óptima que determina la solución dentro del subespacio.

3 Un precondicionamiento efectivo.

6.2. Optimización numérica

La optimización se puede encontrar en diversas áreas, incluyendo economía, administración, manufactura, biología, física e ingeniería (por ejemplo química, eléctrica, mecánica, aeroespacial, entre otras). Dentro de la ingeniería, es común aplicarla como optimización numérica, y es parte del proceso de diseño para hallar la mejor solución al problema o requerimientos planteados [2].

Un problema de optimización tiene una función objetivo que se desea optimizar (hallar el mínimo o máximo de la misma), restricciones que limitan los valores de entrada de la función objetivo y variables de control que permiten modificar el comportamiento del sistema. Las restricciones pueden ser lineales o cuadráticas y estar definidas por ecuaciones de igualdad, desigualdad, o bien, pueden no haber restricciones de ningún tipo [2].

Es importante comprender el problema a resolver para poder escoger un método que lo resuelva de forma confiable y eficiente, pues no hay un método efectivo para todas las situaciones. Algunos factores que ayudan a determinar esto son el tipo de función objetivo y restricciones, el tamaño del problema, así como si el problema se va a resolver en forma repetida (siendo esto último importante al decidir si utilizar optimización convexa) [2].

6.2.1. Optimización convexa

Se basa en la resolución de problemas convexos, los cuales son más sencillos que otro tipo de problemas de optimización dado que no requieren puntos de inicio, ajuste de parámetros, ni derivadas. Sin embargo, los problemas convexos tienen requerimientos estrictos que deben cumplirse [2]. Dentro de la categoría de optimización convexa, se encuentran distintos tipos de problemas, incluyendo: lineales, cuadráticos, cónicos de segundo orden, disciplinados y geométricos. En cada caso, el sistema se denomina programa y el tipo de optimización se denomina programación, tal que la programación lineal abarca programas lineales, la programación cuadrática se enfoca en programas cuadráticos, y así sucesivamente [2].

6.2.2. Programación cuadrática

Los programas cuadráticos (QPs) se caracterizan por tener una función objetivo con un término cuadrático y uno lineal [11] [2]. Se definen de la siguiente manera:

$$\begin{aligned} \underset{x}{\text{minimizar}} \quad & \frac{1}{2}\mathbf{x}^\top \mathbf{Q}\mathbf{x} + \mathbf{x}^\top \mathbf{c} \\ \text{sujeto a} \quad & \mathbf{A}\mathbf{x} + \mathbf{b} = 0 \\ & \mathbf{C}\mathbf{x} + \mathbf{d} \leq 0 \end{aligned} \quad , \tag{16}$$

donde $\mathbf{Q}$, $\mathbf{A}$ y $\mathbf{C}$ son matrices, mientras que, $\mathbf{b}$ y $\mathbf{d}$ son vectores. La matriz $\frac{1}{2}\mathbf{x}^\top \mathbf{Q}\mathbf{x}$ corresponde al término cuadrático y $\mathbf{x}^\top \mathbf{c}$ al término lineal de la función a optimizar. Las ecuaciones $\mathbf{A}\mathbf{x}+\mathbf{b} = 0$ y $\mathbf{C}\mathbf{x}+\mathbf{d} \leq 0$ representan las restricciones de igualdad y desigualdad, respectivamente.

Cabe destacar que, si $\mathbf{Q} = \mathbf{0}$, el QP se reduce a un programa lineal (LP) [2]:

$$\begin{aligned} \underset{x}{\text{minimizar}} \quad & \mathbf{c}^\top \mathbf{x} \\ \text{sujeto a} \quad & \mathbf{A}\mathbf{x} + \mathbf{b} = 0 \\ & \mathbf{C}\mathbf{x} + \mathbf{d} \leq 0 \end{aligned} \quad . \tag{17}$$

Los programas lineales siempre son convexos [2].

Los programas cuadráticos son convexos únicamente si la matriz $\mathbf{Q}$ es positiva semidefinida. Es decir, $\mathbf{Q}$ debe ser tal que $\mathbf{x}^\top \mathbf{Q}\mathbf{x} \geq 0$. Es importante recalcar la diferencia entre positivo semidefinido ($\mathbf{x}^\top \mathbf{Q}\mathbf{x} \geq 0$) y positivo definido ($\mathbf{x}^\top \mathbf{Q}\mathbf{x} > 0$), ya que corresponden a condiciones diferentes: una desigualdad y una igualdad, respectivamente [2].

6.2.2.1. Restricciones por igualdad

Cuando hay restricciones por igualdad ($\mathbf{A}\mathbf{x} + \mathbf{b} = 0$), el problema se puede reescribir como un sistema de matrices llamado Karush-Kuhn-Tucker (KKT) [11], definido como:

$$\begin{bmatrix} \mathbf{G} & \mathbf{A}^\top \\ \mathbf{A} & \mathbf{0} \end{bmatrix} \begin{bmatrix} -\mathbf{p} \\ \boldsymbol{\lambda}^* \end{bmatrix} = \begin{bmatrix} \mathbf{g} \\ \mathbf{h} \end{bmatrix}, \tag{18}$$

en donde $\mathbf{h} = \mathbf{A}\mathbf{x} - \mathbf{b}$, $\mathbf{g} = \mathbf{c} + \mathbf{G}\mathbf{x}$ y $\mathbf{p} = \mathbf{x}^* - \mathbf{x}$. Adicionalmente, $\mathbf{x}^*$ es un vector que soluciona al programa cuadrático, mientras que $\boldsymbol{\lambda}^*$ es un vector de multiplicadores de Lagrange que permite cumplir esta estructura para el sistema [11]. Para solucionarlo, hay métodos directos: factorización simétrica indefinida ($\text{LDL}^\top$), método del complemento de Schur y el método de espacio nulo; así como métodos iterativos, por ejemplo el método del gradiente aplicado al sistema reducido y método del gradiente proyectado [11] [12].

Factorización positiva simétrica indefinida del sistema KKT

Este método aplica la factorización $\text{LDL}^\top$ a la matriz del sistema KKT, solucionando después el sistema lineal $\text{LDL}^\top$ [11]. El método puede resumirse en los siguientes pasos:

1. Calcular factorización $\mathbf{PAP}^\top = \mathbf{LDL}^\top$
2. Resolver $\mathbf{Lz}_1 = \mathbf{Pb}$ con sustitución hacia adelante.
3. Resolver $\mathbf{Dz}_2 = \mathbf{z}_1$ con algún método válido.
4. Resolver $\mathbf{L}^\top \mathbf{z}_3 = \mathbf{z}_2$ con sustitución hacia atrás.
5. Calcular $\mathbf{y} = \mathbf{P}^\top \mathbf{z}_3$

Con la diferencia de que, dentro de este contexto, el vector $\mathbf{y}$ es la solución al sistema KKT, en lugar de la solución del programa cuadrático como tal $\mathbf{x}$. En cambio, el vector $\mathbf{x}$ se obtiene a partir de los primeros valores de $\mathbf{y}$, correspondiente a la cantidad de filas en el vector $\mathbf{c}$ del programa cuadrático.

Solución del sistema KKT con el método del espacio nulo

En este método, se define el vector p del sistema KKT de la siguiente forma:

$$\mathbf{p} = \mathbf{Y}\mathbf{p}_y + \mathbf{Z}\mathbf{p}_z, \tag{19}$$

en donde $\mathbf{Y}$ es una matriz $(m \times n)$ y $\mathbf{Z}$ una matriz $(n \times (n-m))$ que corresponde al espacio nulo de la matriz a descomponer (en este caso, la matriz $\mathbf{A}$ de restricciones del problema cuadrático). Tanto $\mathbf{Y}$ como $\mathbf{Z}$ se pueden obtener a partir de la factorización $\mathbf{QR}$ de $\mathbf{A}$, subdividiendo $\mathbf{Q}$ de la siguiente manera:

$$\begin{aligned} \mathbf{Y} &= \mathbf{Q}(:, 1:m) \\ \mathbf{Z} &= \mathbf{Q}(:, m+1:\text{end}), \end{aligned} \tag{20}$$

donde $\mathbf{Q}(:, 1:m)$ corresponde a todas las filas y las primeras m columnas de $\mathbf{Q}$, mientras que $\mathbf{Q}(:, m+1:\text{end})$ incluye todas las filas y las columnas a partir de la columna m de $\mathbf{Q}$. Después, se obtienen los vectores $\mathbf{p}_y$ y $\mathbf{p}_z$, resolviendo los siguientes sistemas para luego calcular la solución $\mathbf{p}$:

$$\begin{aligned} (\mathbf{AY})\mathbf{p}_y &= \mathbf{b}, \\ (\mathbf{Z}^\top \mathbf{Gz})\mathbf{p}_z &= -\mathbf{Z}^\top \mathbf{GY}\mathbf{p}_y - \mathbf{Z}^\top \mathbf{c}. \end{aligned} \tag{21}$$

6.2.2.2. Restricciones por desigualdad

En el caso de restricciones por desigualdad ($\mathbf{Ax} + \mathbf{b} \leq \mathbf{0}$), se implementan principalmente dos métodos: de conjunto activo y de punto interior. Estos son iterativos, por lo que es más adecuado implementarlos cuando los problemas son muy grandes [11]. El tipo de problema a resolver determina qué método es más adecuado, ya que ambos difieren en varias características:

- Los métodos de conjunto activo requieren una gran cantidad de pasos de bajo costo computacional, mientras que los de punto interior necesitan menos pasos, pero cada uno tiene un costo computacional elevado.

- Para problemas demasiado grandes, los métodos de punto interior son generalmente más eficientes, pero si se tiene un estimado inicial de la solución del problema, los métodos de conjunto activo podrían resolverlo en pocas iteraciones. Por el contrario, los métodos de punto interior no son tan capaces de aprovechar valores iniciales.
- Los métodos de conjunto activo son más complicados de implementar debido a las factorizaciones que se deben realizar, mientras que los métodos de punto interior tienen una implementación más sencilla.

6.2.2.3. Métodos de conjunto activo

Se basan en resolver un subproblema formulado a partir del problema quadrático original, escogiendo algunas de las restricciones como parte del conjunto activo y descartando otras para determinar a qué soluciones está sujeta la solución más óptima [11]. A partir de esto, se realizan distintas iteraciones hasta encontrar la respuesta que corresponda al mínimo de la función objetivo [11]. El procedimiento general es el siguiente:

1 Definir el conjunto activo inicial, escogiendo todas las restricciones por igualdad y alguna de las restricciones por desigualdad (pero escrita como una igualdad). Esto define el subproblema a resolver.

2 Aproximar la solución del subproblema en la iteración actual.

3 Calcular los multiplicadores de Lagrange de la solución obtenida.

4 Si hay algún multiplicador de Lagrange negativo, excluir la restricción correspondiente al multiplicador más negativo. Luego, calcular luego la siguiente iteración con las restricciones escogidas.

5 En caso de que todos los multiplicadores de Lagrange sean positivos, esa solución corresponde a la más óptima.

6.2.2.4. Métodos de punto interior

En este caso, se reescriben las condiciones KKT del sistema a resolver, de manera que las restricciones por desigualdad se convierten en igualdades [11]. Después, se resuelve el problema por medio del método de Newton, obteniendo un sistema lineal (sistema primal-dual), el cual se resuelve por algún otro método para obtener la solución al problema [11]. El procedimiento general es el siguiente:

1 Escribir las condiciones KKT del sistema como igualdades y desigualdades. Luego, introducir variables de holgura para convertir las desigualdades en igualdades.

2 Definir las condiciones KKT con perturbación del sistema, esto establece la trayectoria a seguir para calcular la solución.

3 Aplicar el método de Newton y reescribir como un sistema lineal (sistema primal-dual).

4 Resolver el sistema primal-dual obtenido. Esto se puede hacer de distintas formas, por ejemplo un algoritmo Cholesky o método del gradiente conjugado. Utilizar un método iterativo es útil cuando no es posible aplicar una factorización directa.

6.2.3. Complejidad de los algoritmos y costo computacional

Un aspecto crítico de la computación científica es el costo computacional de los algoritmos, ya sea en el tiempo de operación (cuánto tardan en realizarse) o en el uso de memoria. Para indicar este, se utiliza la notacíon o perspectiva *Big-Oh* (por su nombre en inglés), por medio de la cual se denota, por ejemplo, $O(n)$ una operación con un costo de n flops, siendo cada flop una operación flotante de suma, resta, división o multiplicación [12]. Sin embargo, esto no toma en consideración otros pasos como tráfico de memoria, acceso de elementos en las matrices, etc. por lo que si bien proporciona una idea del costo computacional, no es una medida del todo concluyente [12]. Por ejemplo, una operación de permutación no involucra flops, sin embargo, sí incrementa el tiempo de operación total [12].

6.3. Algoritmos de control y robótica de importancia

El álgebra lineal es fundamental para distintos algoritmos de control y robótica, tanto para cálculos de pose, optimización de trayectorias, controladores, etc. Dependiendo del tema, pueden requerirse operaciones matriciales básicas o algoritmos más complejos como programación cuadrática. En esta sección se describen brevemente los temas principales de control y robótica a implementar en este trabajo de graduación, como base para explicar las funciones desarrolladas para esto.

6.3.1. Controlador PID

Una forma común de implementar un sistema de control con retroalimentacón es por medio del controlador PID, que tiene compensación proporcional (P), integral (I) y derivativa (D) [14]. Este tipo de controlador se puede describir de la siguiente manera:

$$C(s) = k_P + \frac{k_I}{s} + k_D s, \tag{22}$$

en donde C(s) es el controlador, mientras que k_P, k_I y k_D son los coeficientes proporcional, integral y derivativo, respectivamente. Además, s es la frecuencia compleja del sistema. También se puede implementar un filtrado o ajuste en los términos integral o derivativo para reducir el error aún más en estos términos. Por ejemplo, para un ajuste en el término derivativo:

$$C(s) = k_P + \frac{k_I}{s} + \frac{k_D s}{\tau s + 1}. \tag{23}$$

Para implementar un controlador PID en un dispositivo digital es necesario implementar ecuaciones de diferencias para operar en tiempo discreto, para lo cual se sustituye el término continuo s por el término discreto z [14] [15]. Para esto, existen distintos métodos, siendo los más utilizados los siguientes:

$$s = \frac{z-1}{T}, \tag{24a}$$

$$s = \frac{z-1}{Tz}, \tag{24b}$$

$$s = \frac{2}{T}\frac{z-1}{z+1}. \tag{24c}$$

que corresponden a las reglas rectangular hacia adelante (24a), rectangular hacia atrás (24b) y trapezoidal o Tustin (24c) [14] [15].

6.3.2. Sistemas lineales invariantes en el tiempo (LTI)

Para un sistema LTI, el planteamiento general en tiempo continuo es el siguiente:

$$\begin{aligned} \dot{\mathbf{x}} &= \mathbf{f}(\mathbf{x}, \mathbf{u}), \\ \mathbf{y} &= \mathbf{h}(\mathbf{x}, \mathbf{u}), \end{aligned} \tag{25}$$

en donde $\mathbf{x}$, $\dot{\mathbf{x}}$ y $\mathbf{u}$ corresponden al estado actual, respectivamente, estado futuro y entrada actual del sistema, $\mathbf{f}(\mathbf{x}, \mathbf{u})$ es la función que representa la dinámica o comportamiento del sistema y $\mathbf{h}(\mathbf{x}, \mathbf{u})$ es la función para calcular la salida del sistema, siendo $\mathbf{y}$ el resultado obtenido para la salida [15].

Similar al PID, los sistemas LTI se pueden discretizar empleando las mismas sustituciones (24a-24c) [14] [15]. A partir de esto, se obtiene la siguiente estructura para el sistema LTI en tiempo discreto:

$$\begin{aligned} \mathbf{x}_{k+1} &= \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k, \\ \mathbf{y}_k &= \mathbf{C}\mathbf{x}_k + \mathbf{D}\mathbf{u}_k, \end{aligned} \tag{26}$$

en donde, $\mathbf{x}_k$ y $\mathbf{x}_{k+1}$ corresponden al estado actual y futuro del sistema, respectivamente, mientras que $\mathbf{u}_k$ y $\mathbf{y}_k$ corresponden a los vectores de entrada y salida. Por su parte, $\mathbf{A}$ es una matriz cuadrada $(n \times n)$, $\mathbf{B}$ es una matriz $(n \times m)$ y $\mathbf{C}$ una matriz $(m \times n)$ [15].

6.3.3. Simulación de sistemas no lineales

Los sistemas no lineales tienen una mayor complejidad comparado con los sistemas lineales. Debido a esto, suelen simular por medio de métodos de integración numérica para aproximar el comportamiento del sistema por medio de un sistema lineal [15]. Dependiendo del tipo de sistema no lineal, se pueden utilizar distintos métodos de integración, siendo la integración rectangular hacia adelante y el método de Runge-Kutta de orden cuatro (abreviado como Runge-Kutta4 o RK4) métodos que pueden utilizarse [15] [16]. Al operar con estos métodos, la linealización del sistema no lineal queda de la siguiente manera:

- Simulación con integración rectangular hacia adelante:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + f(\mathbf{x}_k)\Delta t, \tag{27}$$

- Simulación con integración por Runge-Kutta4:

$$\begin{aligned}
\mathbf{x}_{k+1} &= \mathbf{x}_k + \frac{\Delta t}{6}(k1 + 2k_2 + 2k_3 + k_4), \\
k_1 &= \mathbf{f}(\mathbf{x}_k), \\
k_2 &= \mathbf{f}\left(\mathbf{x}_k + \frac{\Delta t}{2}k_1\right), \\
k_3 &= \mathbf{f}\left(\mathbf{x}_k + \frac{\Delta t}{2}k_2\right), \\
k_4 &= \mathbf{f}(\mathbf{x}_k + \Delta t k_3),
\end{aligned} \tag{28}$$

en donde $\mathbf{x}_k$ es el estado actual del sistema, $\mathbf{x}_{k+1}$ es el siguiente estado y Δt es el tiempo de muestreo. La función f corresponde al modelo o dinámica que representa al sistema no lineal [15].

6.3.4. Control basado en retroalimentación de estado

Un sistema LTI se puede estabilizar en un punto de operación específico ($\mathbf{u}_{ss}$,$\mathbf{x}_{ss}$), añadiendo además una matriz de ganancia de control $\mathbf{K}$, con lo cual se representa la retroalimentación de estado del sistema [15].

$$\mathbf{u} = \mathbf{K} \times (\mathbf{x}_{ss} - \mathbf{x}) + \mathbf{u}_{ss}, \tag{29}$$

6.3.5. Modelo de control predictivo (MPC)

El modelo de control predictivo (*Model Predictive Control* o MPC, en inglés) es una estrategia de control que permite estimar estados futuros de un sistema dadas las mediciones obtenidas en el estado actual, optimizando así el costo de solucionar un problema a partir de una predicción de los valores futuros [17]. Para esto, se implementa un horizonte móvil N, es decir, que en cada estado se estiman N valores futuros de la variable de decisión [17].

El MPC clásico se implementa en sistemas lineales invariantes en el tiempo (LTI) y tiene que plantearse como un programa cuadrático para su solución, por lo que se pueden implementar métodos de programación cuadrática, por ejemplo, de conjunto activo o punto interior [17].

Para construir un programa cuadrático a partir del sistema LTI, se necesitan las siguientes matrices:

$$\mathbf{M}_x = \begin{bmatrix} \mathbf{\Phi} \\ \mathbf{\Phi}^2 \\ \vdots \\ \mathbf{\Phi}^N \end{bmatrix}, \qquad \mathbf{M}_c = \begin{bmatrix} \mathbf{B} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{\Phi B} & \mathbf{B} & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{\Phi}^{N-1}\mathbf{B} & \Phi^{N-2}\mathbf{B} & \cdots & \mathbf{B} \end{bmatrix}, \tag{30}$$

en donde $\mathbf{\Phi} = \mathbf{A} + \mathbf{BK}$, siendo $\mathbf{K}$ una matriz de ganancia para retroalimentación de estado, por lo que si $\mathbf{K} = \mathbf{0}$, eso significa que $\mathbf{\Phi} = \mathbf{A}$. Adicionalmente, se requieren matrices de penalización del costo $\mathbf{Q}$ y $\mathbf{R}$ que, junto a $\mathbf{M}_x$ y $\mathbf{M}_c$, se emplean para generar las matrices del costo del problema cuadrático convexo [17].

El problema cuadrático del MPC puede ser sin restricciones o tener restricciones lineales para las variables de decisión, las cuales pueden ser el vector de entradas $\mathbf{u}_k$ o el vector de estados $\mathbf{x}_k$ a optimizar [17]. En caso de utilizar únicamente $\mathbf{u}_k$ como la variable de decisión, se pueden introducir de la siguiente manera:

$$\begin{bmatrix} lb \\ \vdots \\ lb \end{bmatrix} \leq \mathbf{u}_k \leq \begin{bmatrix} ub \\ \vdots \\ ub \end{bmatrix}, \tag{31}$$

en donde las restricciones aplican únicamente a las variables de decisión $\mathbf{u}_k$. También se pueden implementar de la siguiente forma, en donde las restricciones para $\mathbf{x}_k$ se introducen por medio de la desigualdad para las variables de decisión $\mathbf{u}_k$:

$$\mathbf{x}_{lb} - \mathbf{B}\mathbf{u}_1 \leq \mathbf{A}\mathbf{u}_k \leq \mathbf{x}_{ub} - \mathbf{B}\mathbf{u}_1, \tag{32}$$

6.3.6. Filtro de Kalman

El Filtro de Kalman es un estimador óptimo que permite calcular el siguiente estado de un sistema lineal compensando el efecto del ruido, por medio de un proceso de predicción y corrección del siguiente estado [18]. La predicción involucra el valor previo del estado y de las entradas que se utilizaron, mientras que la corrección utiliza nueva información (llamada la innovación) obtenida a partir de las mediciones de los sensores del sistema [18].

Para un sistema lineal invariante en el tiempo discreto:

$$\begin{aligned} \mathbf{x}_{k+1} &= \mathbf{F}\mathbf{x}_k + \mathbf{G}\mathbf{u}_k + \mathbf{v}_k, \\ \mathbf{z}_{k+1} &= \mathbf{H}\mathbf{x}_k + \mathbf{w}_k, \end{aligned} \tag{33}$$

las ecuaciones para la predicción son las siguientes:

$$\begin{aligned} \hat{\mathbf{x}}_{k+1|k} &= \mathbf{F}\hat{\mathbf{x}}_k + \mathbf{G}\mathbf{u}_k, \\ \hat{\mathbf{P}}_{k+1|k} &= \mathbf{F}\hat{\mathbf{P}}_{k|k}\mathbf{F}^\top + \hat{\mathbf{V}}, \end{aligned} \tag{34}$$

mientras que las ecuaciones para la corrección son:

$$\begin{aligned} \boldsymbol{\nu}_{k+1} &= \mathbf{z}_{k+1} - \mathbf{H}\hat{\mathbf{x}}_{k+1|k}, \\ \hat{\mathbf{x}}_{k+1|k+1} &= \hat{\mathbf{x}}_{k+1|k} + \mathbf{K}_{k+1}\boldsymbol{\nu}_{k+1}, \\ \hat{\mathbf{P}}_{k+1|k+1} &= \hat{\mathbf{P}}_{k+1|k} - \mathbf{K}_{k+1}\mathbf{H}\hat{\mathbf{P}}_{k+1|k}, \end{aligned} \tag{35}$$

en donde $\hat{\mathbf{x}}$ y $\hat{\mathbf{P}}$ son el vector de estados y la covarianza estimada generados por el Filtro de Kalman, $\mathbf{w}$ y $\mathbf{v}$ corresponden al ruido introducido al sistema y $\boldsymbol{\nu}$ es la innovación calculada para el próximo estado [18].

6.3.7. Marcos de referencia y transformaciones homogéneas

Para representar la posición y orientación de un robot, se utilizan marcos de referencia con ejes en dos o tres dimensiones, dependiendo de la aplicación. Sin embargo, suele utilizarse más de un marco, uno como global y uno o más en distintas partes del robot, por ejemplo, robots móviles o en un brazo robot con múltiples articulaciones (cada uno con su posición y orientación) [18]. A partir de esto, se implementan matrices de transformación homogénea, para trasladarse de un marco a otro, permitiendo calcular la pose de cualquier parte del robot a través de otros marcos [18].

Una transformación homogénea se define como:

$$ {}^{A}\mathbf{p} = {}^{A}\,\boldsymbol{\xi}_{B}\,.{}^{B}\,\mathbf{p}, \tag{36} $$

en donde los vectores ${}^{A}\mathbf{p}$ y ${}^{B}\mathbf{p}$ son puntos de coordenadas ubicados con respecto de dos marcos de referencia diferentes (en este caso marcos A y B, respectivamente), mientras que ${}^{A}\boldsymbol{\xi}_{B}$ es la matriz de transformación homogénea para pasar del marco de referencia B al marco A:

$$ {}^{A}\boldsymbol{\xi}_{B} = \begin{bmatrix} {}^{T}\mathbf{R}_{B} & \mathbf{t} \\ \mathbf{0}_{1x2} & \mathbf{1} \end{bmatrix} = \begin{bmatrix} R_{11} & R_{12} & R_{13} & x \\ R_{21} & R_{22} & R_{23} & y \\ R_{31} & R_{32} & R_{33} & z \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{37} $$

en donde $\mathbf{t}$ es un vector de coordenadas y $\mathbf{R}$ es una matriz de rotación, cuya forma depende del eje con respecto del cual se realiza la rotación [18].

$$ \mathbf{R}_{x} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}, \quad \mathbf{R}_{y} = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}, \quad \mathbf{R}_{z} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{38} $$

Las matrices de rotación se pueden generar a partir de un ángulo θ, por medio de las secuencias de ángulos de Euler: *XYX*, *XZX*, *YXY*, *YZY*, *ZXZ*, *ZYZ*, ángulos de Cardán: *XYZ*, *XZY*, *YXZ*, *YZX*, *ZXY*, *ZYX*, o a partir de cuaterniones [18] [19].

6.3.8. Cuaterniones unitarios para representar orientación

Los cuaterniones se definen de la siguiente manera:

$$ \mathring{q} = s + \mathbf{v}, \tag{39} $$

en donde s es una cantidad escalar y un vector de tres dimensiones $\mathbf{v} = v_1\mathbf{i} + v_2\mathbf{j} + v_3\mathbf{k}$ que corresponde a la parte compleja del cuaternión [18].

Los cuaterniones se emplean en las áreas de robótica, visión por computadora, gráficos de computadora y para aplicaciones de navegación aeroespacial [18]. No obstante, para

representar rotaciones se utilizan cuaterniones unitarios, dado que en esta forma es posible relacionar los componentes del cuaternión unitario con la magnitud y orientación de un ángulo de rotación θ [18]. Esta relación se describe por medio de las siguientes ecuaciones:

$$s = \cos\frac{\theta}{2}, \quad \mathbf{v} = \left(sin\frac{\theta}{2}\right)\hat{\mathbf{n}} \tag{40}$$

en donde el vector unitario $\hat{\mathbf{n}}$ corresponde a la dirección de la rotación con el ángulo θ.

Entonces, para convertir un punto de coordenadas de un marco de referencia a otro, se realiza la siguiente operación:

$$^{A}\mathbf{p} = \mathring{q} \cdot {}^{B}\mathbf{p} + \mathbf{t}, \tag{41}$$

siendo $^{A}\mathbf{p}$ y $^{B}\mathbf{p}$ puntos de coordenadas asociados a los marcos de referencia A y B, respectivamente, mientras que $\mathbf{t}$ es un vector de coordenadas [18].

6.4. Dispositivos embebidos relevantes

A continuación se incluye un cuadro comparativo con las características más relevantes de los dispositivos embebidos empleados en este trabajo [20] [21] [22]. Los dispositivos escogidos tienen distintas architecturas, de entre 8 y 32 bits acorde a las características de la librería a evaluar.

Cuadro 1. Características relevantes de los dispositivos embebidos a utilizar

Dispositivo	Memoria RAM	Memoria FLASH	Arquitectura
ESP32	520KB	4MB	Xtensa 32-Bits
Arduino MEGA	8KB	256KB	AVR 8-Bits
STM NUCLEO F446RE	128KB	512KB	Arm 32-Bits

Nota. Elaboración propia.

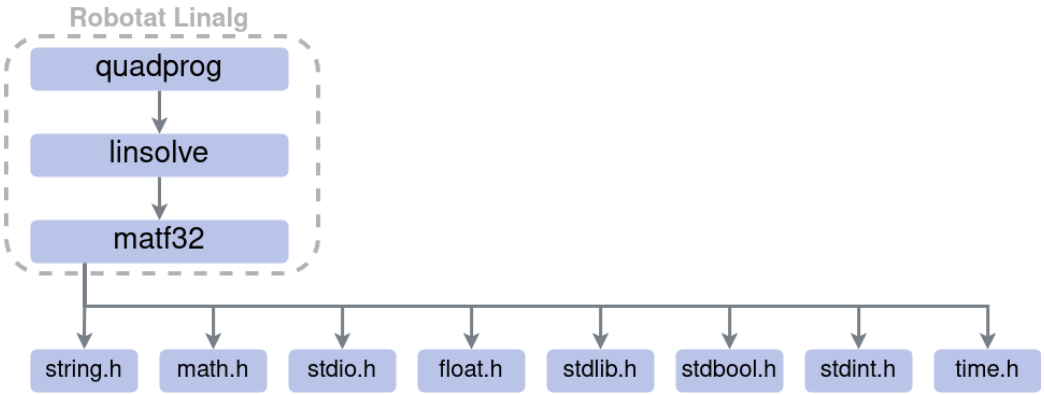
CAPÍTULO 7

Implementación y evaluación del rendimiento de las librerías de Robotat Linalg en distintos dispositivos

La librería Robotat Linalg está conformada por tres librerías de computación numérica: `matf32` para álgebra lineal, `linsolve` para resolver sistemas lineales y `quadprog` para programación cuadrática. Estas emplean únicamente las rutinas propias del lenguaje C, es decir, sin librerías externas, acorde a los requerimientos de una librería embebida de optimización [5]. En la Figura 1 se muestra el diagrama de dependencias de Robotat Linalg.

El desarrollo de estas librerías se llevó a cabo en un trabajo de graduación previo de la UVG, por lo que en el presente trabajo se implementaron las rutinas pendientes de cada una y se evaluó el rendimiento de estas. En este capítulo se presentan los cambios realizados en cada librería comparado con la versión anterior, seguido de los resultados de su evaluación.

Figura 1. Diagrama de dependencias de las librerías de Robotat Linalg



Nota. Elaboración propia.

7.1. Metodología para la evaluación de las funciones

7.1.1. Plataformas empleadas y funciones evaluadas

A continuación se resume el procedimiento de evaluación empleado para todas las librerías de Robotat Linalg. Cada una se evaluó en los siguientes dispositivos embebidos: ESP32, Arduino MEGA y STM NUCLEO F446RE, y también una computadora con arquitectura amd-64 y sistema operativo Linux Mint 21.3 como referencia del rendimiento de las librerías en un dispositivo con mayor capacidad. Se empleó MATLAB para comparar la exactitud numérica y el tiempo de operación de cada función, ya que MATLAB se utiliza en una computadora con mayor capacidad pero que realiza múltiples procesos a la vez, en contraste con dispositivos embebidos que se enfocan en tareas específicas.

Para cada librería se evaluaron las funciones principales, es decir: los métodos en `quadprog` y en `linsolve`, y las operaciones en estructuras `matf32_t` con las cuales se definen las matrices de `matf32`. Las operaciones que involucran la SVD no tienen equivalente directo en MATLAB, por lo que se implementó el algoritmo correspondiente en este último para comparar su rendimiento. Por el contrario, se empleó el método de conjunto activo de MATLAB para comparar todos los métodos de `quadprog`, ya que el enfoque fue evaluar contra el solucionador propio de MATLAB pero la mayoría no tienen equivalente en este último. En el Cuadro 2 se presenta un resumen las funciones evaluadas y el equivalente empleado.

Cuadro 2. Funciones evaluadas en Robotat Linalg y MATLAB

Robotat Linalg	**Matlab**	**Operación**
matf32_add	A+A	Suma de matrices
matf32_sub	A-A	Resta de matrices
matf32_scale	A*c	Multiplicación matriz-escalar
matf32_trans	A'	Transpuesta de una matriz
matf32_mul	A*A	Multiplicación matriz-matriz
matf32_inv	inv	Inversa de una matriz
matf32_pinv	pinv	Pseudoinversa de una matriz
matf32_dot	dot	Producto punto entre dos vectores
matf32_vecposmul	rowvec*A	Multiplicación vector-matriz
matf32_vecpremul	A*colvec	Multiplicación matriz-vector
matf32_vecmul_col_row	rowvec*colvec	Multiplicación vector-vector
matf32_arr_add	A+A+A	Suma de un arreglo de matrices
matf32_arr_sub	A-A-A	Resta de un arreglo de matrices
matf32_arr_mul	A*A*A	Multiplica un arreglo de matrices
matf32_exp	A^c	Potencia de una matriz cuadrada
matf32_cholesky	chol	Factorización Cholesky
matf32_qr	qr	Factorización QR
matf32_lu	lu	Factorización LU
matf32_jacobi_svd	one_sided_jacobi_svd**	Factorización SVD con Jacobi
linsolve (distintos métodos)	linsolve (\)	Solucionador lineal
quadprog (distintos métodos)	quadprog (con active-set)	Solucionador cuadrático

Nota. **Funciones creadas en MATLAB con el mismo algoritmo empleado en Robotat Linalg. Notación: **A** es una matriz y **rowvec** y **colvec** vectores fila y columna y c es un número escalar. Elaboración propia.

7.1.2. Procedimiento para evaluar el tiempo de operación de las funciones

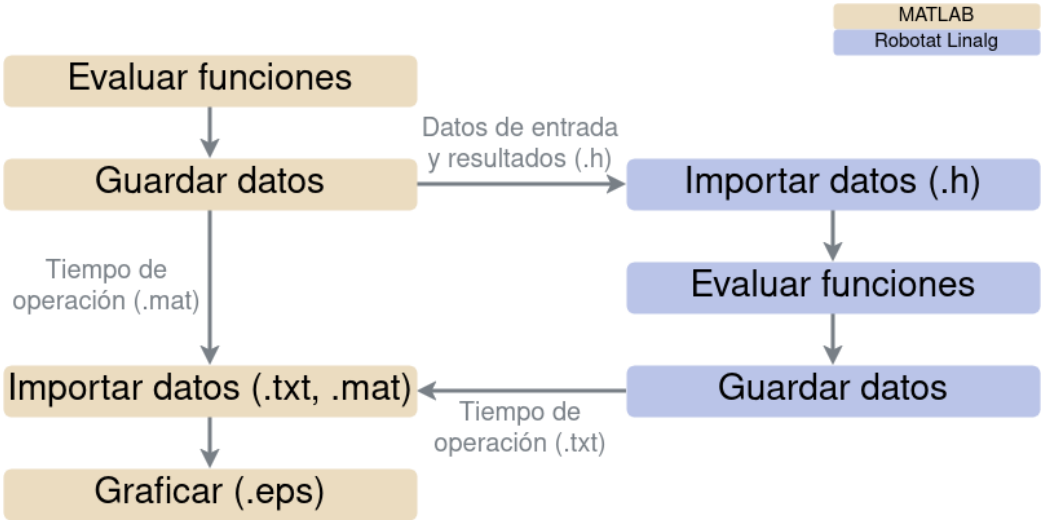
Cada función se ejecutó un total de 100 veces seguidas, midiendo el tiempo antes de la primera operación y después de la última, luego dividiendo entre 100 para generar un estimado del tiempo promedio de operación. Esto se repitió para distintos tamaños de matrices: desde 1×1 hasta 10×10 para la mayoría de operaciones para `matf32` (excepto las factorizaciones que se midieron con matrices desde 2×2 hasta 10×10) y desde 2×2 hasta 10×10 en `linsolve`. En el caso de `quadprog`, las dimensiones se manejaron de forma distinta por la estructura de las operaciones, por lo que esto se explica en la sección correspondiente a esta librería. Luego, se graficaron los resultados de tiempo contra la dimensión de la matriz, exportando las mediciones de tiempo en cada plataforma (MATLAB, ESP32, etc.) y utilizando MATLAB para crear las gráficas, como se muestra en la Figura 2.

La expectativa en cuanto al tiempo de operación es que las funciones de las librerías de Robotat Linalg sean más rápidas de ejecutar que sus equivalentes en MATLAB. Sin embargo, al mismo tiempo es de esperar que los dispositivos embebidos requieran más tiempo para realizar algunas operaciones, dado que tienen menor capacidad que una computadora. A partir de esto, para definir una métrica general con la cual determinar si el tiempo de operación de las funciones es eficiente o lento, primero se escogió el tiempo promedio máximo que se midió para cada función, generando así una lista de las funciones con mayor costo en tiempo en cada plataforma (Anexos 23-27). De esta lista, se identificó que el tiempo máximo medido en MATLAB estuvo en orden de milisegundos, por lo que cualquier tiempo de operación en Robotat Linalg en milisegundos (1×10^{-3}) o menor se consideró eficiente, y por el contrario, si supera este orden de magnitud (es decir, 1×10^{-2} segundos o superior) se consideró que la operación es lenta, esto para cada dispositivo embebido.

7.1.3. Procedimiento para evaluar el resultado numérico de las funciones

Para cada operación, las matrices, vectores, escalares y resultados numéricos obtenidos en MATLAB se exportaron a un documento *header* para importar y utilizar los mismos datos al evaluar las funciones de `matf32`, `linsolve` y `quadprog`. Esto permitió comparar los resultados numéricos entre las distintas plataformas para estimar su precisión numérica (Figura 2). Para estimar la exactitud de los resultados numéricos, se comparó la cantidad de decimales de los resultados con estas librerías contra el resultado de la operación respectiva en MATLAB. Se utilizó además una tolerancia de 1×10^{-5} como métrica deseada, es decir, se consideró un resultado totalmente correcto al coincidir por lo menos cinco decimales con MATLAB, tomando en cuenta que la precisión límite de los números flotantes es alrededor de seis o siete decimales (error en orden de 1×10^{-7} o 1×10^{-8}) [23]. En el caso de las matrices, esto se realizó empleando la función `matf32_is_equal`, que compara todos los elementos de una matriz con base en una tolerancia, mientras que para los números escalares se utilizó la función `fabs` de C, que calcula el valor absoluto de la diferencia entre dos números flotantes.

Figura 2. Diagrama de flujo del envío de archivos entre plataformas



Nota. Elaboración propia.

7.1.4. Procedimiento para evaluar el uso de memoria de los dispositivos embebidos

El uso de memoria en los dispositivos embebidos es un aspecto crítico de la implementación de Robotat Linalg, ya que el propósito es construir aplicaciones complejas a partir de esta librería, por lo que al emplear estas funciones debe quedar suficiente espacio disponible para otros algoritmos. El código de Robotat Linalg se trabajó en PlatformIO, el cual proporciona un estimado de la memoria RAM y FLASH empleadas al compilar un programa, por lo que registraron estos datos para los archivos utilizados en la evaluación de las funciones en cada dispositivo. Cabe resaltar que, para la evaluación de memoria, se excluyeron los datos de los resultados numéricos de MATLAB para no ocupar más espacio del necesario en el archivo para la evaluación. Por esto mismo, los estimados de memoria corresponden únicamente a ejecutar las funciones para todas las matrices correspondientes (por ejemplo, de 1×1 o 2×2 hasta 10×10 para matf32, o según correspondan las dimensiones).

7.2. Librería `matf32`: álgebra lineal

7.2.1. Cambios realizados a los archivos de la librería

Originalmente, esta librería se encontraba dividida en cuatro archivos para agrupar los distintos tipos de funciones: `math_util`, `matf32_math`, `matf32_check` y `matf32_def`. En este trabajo, se unificaron en una única librería llamada `matf32`, simplificando su organización y uso. Adicionalmente, se trasladaron hacia `matf32` las factorizaciones de matrices que estaban previamente en `linsolve` (Cholesky, LU y QR), separando así las factorizaciones como tal de sus aplicaciones específicas y también para utilizarlas en otras operaciones de `matf32`

donde sea necesario, sin crear dependencia de `matf32` hacia `linsolve`.

7.2.2. Funciones básicas y auxiliares

Se corrigió la implementación de la función `ones` y se añadieron dos nuevas funciones, como se describe a continuación:

- `ones`: asigna el valor de 1 a todos los elementos en un arreglo de tipo flotante. Se sustituyó la implementación con la función `memset` propia de C, utilizando en su lugar un ciclo *for* para asignar manualmente los valores, ya que `memset` no funcionaba adecuadamente para este propósito en específico.
- `matf32_check_symposdef`: implementa la factorización Cholesky para determinar si una matriz es simétrica positiva definida, ya que esta operación se realiza éxitosamente solo en este tipo de matrices. Por eso mismo, en `linsolve` se emplea esta función como condición para determinar si se puede utilizar la factorización Cholesky para solucionar sistemas lineales.
- `matf32_cond`: calcula el número de acondicionamiento de una matriz. Utilizada como herramienta de diagnóstico cuando se considere útil revisar el acondicionamiento de una matriz y también para la condición de activación de la solución de sistemas lineales por medio de la SVD en `linsolve`.

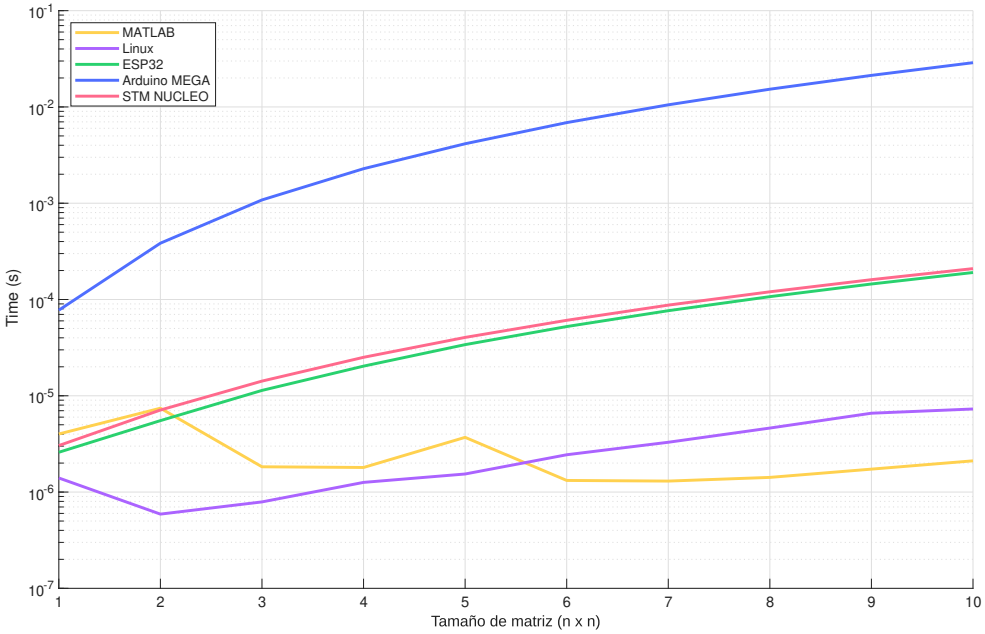
7.2.3. Operaciones matriciales básicas

En las siguientes funciones: `matf32_add`, `matf32_sub`, `matf32_scale`, `matf32_trans`, `matf32_vecposmul`, `matf32_vecpremul`, `matf32_vecmul_col_row`, `matf32_dot`, `matf32_arr_add`, `matf32_arr_sub`, se obtuvieron resultados dentro de lo esperado, con exactitud numérica igual o superior a cinco decimales y tiempos de operación iguales o menores al orden de milisegundos, en cada plataforma. Por esto mismo, los cuadros y resultados de tiempo para estas funciones se incluyen en los Anexos 28-42, mientras que a continuación se presentan las gráficas de las funciones que tuvieron tiempos de operación mayores.

`matf32_inv`

Esta función calcula la inversa de una matriz, para lo cual realiza múltiples operaciones incluyendo una transpuesta e incluso un algoritmo de sustituciones hacia adelante y hacia atrás (distintos de los implementados en `linsolve`) y una factorización LU (diferente de la empleada en `matf32_lu`). Se identificó que esta función tiende a acumular error con matrices con mayor acondicionamiento o que se aproximan a ser singulares. En cuanto al tiempo de operación, en el Arduino MEGA superó el orden de milisegundos a partir de las matrices 6×6, por lo que en estos casos se consideró lento, mientras que el resto de dispositivos tuvieron un tiempo en milisegundos o menor, dentro de lo esperado.

Figura 3. Evaluación del tiempo de operación de matf32_inv



Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

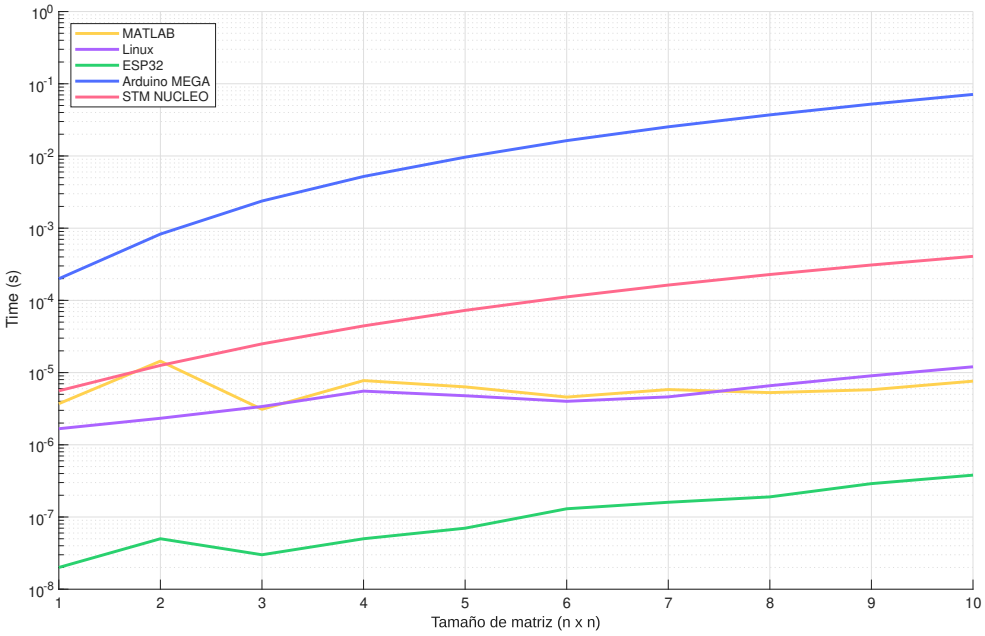
`matf32_pinv`

Como alternativa al uso de `matf32_inv`, especialmente en casos donde la matriz no tiene inversa o un mal acondicionamiento pueda generar un mayor error al generar esta misma, se agregó la función (`matf32_pinv`) para calcular la pseudoinversa de una matriz. Se implementaron los siguientes métodos, de los cuales se puede seleccionar cuál utilizar por medio de los argumentos de la función:

- La ecuación: $\mathbf{A}^{+} = (\mathbf{A}^{\top}\mathbf{A})^{-1}\mathbf{A}^{\top}$ [10].
- Por medio de las matrices de la SVD: $\mathbf{A}^{+} = \mathbf{V}\mathbf{\Sigma}^{-1}\mathbf{U}^{\top}$ [10] [12].

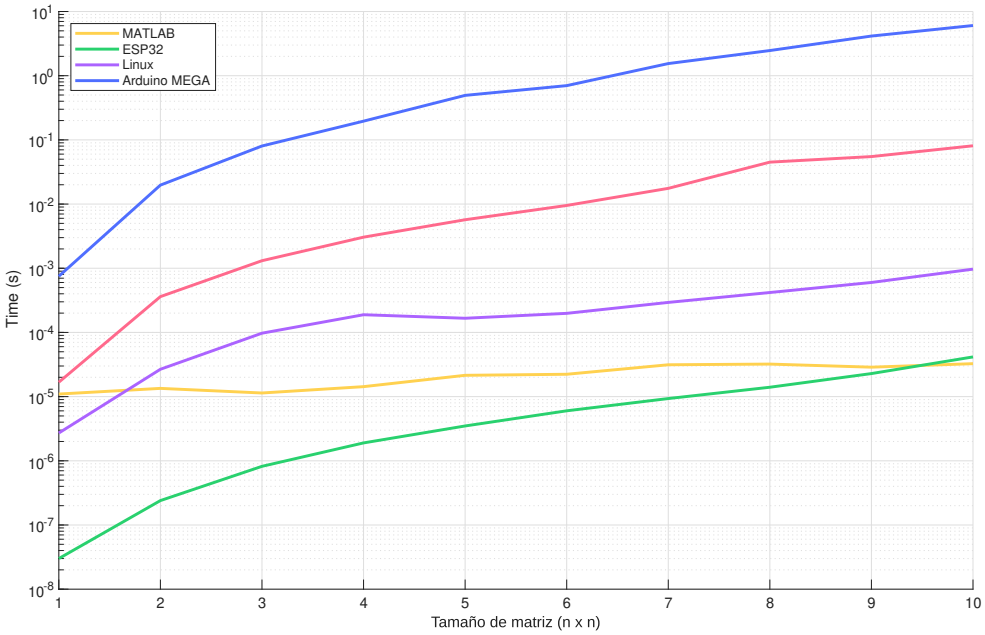
No obstante, calcular la pseudoinversa por medio de la SVD se consideró impráctico en el Arduino MEGA, dado que superó su tiempo de operación superó el orden de milisegundos a partir de matrices 2×2, por lo que es preferible aplicar el método con la ecuación $\mathbf{A}^{+} = (\mathbf{A}^{\top}\mathbf{A})^{-1}\mathbf{A}^{\top}$ en este dispositivo (y únicamente para matrices 5×5 o menores, dado el tiempo de operación resultante), como se puede observar en las Figuras 4 y 5. De forma similar, en la STM NUCLEO se consideró lento calcular la pseudoinversa con la SVD a partir de matrices 7×7, por lo que es preferible emplear el otro método en estos casos (Figura 5).

Figura 4. Evaluación del tiempo de operación de matf32_pinv empleando la ecuación $A^{+} = (A^{\top}A)^{-1}A^{\top}$



Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

Figura 5. Evaluación del tiempo de operación de matf32_pinv empleando la SVD

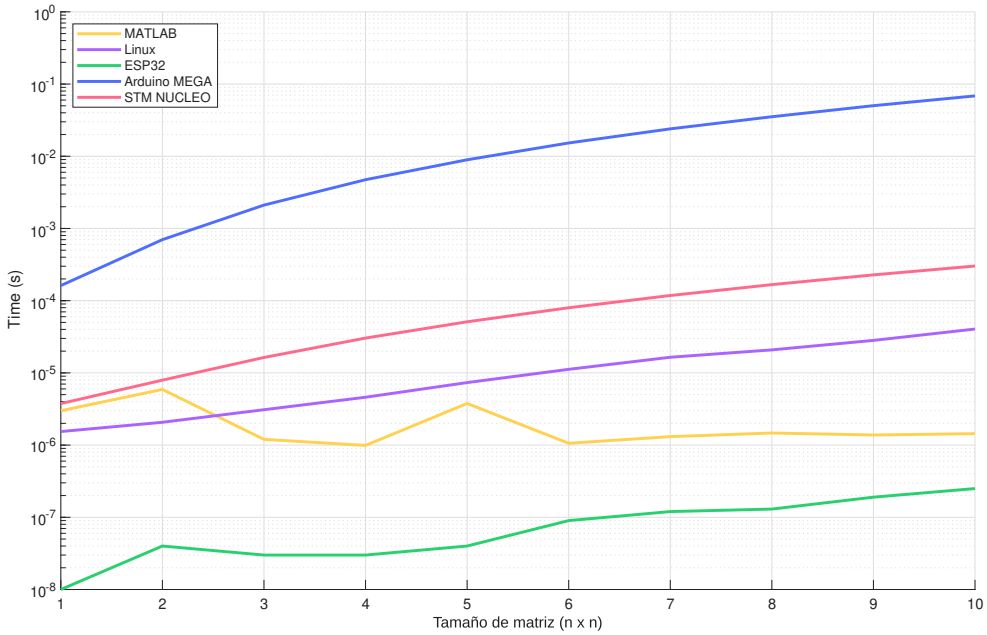


Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

`matf32_exp`

Esta función se agregó para calcular la potencia de una matriz cuadrada, ingresando el exponente en los argumentos de la función. Se validó su funcionamiento con distintos exponentes, sin embargo, el tiempo de operación y uso de memoria se evaluaron únicamente con un exponente igual a cuatro, dado que este en específico se empleó después en la librería de control para el control de modelo predictivo (Capítulo 8). El único dispositivo que superó el tiempo de operación de milisegundos fue el Arduino MEGA para matrices a partir de 6×6. Por otro lado, dada la naturaleza de esta operación, se puede interpretar que el tiempo de operación incrementará conforme se aumente el exponente empleado, por lo que el Arduino MEGA podría no ser adecuado si se requiere calcular potencias con exponentes aún mayores, y también es algo que debe considerarse para su uso en otros dispositivos (Figura 6).

Figura 6. Evaluación del tiempo de operación de matf32_exp

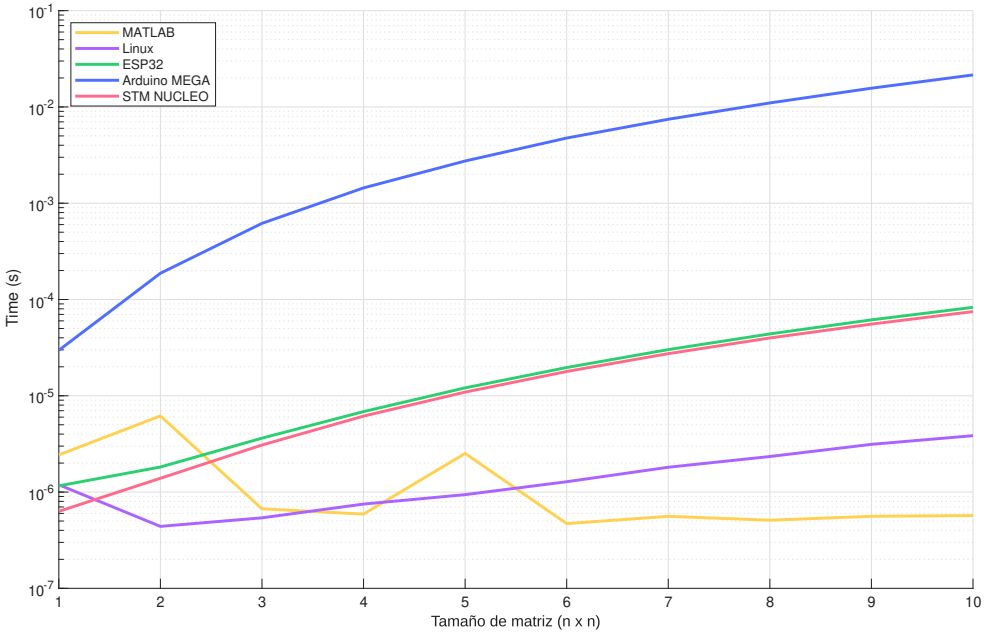


Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

`matf32_mul` **y** `matf32_arr_mul`

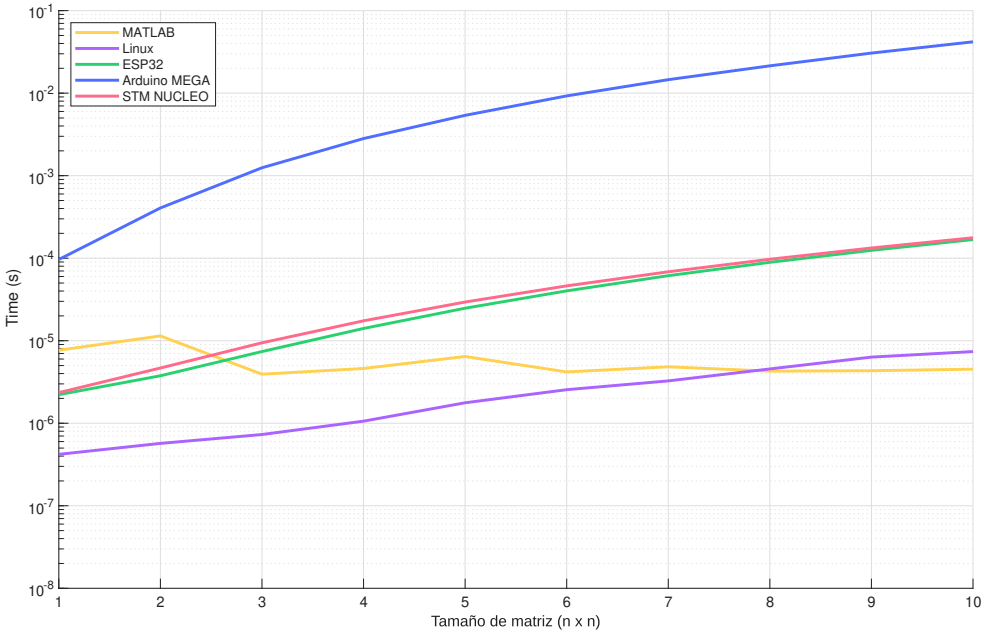
Estas funciones corresponden a multiplicaciones de dos matrices (`matf32_mul`) y de las matrices en un arreglo (`matf32_arr_mul`. Se validó que ambas son capaces de generar resultados con precisión de cinco decimales según lo esperado, aunque también son propensas a acumular error, lo que se atribuyó a la mayor cantidad de operaciones que realizan (comparado con sumas o restas). Además, dado que `matf32_arr_mul` implementa `matf32_mul` acorde a la cantidad de matrices (en este caso 3), es posible que el error (y el tiempo de operación) incremente al operar arreglos más grandes. Las Figuras 7 y 8 muestran que únicamente el Arduino MEGA obtuvo tiempos superiores a la métrica de milisegundos.

Figura 7. Evaluación del tiempo de operación de matf32_mul



Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

Figura 8. Evaluación del tiempo de operación de matf32_arr_mul



Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

Uso de memoria de las operaciones matriciales básicas

En los cuadros 3 y 4 se presenta un resumen del porcentaje de memoria necesaria durante las evaluaciones de las funciones de suma, resta, multiplicaciones, transpuesta, inversas y pseudoinversas. Esto en los tres dispositivos embebidos que se emplearon y, como se describió en la metodología, se incluyeron en la estimación de memoria los datos de todas las matrices empleadas en las operacionse, mientras que se excluyeron los datos de resultados numéricos de MATLAB ya que estos últimos no son relevantes en este caso.

En cuanto a los resultados, el Arduino MEGA fue el dispositivo que empleó más memoria RAM, obteniendo un rango aproximado entre 24 % y 37 %. Adicionalmente, en este mismo dispositivo, la inversa y la pseudoinversa calculada con la ecuación $\mathbf{A}^{+} = (\mathbf{A}^{\top}$ tuvieron el mayor costo en RAM. Por otro lado, la RAM del ESP32 alcanzó un máximo de 7.4 %, correspondiente a la inversa, ambos métodos de la pseudoinversa y la multiplicación de arreglos de matrices, e igualmente se mantuvo alrededor del 7 % en las demás operacionse, lo que implica que cambiar la operación a realizar no influyó significativamente en el ESP32. En cambio, el menor consumo de RAM fue en la NUCLEO, con un máximo de 3.1 % para la inversa y ambos métodos de la pseudoinversa.

En cuanto al uso de memoria FLASH, el ESP32 fue el dispositivo que requiriró un mayor porcentaje, con un rango de 20.4 %-20.8 %. Por el contrario, el Arduino MEGA utilizó la menor cantidad de FLASH en la mayoría de operaciones (<3 %), exceptuando la pseudoinversa con SVD que requirió 3.6 %. Por su parte, la STM NUCLEO utilizó entre 2.9 % y 3.7 % de la memoria FLASH.

Estos resultados se consideraron dentro de lo esperado, dado que el ESP32 y la STM NUCLEO tienen una mayor capacidad de memoria RAM comparados con el Arduino MEGA 1. A pesar de eso, más del 60 % de la memoria RAM del Arduino MEGA permaneció sin uso con todas estas operaciones, lo que indica que pueden añadirse una mayor cantidad de datos como será necesario en `linsolve` y incluso `quadprog`).

Cuadro 3. Porcentaje de uso de memoria RAM en los dispositivos embebidos para evaluar las operaciones matriciales básicas de `matf32`

Dispositivo	**Arduino MEGA**	**ESP32**	**STM NUCLEO**
Función	**RAM**	**RAM**	**RAM**
matf32_add	26.4 %	7.2 %	2.4 %
matf32_sub	26.4 %	7.2 %	2.4 %
matf32_scale	26.4 %	7.2 %	2.4 %
matf32_trans	31.2 %	7.3 %	2.7 %
matf32_mul	26.4 %	7.2 %	2.4 %
matf32_inv	37 %	7.4 %	3.1 %
matf32_dot	8.3 %	6.8 %	1.6 %
matf32_vecposmul	24.9 %	7.3 %	2.6 %
matf32_vecpremul	24.9 %	7.3 %	2.6 %
matf32_vecmul_col_row	13.3 %	6.8 %	1.6 %
matf32_arr_add	31.4 %	7.3 %	2.7 %
matf32_arr_sub	31.3 %	7.3 %	2.7 %
matf32_arr_mul	36.3 %	7.4 %	3.0 %
matf32_exp	26.4 %	7.2 %	2.4 %
matf32_pinv	37.0 %	7.4 %	3.1 %
matf32_pinv (con SVD)	31.2 %	7.4 %	3.1 %

Nota. Elaboración propia.

Cuadro 4. Porcentaje de uso de memoria FLASH en los dispositivos embebidos para evaluar las operacions matriciales básicas de `matf32`

Dispositivo	**Arduino MEGA**	**ESP32**	**STM NUCLEO**
Función	**FLASH**	**FLASH**	**FLASH**
matf32_add	1.6 %	20.5 %	3.1 %
matf32_sub	1.6 %	20.5 %	3.1 %
matf32_scale	1.6 %	20.5 %	3.1 %
matf32_trans	1.5 %	20.5 %	3.1 %
matf32_mul	1.8 %	20.5 %	3.1 %
matf32_inv	2.4 %	20.6 %	3.2 %
matf32_dot	1.0 %	20.4 %	2.9 %
matf32_vecposmul	1.9 %	20.5 %	3.1 %
matf32_vecpremul	1.9 %	20.5 %	3.1 %
matf32_vecmul_col_row	1.2 %	20.4 %	2.9 %
matf32_arr_add	1.7 %	20.5 %	3.1 %
matf32_arr_sub	1.8 %	20.5 %	3.1 %
matf32_arr_mul	2.0 %	20.5 %	3.1 %
matf32_exp	2.0 %	20.5 %	3.1 %
matf32_pinv	2.8 %	20.8 %	3.7 %
matf32_pinv (con SVD)	3.6 %	20.8 %	3.7 %

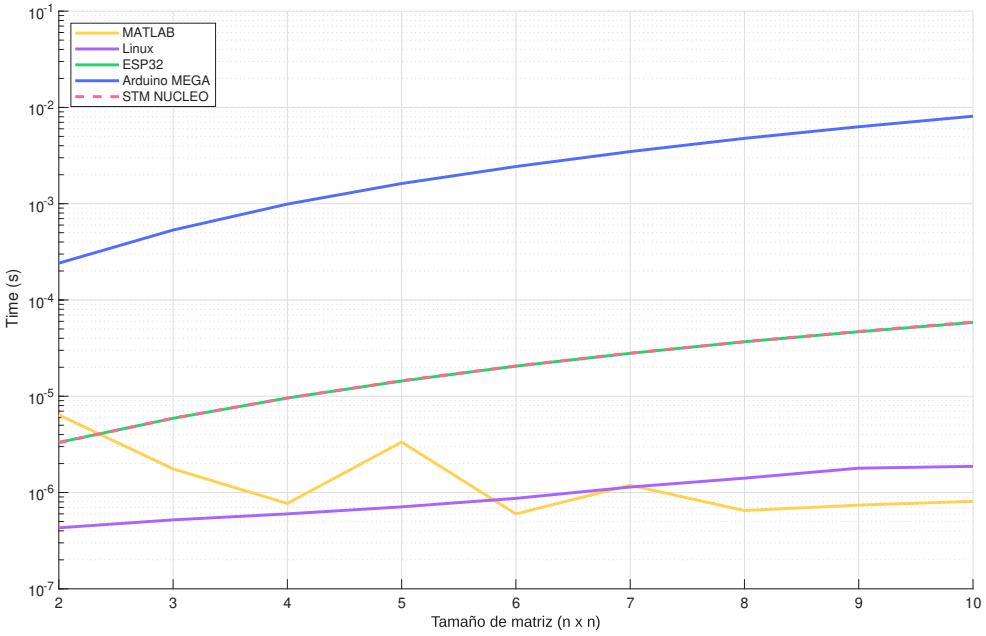
Nota. Elaboración propia.

7.2.4. Factorizaciones de matrices

`matf32_cholesky`

En esta función modificaron algunas operaciones para asignar los valores de la matriz generada en la factorización, ya que se identificó que algunos elementos de la matriz no se estaban generando correctamente. En cuanto a su tiempo de operación, fue casi igual para el ESP32 y la STM NUCLEO. Adicionalmente, en todos los dispositivos el tiempo permaneció en el orden de milisegundos o menor, por lo que es un resultado adecuado (Figura 9).

Figura 9. Evaluación del tiempo de operación de matf32_cholesky



Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

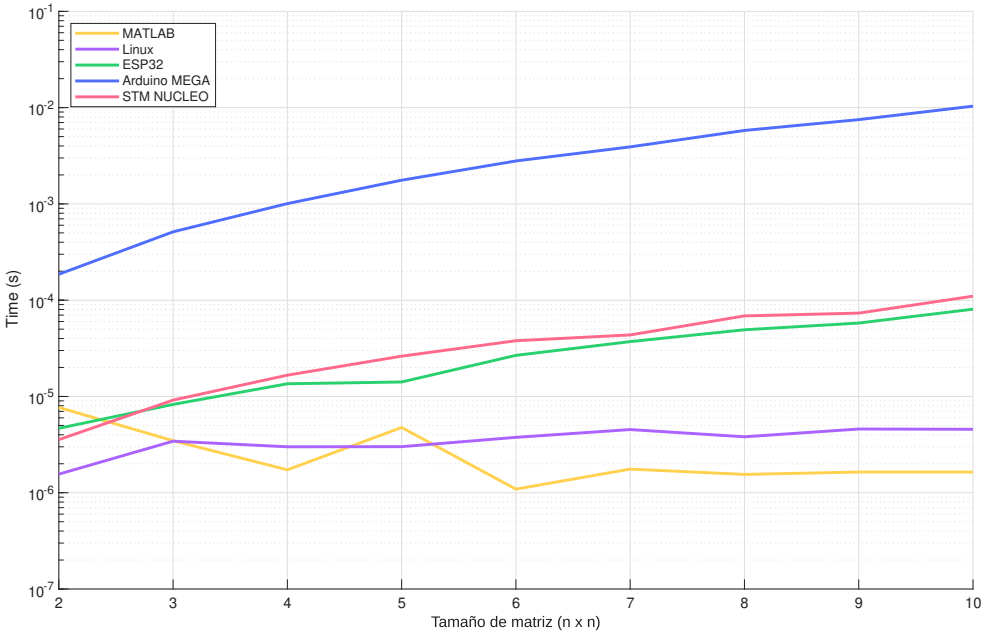
`matf32_lu`

Se modificó el algoritmo para calcular la factorización LU para incluir pivoteo o permutaciones parciales, porque durante su validación numérica, se identificó que en algunos casos el resultado tenía valores extremadamente elevados, lo que se atribuyó a que posiblemente se realizaban divisiones entre números cercanos a cero. Estos cambios permitieron mejorar la estabilidad y exactitud de los resultados, ya que el nuevo algoritmo coloca los valores más grandes en los pivotes que luego se emplean para dividir algunos valores, evitando divisiones entre cantidades reducidas (algoritmo 21.1 en [10]). También existe una versión de este algoritmo con pivoteo completo o total, en lugar de parcial, pero se escogió no emplearlo porque requiere una mayor cantidad de tiempo y la mejora en la estabilidad del algoritmo es reducida [10].

En cuanto a su tiempo de operación, el único caso en que este fue mayor al orden de

milisegundos fue en matrices 10×10 en el Arduino MEGA, aunque por una cantidad reducida, por lo que se consideró que el algoritmo es suficientemente eficiente en este dispositivo. El resto de evaluaciones en todos los dispositivos permanecieron en milisegundos o menos.

Figura 10. Evaluación del tiempo de operación de matf32_lu



Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

`matf32_jacobi_svd`

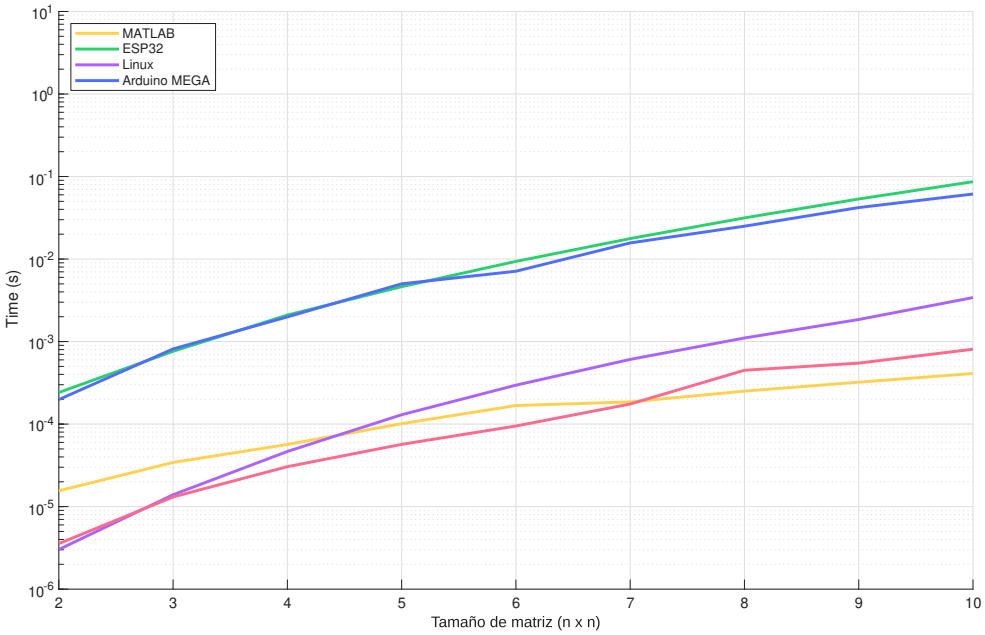
Adicionalmente, se añadieron las siguientes funciones para calcular la SVD en `matf32` de manera que pueda ser implementarada en `linsolve`:

- `matf32_one_sided_jacobi`: implementa un algoritmo de rotación de Jacobi de un solo lado a una matriz ingresada. Cabe destacar que modifica la matriz original, por lo que si esta se utilizará después (o incluso para verificar el resultado) es recomendable ingresar una copia de la matriz original (algoritmo 5.12 en [13]).
- `matf32_jacobi_svd`: implementa `matf32_one_sided_jacobi` con una cantidad específica de iteraciones, generando las matrices ortogonal $\mathbf{V}$ de la SVD y $\mathbf{AV} = \mathbf{U\Sigma}$ que se utilizan para generar las matrices $\mathbf{U}$ y $\mathbf{\Sigma}$ de la SVD. (algoritmo 5.13 en [13]).

La función `matf32_jacobi_svd` corresponde a la factorización con mayor tiempo de operación, superando el orden de milisegundos en el ESP32 y en el Arduino MEGA para matrices 7×7 y mayores (Figura 11), por lo que no se consideró práctico emplearla en estos casos. Aunque, cabe destacar que se empleó un total de 100 iteraciones para el algoritmo de `matf32_one_sided_jacobi`, como ejemplo de un caso en el que se requiera una cantidad elevada de iteraciones para converger a una precisión suficiente. Esto mismo pudo haber

influido en el tiempo de operación elevado, ya que en el algoritmo implementado se ejecutan todas las iteraciones indicadas, por lo que el tiempo se podría reducir en cierta medida si se disminuye la cantidad de iteraciones, lo cual puede funcionar en problemas con una convergencia rápida.

Figura 11. Evaluación del tiempo de operación de matf32_jacobi_svd

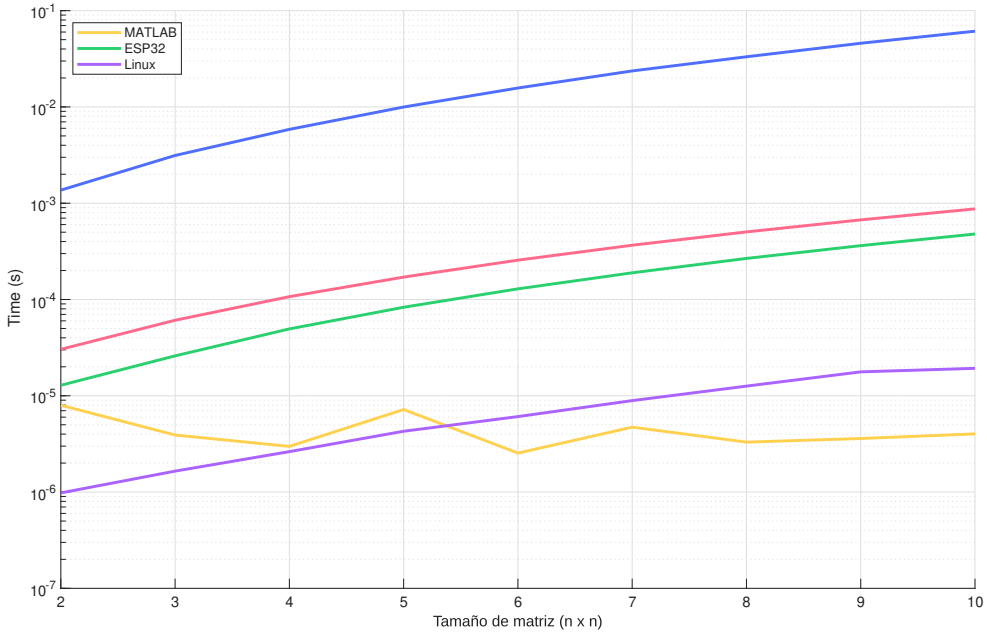


Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

`matf32_qr`

Esta función no requirió cambios a su versión anterior. Se validó su funcionamiento tanto para matrices cuadradas como rectangulares. No obstante, la evaluación del tiempo de operación que se presenta en la Figura 12 se realizó con matrices cuadradas únicamente, ya que en general estas son más grandes que las matrices rectangulares y, por consecuencia, pueden requerir más tiempo, generando un estimado amplio del tiempo de operación. En el Arduino MEGA el tiempo superó el orden de milisegundos, lo que se consideró lento para este, específicamente para matrices a partir de 6×6. No obstante, en el resto de dispositivos el tiempo se mantuvo por debajo del orden de milisegundos, motrando un rendimiento adecuado.

Figura 12. Evaluación del tiempo de opearción de matf32_qr



Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

Uso de memoria de las factorizaciones de matrices

En los cuadros 5 y 6 se resumen el consumo de memoria de los dispositivos embebidos al evaluar las factorizaciones de matrices. Este presentó un patrón similar al de las operaciones matriciales básicas. El mayor uso de RAM fue en el Arduino MEGA (26.4-36.8 %), mientras que el ESP32 empleó el mayor porcentaje de memoria FLASH (20.6-20.7 %). No obstante, en el Arduino MEGA hubo un ligero incrementó en el uso de memoria FLASH comparado con las operaciones básicas, dado que representó entre 2 y 3.3 % de la memoria, cuando en las operaciones básicas este se mantuvo menor al 2 %. No obstante, no se consideró un incremento significativo.

Cuadro 5. Porcentaje de uso de memoria RAM en los dispositivos embebidos para evaluar las factorizaciones de matrices de `matf32`

Dispositivo	**Arduino MEGA**	**ESP32**	**STM NUCLEO**
Función	**RAM**	**RAM**	**RAM**
matf32_lu	36.3 %	7.3 %	2.7 %
matf32_cholesky	26.4 %	7.2 %	2.4 %
matf32_qr	36.8 %	7.4 %	2.7 %
matf32_jacobi_svd	36.5 %	7.5 %	2.1 %

Nota. Elaboración propia.

Cuadro 6. Porcentaje de uso de memoria FLASH en los dispositivos embebidos para evaluar las factorizaciones de matrices de `matf32`

Dispositivo	**Arduino MEGA**	**ESP32**	**STM NUCLEO**
Función	**FLASH**	**FLASH**	**FLASH**
matf32_lu	2.7 %	20.6 %	3.2 %
matf32_cholesky	2.3 %	20.6 %	3.2 %
matf32_qr	3.0 %	20.6 %	3.3 %
matf32_jacobi_svd	3.3 %	20.7 %	3.5 %

Nota. Elaboración propia.

7.3. Librería `linsolve`: solucionador lineal

7.3.1. Cambios realizados a los archivos de la librería

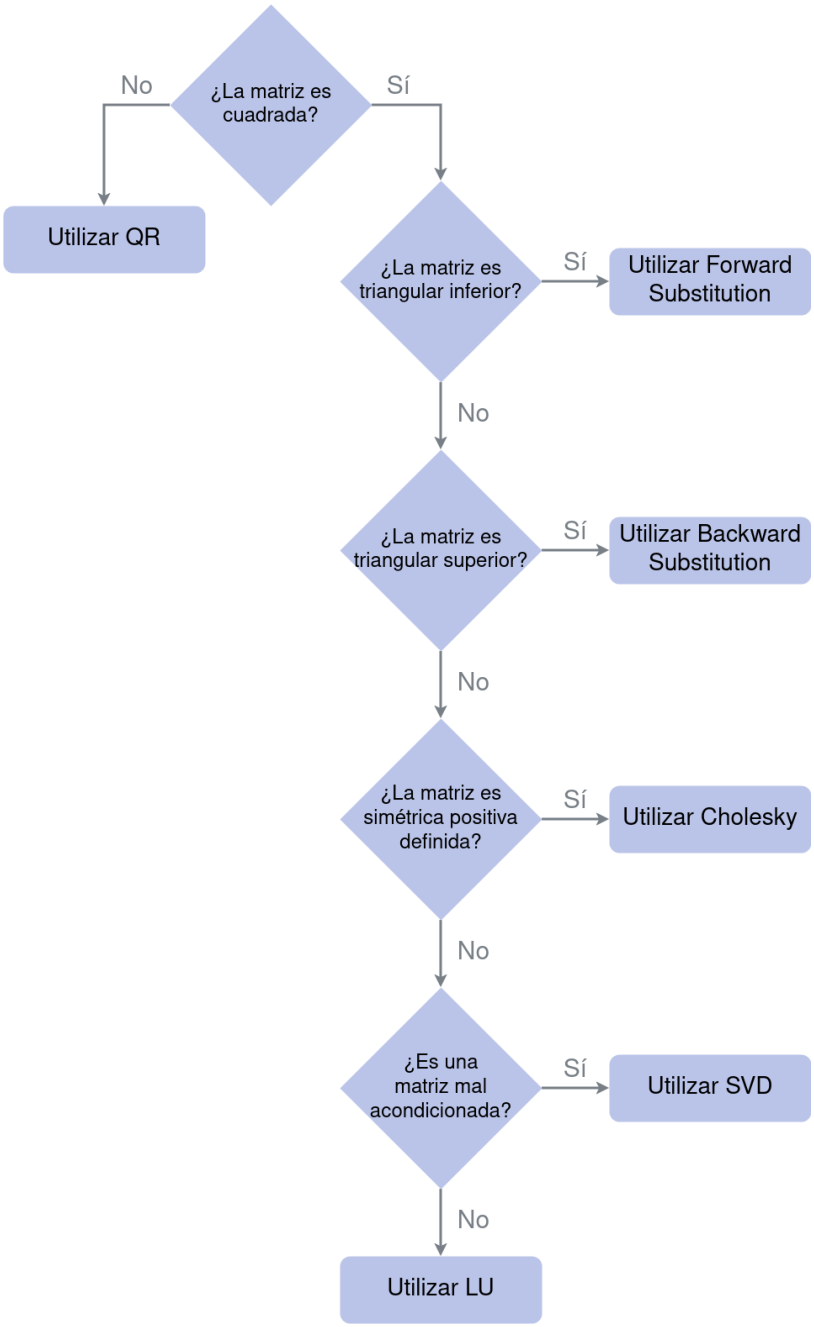
En esta librería se implementan las factorizaciones de matrices disponibles en `matf32` para resolver sistemas lineales $\mathbf{Ax} = \mathbf{b}$. Con respecto de la versión de `linsolve`, se modificaron los nombres originales de las funciones para iniciar con `linsolve_`, indicando así que pertenecen a esta librería (ya que anteriormente iniciaban con `matf32_`).

7.3.2. Estructura del solucionador lineal

El solucionador está planteado acorde al solucionador lineal de MATLAB. Esto quiere decir que cada método se emplea para resolver un tipo de matriz en específico, acorde a cómo MATLAB realiza esto. Por ejemplo: QR para sistemas con matrices $\mathbf{A}$ rectangulares, Cholesky para matrices simétricas positivas definidas y así sucesivamente. Esto se puede verificar en el diagrama de flujo del solucionador, en la Figura 13.

Por otro lado, la SVD no tiene equivalente en el solucionador de MATLAB, en cambio, se añadió específicamente para casos en los que se considere que las matrices tienen un mal acondicionamiento y se requiera un método más robusto que LU. Esta condición es configurable como una constante en el archivo `constants.h` que contiene los valores para cantidades como dimensiones límite de matrices, cantidad de iteraciones en algunas operaciones, entre otras configuraciones.

Figura 13. Diagrama de flujo para la ruta del algoritmo del solucionador lineal de la librería `linsolve`



Nota. En este diagrama se muestran las condiciones bajo las cuales se emplea cada método del solucionador lineal, acorden a cierto tipo específico de matriz. Elaboración propia.

El solucionador lineal se puede ejecutar con las siguientes funciones, una de las cuales selecciona el método automáticamente y en la otra se debe específicar esto de forma manual, por lo que cuál utilizar depende de la necesidad del usuario:

- `linsolve`: esta función ejecuta el solucionador lineal, escogiendo automáticamente el método dependiendo del tipo de matriz ingresada en el sistema. Para esto, emplea dos funciones: `linsolve_get_method` para determinar el método a utilizar y `linsolve_method`, que se explica en el siguiente punto. Si no se está seguro de qué método de solución es mejor, o no importa cuál se utilice, se recomienda ejecutar esta función.
- `linsolve_method`: genera la factorización correspondiente (por ejemplo, con `matf32_qr`, `matf32_lu`, etc.) y luego utiliza las matrices generadas para solucionar el sistema lineal (por ejemplo, `linsolve_qr`, `linsolve_lu` y así para cada método). En los métodos de *Forward* y *Backward substitution* no requieren factorizaciones de matrices ya que son métodos específicos para solucionar sistemas lineales. Aparte de eso, el método a utilizar se ingresa en los argumentos de la función, por esto, si se necesita un método específico, se recomienda emplear esta función.

Adicionalmente, ya que en la versión anterior de `linsolve` únicamente se implementaron los métodos de solución con LU, sustitución hacia adelante (*Forward Substitution*) y sustitución hacia atrás (*Backward Substitution*). En este trabajo se completó la implementación para QR y Cholesky, y además se añadió la solución por medio de SVD. Luego, se validó el funcionamiento de todos estos y se evaluó el tiempo de operación, exactitud numérica y uso de memoria con el procedimiento descrito en secciones anteriores.

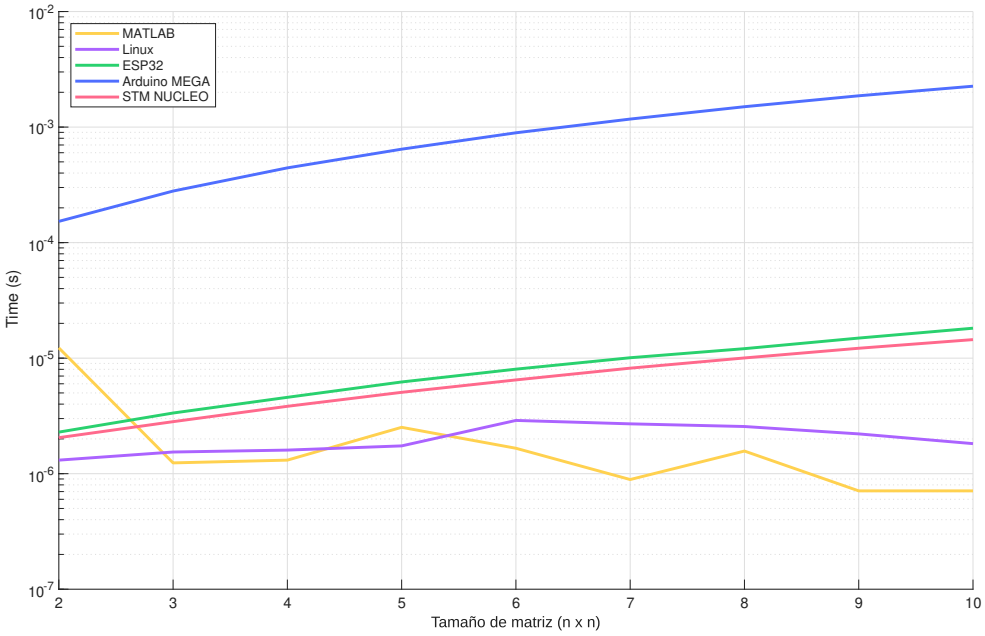
A continuación, se presenta un resumen de los resultados de tiempo y exactitud numérica de los distintos métodos de `linsolve`, y después de eso se presenta la evaluación del uso de memoria en cada dispositivo embebido.

7.3.3. `linsolve` con *Forward* y *Backward Substitution*

El tiempo de operación de estos métodos permaneció en el orden de milisegundos o menor en todos los dispositivos, para todas las matrices evaluadas, por lo que se consideró aceptable en todos los casos (Figuras 14-15). Aunque, cabe destacar que la solución por medio de LU y Cholesky utilizan *Forward* y *Backward Substitution*, y de forma similar, al emplear QR para solucionar sistemas lineales se debe utilizar *Backward Substitution*. Esto quiere decir que el tiempo de operación de *Forward* y *Backward Substitution* va a formar parte del tiempo para ejecutar los demás métodos, generando por consecuencia tiempos de operación más elevados en esos, lo que debe tomarse en cuenta a la hora de aplicarlos.

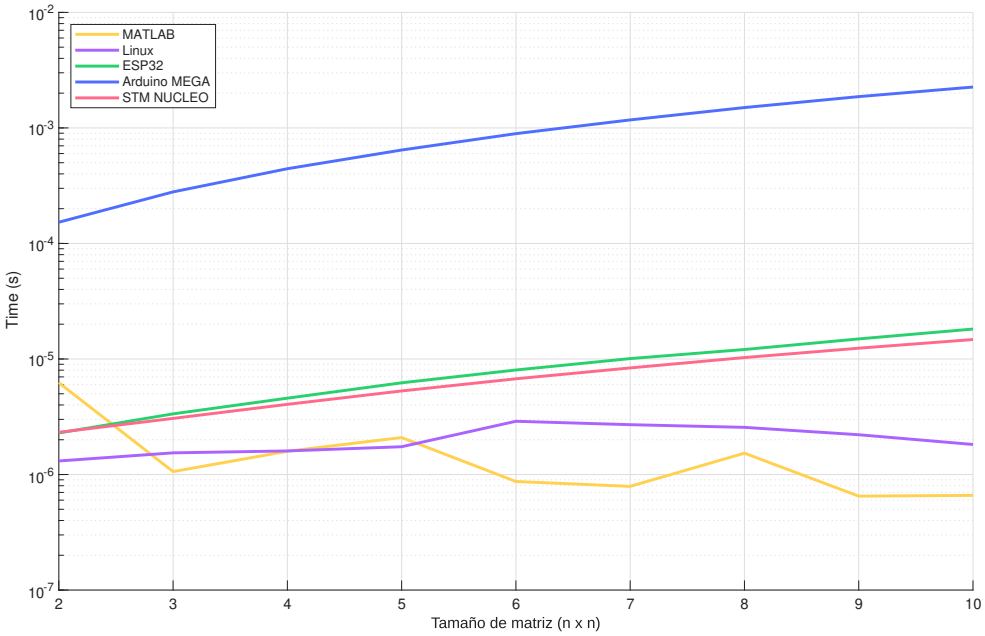
En cuanto a la exactitud numérica, tanto con *Forward* y *Backward Substitution* fue posible obtener resultados con errores menores a la tolerancia de 1×10^{-5} decimales, por lo que se consideró que funcionan adecuadamente. Sin embargo, en el caso de matrices con un mayor acondicionamiento, se observó un incremento en el error numérico, lo que podría propagarse hacia la solución de los demás métodos. Ambas sustituciones tienden a ser estables, sin embargo, en caso de necesitar una solución aún más robusta, existen otros algoritmos para estas sustituciones que podrían evaluarse [12].

Figura 14. Evaluación del tiempo de operación de `linsolve` usando *Forward Substitution*



Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

Figura 15. Evaluación del tiempo de operación de `linsolve` usando *Backward Substitution*



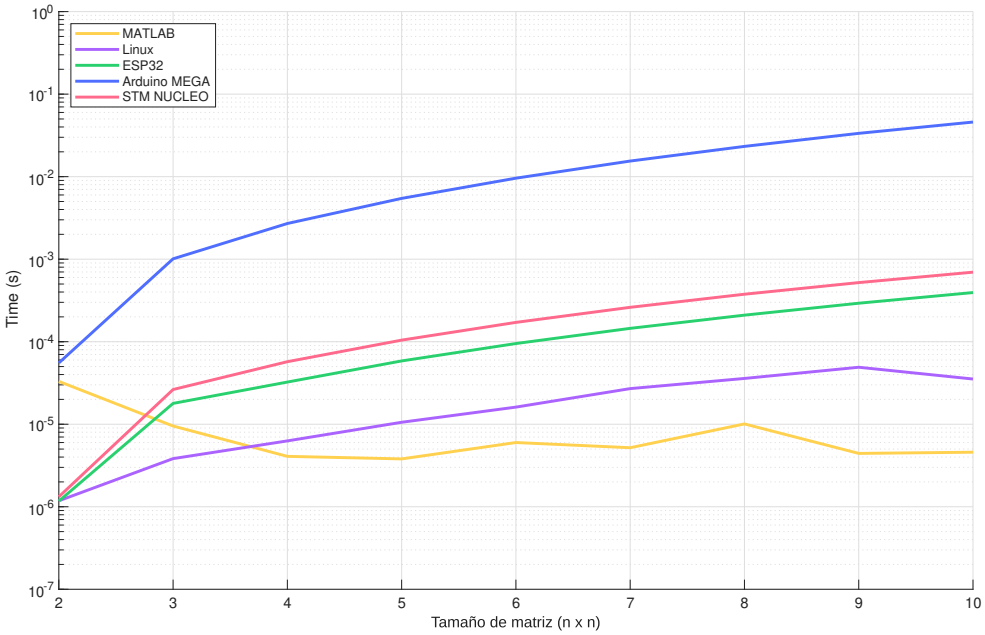
Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

7.3.4. `linsolve` con QR, LU y Cholesky

Se agregó la función `linsolve_qr`: Resuelve un sistema lineal a partir de la factorización QR. Se incluyó un argumento para seleccionar si la matriz del sistema es cuadrada o rectangular, ya que ambos casos difieren en un paso (con matrices rectangulares se debe eliminar una fila extra de ceros en la matriz **R** antes de aplicar *Backward Substitution*). En el solucionador lineal automático (`linsolve`, se implementa únicamente para resolver problemas con matrices rectangulares, acorde al solucionador de MATLAB, mientras que en `quadprog` se implementó por medio de `linsolve_method` ya que es necesario resolver específicamente para matrices cuadradas, que es la forma de las matrices en los sistemas KKT (18).

En cuanto a la evaluación de su tiempo de operación, se emplearon únicamente matrices rectangulares verticales ($n \times n-1$, con n $= 1:10$), para poder comparar con el solucionador de MATLAB. Como era esperado, se alcanzaron tiempos de operación mayores a los de *Forward* y *Backward Substitution*, atribuído parcialmente al uso de *Backward Substitution* dentro del procedimiento, como se explicó en la sección anterior. Sin embargo, en la mayoría de casos el tiempo permaneció por debajo del orden de milisegundos, lo que se consideró relativamente rápido, exceptuando el Arduino MEGA que obtuvo un tiempo igual o mayor a milisegundos en la mayoría de casos (Figura 16).

Figura 16. Evaluación del tiempo de operación de `linsolve` usando QR

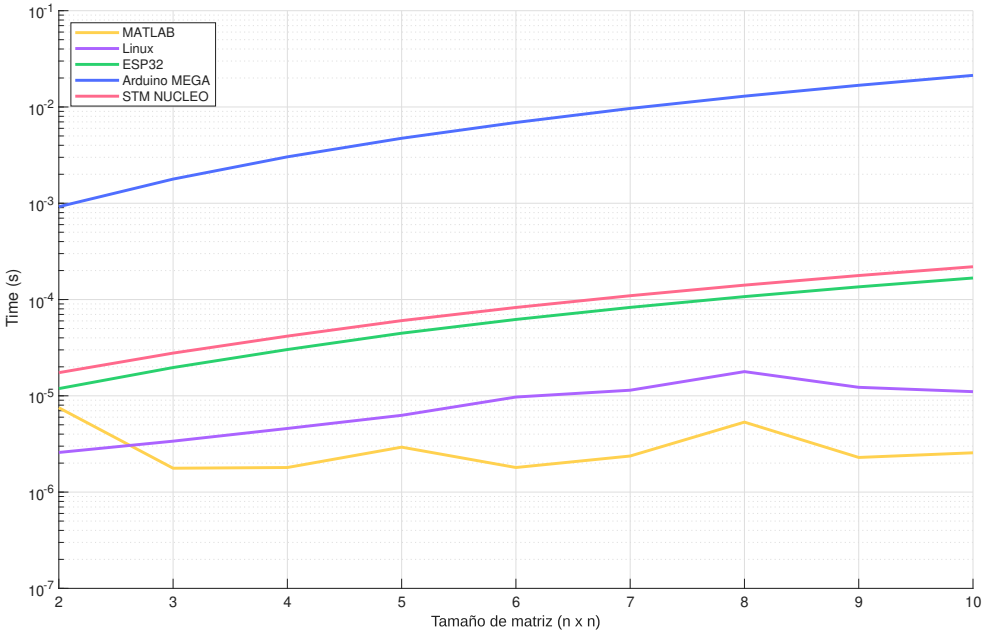


Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

Se añadió la función `linsolve_cholesky` par resolver un sistema lineal a partir de la factorización Cholesky. Se asume que el factor Cholesky (la matriz **L** generada en la factorización), se recibe como una matriz triangular superior, dado que `matf32_cholesky` devuelve el factor en esta forma. Comparado con QR, la solución con Cholesky tuvo un tiempo ligeramente menor, incluso en el Arduino MEGA a pesar de que algunos casos en este último

superaron el orden de milisegundos (Figura 17).

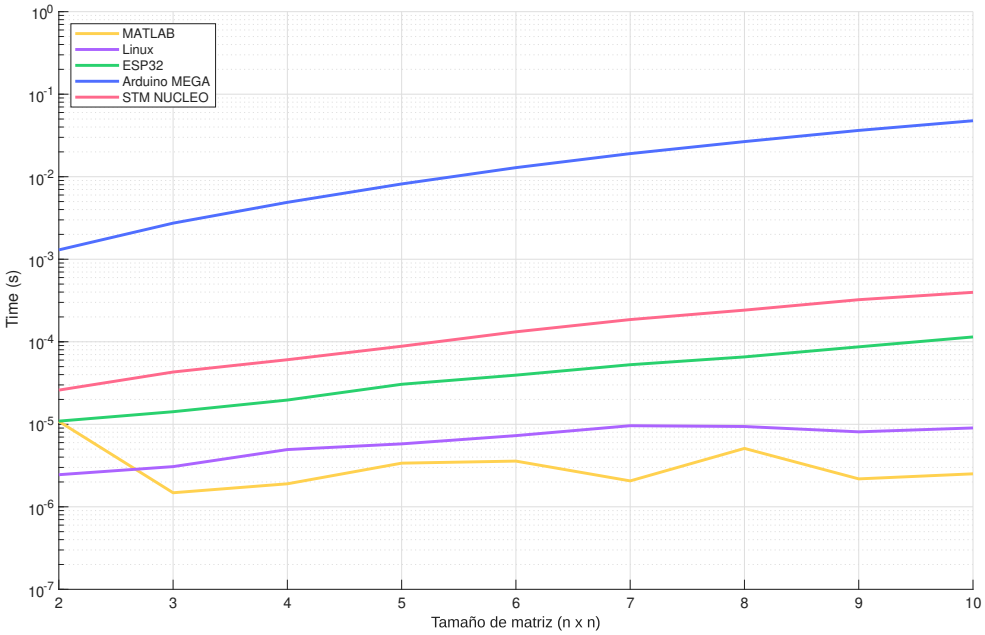
Figura 17. Evaluación del tiempo de operación de `linsolve` usando Cholesky



Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

Aparte de eso, la función `linsolve_lu` se modificó de su versión anterior, acorde a los cambios realizados en `matf32_lu`, ya que es necesario aplicar al vector **b** del sistema las mismas permutaciones generadas durante la factorización. El algoritmo resultante requirió un tiempo de operación similar a Cholesky en la mayoría de dispositivos, excepto en el Arduino MEGA, cuyó tiempo de operación fue más semejante con QR (Figuras 16-18). Esto quiere decir que entre emplear QR y LU no hay mucha diferencia de tiempo con el Arduino MEGA, sin embargo, se recomienda limitar su utilización a matrices menores a 6×6 en este dispositivo para que el tiempo se mantenga dentro del rango de tiempo aceptable.

Figura 18. Evaluación del tiempo de operación de `linsolve` usando LU



Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

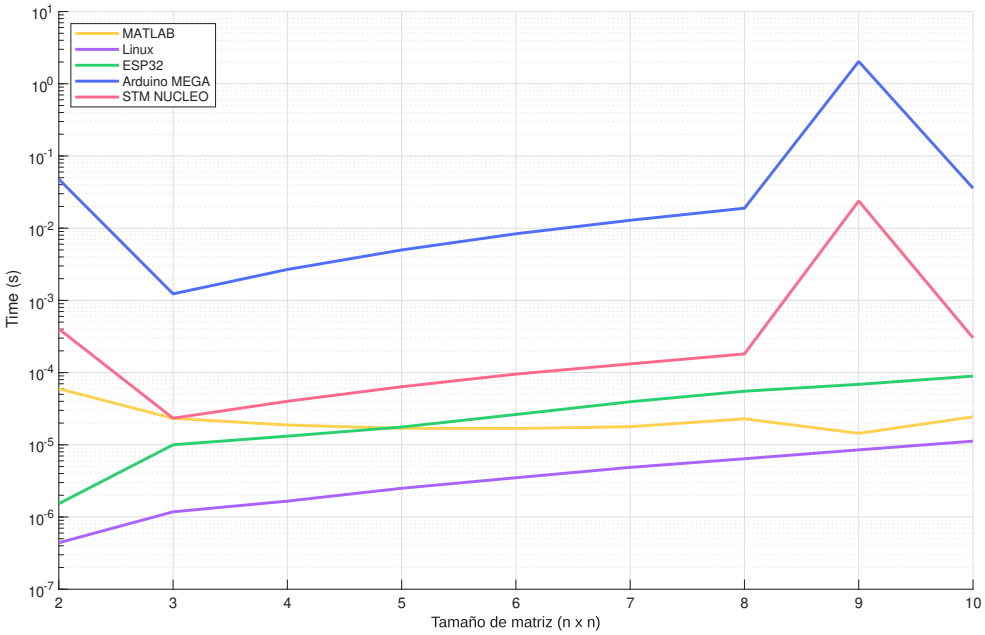
La exactitud numérica de QR, Cholesky y LU para solución de sistemas lineales fue relativamente similar. En los tres métodos se validó que es posible obtener resultados con un error numérico menor a la tolerancia de 1×10^{-5} decimales, por lo que se consideró que los algoritmos implementados funcionan adecuadamente y son robustos. Sin embargo, hubo casos en los que el resultado tuvo un error ligeramente superior a la tolerancia lo que, dependiendo de la matriz empleada. Esto fue especialmente notorio al emplear LU, cuya factorización correspondiente es más sensible que los otros métodos a la magnitud de los elementos, incluso al haber implementado permutaciones dado que depende mucho de los valores como tal de la matriz. Por esto mismo, puede ser parcialmente por acumulación de error en las operaciones o por mal acondicionamiento de las matrices en el sistema.

7.3.5. `linsolve` con SVD

Se añadió la función `linsolve_svd` para resolver un sistema lineal por medio de la SVD, empleando la siguiente ecuación: $\mathbf{x} = \mathbf{V}\mathbf{\Sigma}^{-1}\mathbf{U}^{\top}\mathbf{b}$ [12]. En cuanto al tiempo de operación, en todos los casos evaluados con el ESP32 se mantuvo por debajo del orden de milisegundos, lo que se consideró adecuado. Por otro lado, con el Arduino MEGA y la STM NUCLEO, se puede observar en la Gráfica 19 que el tiempo no siguió un patrón definido, sino que presenta ciertas irregularidades. Por ejemplo, en la STM NUCLEO, el caso con matrices 9×9 fue el único que superó el orden de milisegundos, mientras que todos los demás casos permanecieron por debajo de esta métrica. Hubo un comportamiento similar con el Arduino MEGA, aunque este último fue más lento que la STM NUCLEO. Esto quiere decir que, en contraste con los demás métodos, el tiempo de `linsolve` empleando la SVD no necesariamente va a aumentar

de forma consistente y podría ser propenso a incrementos repentinos independiente de las dimensiones de la matriz.

Figura 19. Evaluación del tiempo de operación de `linsolve` usando SVD



Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

7.3.6. Uso de memoria de los métodos de `linsolve`

En los cuadros 7 y 8 se presentan los porcentajes de memoria empleada durante las evaluaciones de los métodos de `linsolve`. Al igual que con `matf32`, se excluyeron de esta medición los datos de los resultados numéricos de MATLAB, para enfocar la estimación únicamente en la memoria que pueden ocupar los métodos y sus respectivos datos de entrada.

Se midió que los métodos de `linsolve` ocuparon entre el 77.3 y 80.1 % de la RAM del Arduino MEGA, lo cual es más del doble de la memoria requerida durante las evaluaciones de `matf32` en este dispositivo. Esto se atribuyó a varios aspectos. Primero, la cantidad de datos de entrada fue mayor que en las evaluaciones de `matf32`, en donde esto correspondía a las matrices de entrada de 2×2 hasta 10×10 para las factorizaciones, además de una o dos matrices adicionales (dependiendo de la operación) para guardar el resultado de las operaciones. Ahora, en el caso de `linsolve`, se emplearon los datos de las matrices y vectores de los sistemas, igualmente para sistemas desde 2×2 hasta 10×10, junto a un vector adicional para el resultado. Entonces, la cantidad de datos de entrada es mayor y es de esperar que se utilice más memoria.

Sin embargo, se identificó que la mayor parte de la memoria ocupada realmente corresponde a que dentro de cada método de `linsolve` se generan varias matrices para almacenar tanto las factorizaciones generadas y resultados parciales necesarios en distintos pasos del procedimiento. Combinando todos estos aspectos, es razonable el incremento en el uso de

memoria del Arduino MEGA, no obstante, que los métodos como tal requieran una gran cantidad de memoria también limita las aplicaciones que se pueden construir a partir de `linsolve` en este dispositivo específico.

Por otro lado, la memoria FLASH del Arduino MEGA empleó entre 7.5 y 7.6 % para todos los métodos de `linsolve`, lo cual está dentro de lo esperado ya que es mayor que en sus evaluaciones con `matf32`, pero al ser menor al 10 % se consideró que no incrementó significativamente.

En cuanto al ESP32, el incremento en el uso de memoria con respecto de las operaciones de `matf32` fue menor a 1 % tanto en la memoria RAM como en la FLASH, por lo que no se consideró relevante. Por su parte, la STM NUCLEO requirió alrededor del doble de memoria al emplear `linsolve` con respecto de `matf32`, mientras que en la FLASH empleó poco más de 1 % en comparación. A partir de esto, el uso de `linsolve` es más adecuado en el ESP32 y la STM NUCLEO, o bien, en caso de ser necesario utilizarlo, en aplicaciones con dimensiones y cantidad de datos reducidas en el Arduino MEGA. A partir de esto, se pueden identificar las ventajas que tienen el ESP32 y la STM NUCLEO ya que, al tener una mayor capacidad, el impacto en su memoria al añadir operaciones más complejas es menor, a comparación con el Arduino MEGA (1).

Cuadro 7. Porcentaje de uso de memoria RAM en los dispositivos embebidos para evaluar los métodos de `linsolve`

Dispositivo	**Arduino MEGA**	**ESP32**	**STM NUCLEO**
Función	**RAM**	**RAM**	**RAM**
linsolve_forward_substitution	80.0 %	7.6 %	5.8 %
linsolve_backward_substitution	80.0 %	7.6 %	5.8 %
linsolve_cholesky	80.0 %	7.6 %	5.8 %
linsolve_qr	77.3 %	7.6 %	5.6 %
linsolve_lu	80.0 %	7.6 %	5.8 %
linsolve_svd	80.1 %	7.6 %	5.8 %

Nota. Elaboración propia.

Cuadro 8. Porcentaje de uso de memoria FLASH en los dispositivos embebidos para evaluar los métodos de `linsolve`

Dispositivo	**Arduino MEGA**	**ESP32**	**STM NUCLEO**
Función	**FLASH**	**FLASH**	**FLASH**
linsolve_forward_substitution	7.6 %	20.9 %	4.4 %
linsolve_backward_substitution	7.6 %	20.9 %	4.4 %
linsolve_cholesky	7.6 %	20.9 %	4.4 %
linsolve_qr	7.5 %	20.8 %	4.4 %
linsolve_lu	7.6 %	20.9 %	4.4 %
linsolve_svd	7.6 %	20.9 %	4.4 %

Nota. Elaboración propia.

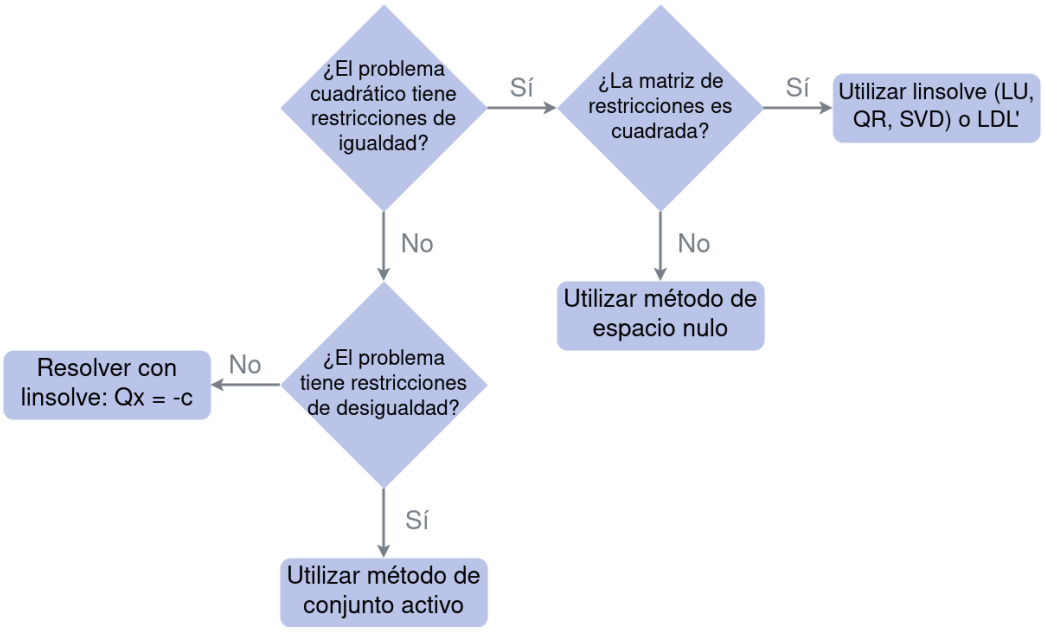
7.4. Librería `quadprog`: solucionador cuadrático

7.4.1. Cambios en la estructura del solucionador cuadrático

El solucionador cuadrático de `quadprog` consistía originalmente de un método directo que emplea `linsolve` para resolver problemas cuadráticos con restricciones de igualdad (`quadprog_qp`), así como un método de conjunto activo que implementaba `quadprog_qp` para resolver problemas con restricciones de desigualdad. Durante la evaluación de esta librería, se identificó que los métodos de `linsolve` que se pueden aplicar a los sistemas KKT, debido a las características de este, son QR, LU y SVD. Por eso mismo, se modificó `quadprog_qp` para poder escoger entre estos tres métodos específicamente por medio de un argumento en la función y se renombró a esta función como `quadprog_qp_linsolve`. Adicionalmente, se añadieron dos métodos recomendados específicamente para la solución de sistemas lineales KKT, los cuales emplean, asímismo, métodos de linsolve para resolver los sistemas lineales que construyen en el proceso de solución al sistema KKT [11].

Aparte de eso, se modificó y validó el método de conjunto activo para resolver problemas con restricciones de desigualdad. Finalmente, se añadió al solucionador cuadráticos un caso para resolver problemas sin restricciones, cuando no aplique utilizar alguno de los métodos para restricciones de igualdad ni el de conjunto activo. A partir de eso, el diagrama de flujo del solucionador cuadrático es el siguiente para escoger qué método utilizar:

Figura 20. Diagrama de flujo para la ruta del algoritmo del solucionador cuadrático



Nota. En este diagrama se muestran los casos para los cuales se emplea cada método del solucionador cuadrático. Elaboración propia.

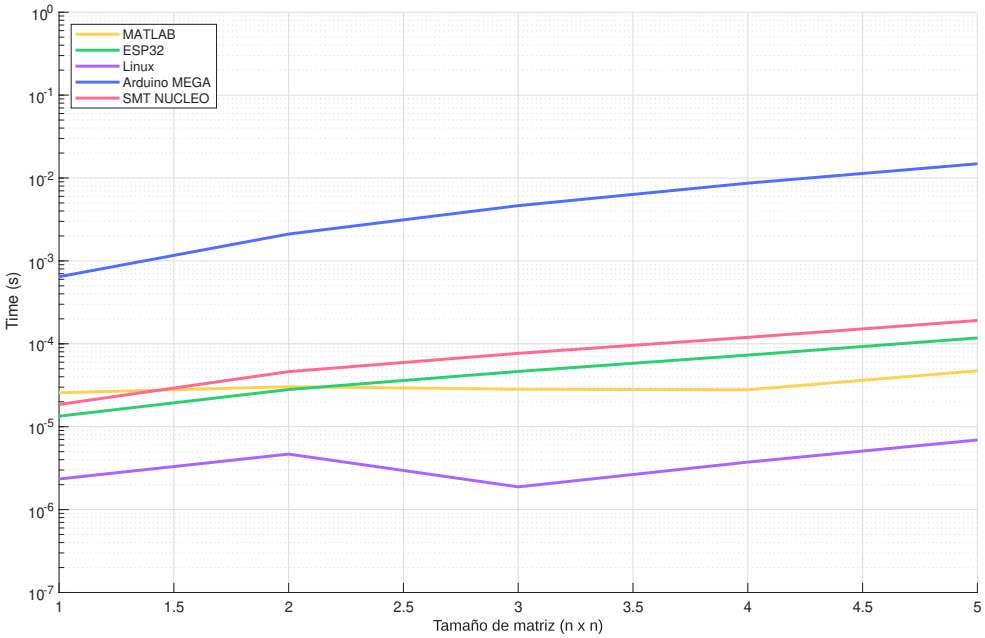
7.4.2. Métodos para solución de problemas con restricciones de igualdad

`quadprog_qp_linsolve`

Las dimensiones de las matrices a utilizar se definieron de forma distinta a las librerías anteriores. Primero, dentro de `quadprog_qp_linsolve`, se emplea linsolve a la matriz del sistema KKT. Dada la configuración actual de Robotat Linalg, esta matriz puede ser de hasta 10×10, como máximo. A su vez, esta se construye a partir de la matriz $\mathbf{Q}$ del programa cuadrático y sus restricciones lineales $\mathbf{A}$ (18). Con base en eso, se emplearon matrices $\mathbf{Q}$ y $\mathbf{A}$ cuadradas, con dimensiones $(n \times n)$, y sus respectivos vectores $\mathbf{c}$ y $\mathbf{b}$ $(n \times 1)$.

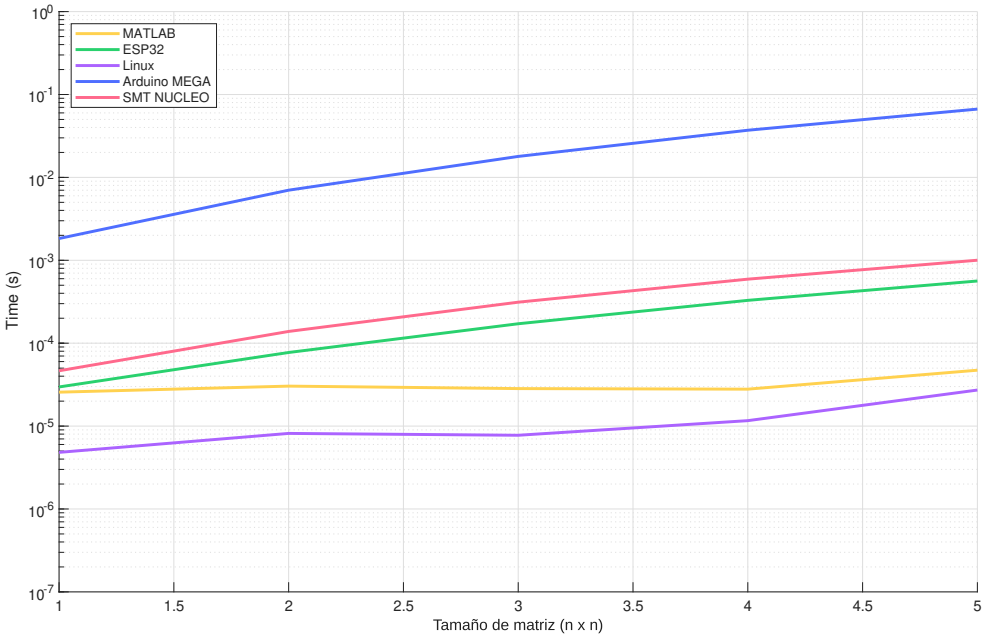
En las gráficas 21-23 se incluye la evaluación de tiempo de los tres métodos en esta función. En el eje x se indican las dimensiones de las matrices $\mathbf{Q}$ y $\mathbf{A}$. Se consideró que los métodos más adecuados fueron al emplear LU y QR, ya que mantuvieron tiempos en el orden de milisegundos o menor tanto para el ESP32 como para la STM NUCLEO. Esto, contrario al emplear SVD que para ambos dispositivos el tiempo superó esta métrica. En cambio, el método más rápido en el Arduino MEGA fue LU, dado que solo al emplear matrices $\mathbf{Q}$ y $\mathbf{A}$ de 5×5 se obtuvo un tiempo mayor al orden de milisegundos. Acerca de la exactitud numérica, en los tres métodos se logró obtener resultados con un error por debajo de la tolerancia de 1×10^{-5} decimales, aunque también se observó que son propensos a acumular error, lo que puede deberse a la propia sensibilidad de los métodos de linsolve como se discutió en secciones anteriores. Por esto mismo, es importante emplear el método que proporcione la solución más exacta al problema específico a resolver.

Figura 21. Evaluación del tiempo de operación de `quadprog_qp_linsolve` empleando la factorización LU



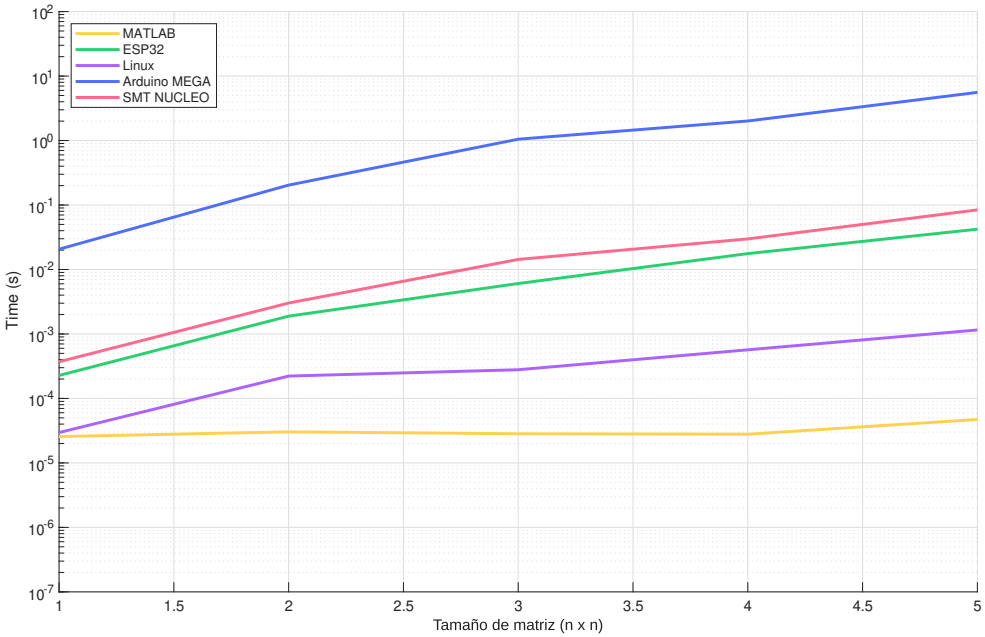
Nota. Tiempo de operación promedio contra el tamaño de la matriz, para matrices $\mathbf{Q}$ y $\mathbf{A}$ $(n \times n)$, para $n = 1:1:5$. Elaboración propia.

Figura 22. Evaluación del tiempo de operación de `quadprog_qp_linsolve` empleando la factorización QR



Nota. Tiempo de operación promedio contra el tamaño de la matriz, para matrices **Q** y **A** $(n \times n)$, para $n = 1:1:5$. Elaboración propia.

Figura 23. Evaluación del tiempo de operación de `quadprog_qp_linsolve` empleando la SVD



Nota. Tiempo de operación promedio contra el tamaño de la matriz, para matrices **Q** y **A** $(n \times n)$, para $n = 1:1:5$. Elaboración propia.

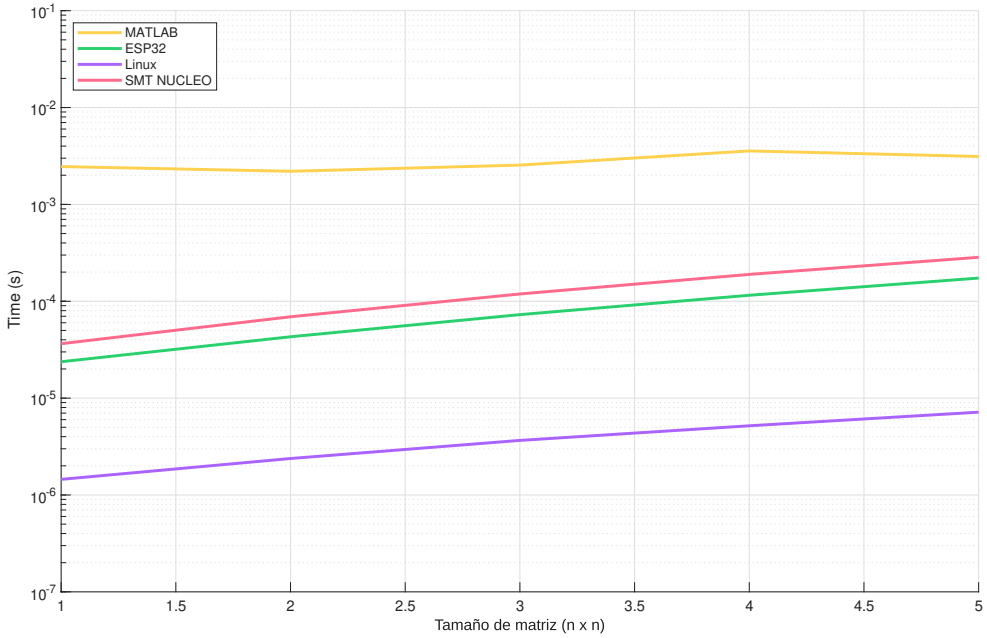
`quadprog_qp_nullspace`

Se añadió la función `quadprog_qp_nullspace`, que implementa un método del espacio nulo para solucionar problemas cuadráticos con restricciones por igualdad [11], empleando `linsolve` para solucionar los sistemas que construye en este proceso.

La principal ventaja de `quadprog_qp_nullspace` es que no construye el sistema KKT, en cambio este método genera soluciones parciales directamente con la matriz de restricciones $\mathbf{A}$ y la matriz del término cuadrático $\mathbf{Q}$ del problema convexo (19-21), por lo que los sistemas a resolver son más pequeños en comparación con aplicar `linsolve` directamente al sistema KKT como hace `quadprog_qp_linsolve`. Luego estas soluciones parciales se suman $\mathbf{p} = \mathbf{Y}\mathbf{p}_y + \mathbf{Z}\mathbf{p}_z$. No obstante, en caso de que $\mathbf{A}$ sea cuadrada, la matriz $\mathbf{Z}$ será inexistente por la manera en la que esta última se construye (20). En este caso, la solución será únicamente $\mathbf{p} = \mathbf{Y}\mathbf{p}_y$, omitiendo la solución parcial que se genera a partir de los términos $\mathbf{Q}$ y $\mathbf{c}$ de la función a optimizar, generando un resultado que puede diferir de la solución de MATLAB. Por esto mismo, se consideró adecuado utilizar `quadprog_qp_nullspace` únicamente en problemas con matrices de restricción $\mathbf{A}$ rectangulares.

Para evaluar este método, se emplearon matrices $\mathbf{Q}$ cuadradas $(n \times n)$ y matrices de restricción $\mathbf{A}$ rectangulares verticales $(n \times n + 1)$. El tiempo de operación de `quadprog_qp_nullspace` fue un resultado aceptable en todos los dispositivos evaluados (milisegundos o menor). En cuanto a su exactitud numérica, se obtuvieron resultados de cinco o más decimales, en todos los casos evaluados. Por último, no se observó acumulación de error, por lo que se consideró un método robusto y eficiente.

Figura 24. Evaluación del tiempo de operación de `quadprog_qp_nullspace`



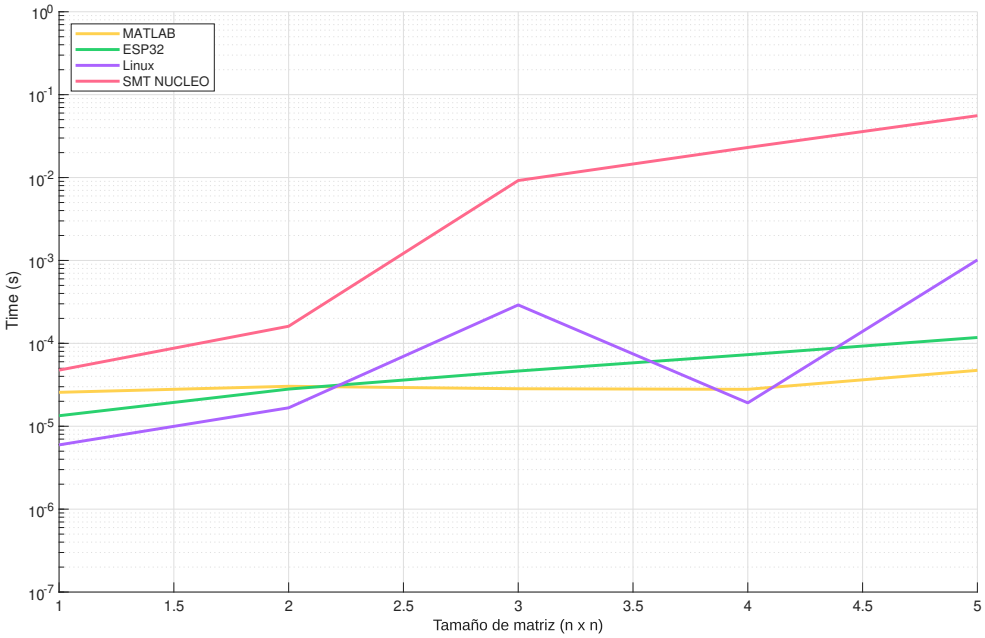
Nota. Tiempo de operación promedio contra el tamaño de la matriz, para matrices $\mathbf{Q}$ $(n \times n)$ y $\mathbf{A}$ $(n \times n + 1)$, para $n = 1 : 1 : 5$, con incrementos de 1. Elaboración propia.

`quadprog_qp_ldlt`

Se agregó la función `quadprog_qp_ldlt` para solucionar un problema cuadrático con restricciones de igualdad por medio de la factizarión $LDL^\top$ con permutaciones de Bunch y Parlett [12] [11]. Esta factorizaición es recomendada para solucionar sistemas con matrices simétricas indefinidas, como las que se generan para un sistema KKT, ya que permite una mayor estabilidad numérica en la solución del sistema [11]. Esto se debe principalmente a la simetría de la matriz del sistema KKT, la cual es aprovechada por el algoritmo de permutación de Bunch y Parlett, a comparación de otras factorizaciones (como LU) y otros métodos de permutación (por ejemplo, permutación simétrica) que ignoran esto y pueden causar que se pierda, generando inestabilidad en la solución del sistema [11] [12].

Esta función se evaluó con las mismas matrices que `quadprog_qp_linsolve`. El tiempo de operación promedio superó el orden de milisegundos en algunos casos en la STM NUCLEO, por lo que este método se consideró más lento en este dispositivo. Cabe destacar, que el mayor tiempo de operación es esperado dado que este método emplea `linsolve` tres veces, para resolver los distintos sistemas que construye, por lo que acumula el tiempo de cada solución. En cuanto a su exactitud numérica, se validó que es posible obtener resultados con un error menor a la tolerancia empleada. No obstante, se observó cierta acumulación de error por lo que dependiendo de las característias del sistema esto puede superar la tolerancia en algunos elementos de la matriz. Por otro lado, al calcular el costo del problema cuadrático con la solución obtenida, la diferencia fue mínima a comparación de los métodos en `quadprog_qp_linsolve`, por lo que no se consideró significativa y, por consecuencia, el rendimiento es comparable con el de las soluciones con LU, QR y SVD.

Figura 25. Evaluación del tiempo de operación de `quadprog_qp_ldlt`



Nota. Tiempo de operación promedio contra el tamaño de la matriz, para matrices $\mathbf{Q}$ y $\mathbf{A}$ $(n \times n)$, para $n = 1 : 1 : 5$. Elaboración propia.

Uso de memoria de los métodos para problemas cuadráticos con restricciones de igualdad

En los cuadros 9 y 10 se incluye el porcentaje estimado de la memoria necesaria para las evaluacions de los métodos para solución de problemas cuadráticos con restricciones de igualdad. Para esto, se siguió la misma metodología que en `matf32` y `linsolve`.

El uso de memoria FLASH en el ESP32 y la STM NUCLEO fue similar que con las librerías anteriores, mientras que se empleó un porcentaje mayor de RAM en estos dispositivos, principalmente par los métodos del espacio nulo y la solución empleando la factorización $\mathrm{LDL}^\top$. De forma similar, la memoria FLASH del Arduino MEGA permaneció por debajo del 10 %, lo que se consideró positivo ya que no incrementó considerablemente con respecto de las demás librerías.

En cuanto a la memoria RAM del Arduino MEGA, contrario a lo esperado, la solución de los problemas cuadráticos con restricciones de igualdad no implicó un incrementó significativo en la memoria del dispositivo. En cambio, si bien en `linsolve` se había empleado alrededor del 80 % de la RAM del Arduino MEGA, emplear LU y QR para la solución de programas cuadráticos requirió alrededor del 50 %. Esto se consideró eficiente, ya que implica que posiblemente pueden emplearse para operaciones adicionales que requieran de estos métodos. En el caso de la solución por medio de SVD, utilizó 79.4 % de la memoria RAM del Arduino MEGA, por lo que el rendimiento fue similar al del solucionador lineal con SVD.

No obstante, la capacidad de RAM del Arduino MEGA no fue suficiente para ejecutar `quadprog_qp_nullspace` y `quadprog_qp_ldlt`, ni en su forma mínima. Para comprobar esto, se implementaron los datos de únicamente un sistema, con matrices $\mathbf{Q}$ y $\mathbf{A}$ de 1×1 (lo que correspondería a una matriz KKT de 2×2). A pesar de eso, el porcentaje de memoria necesario superó el la capacidad del Arduino MEGA, por lo que se descartó que estos dos métodos puedan emplearse en el mismo.

Cuadro 9. Porcentaje de uso de memoria RAM en los dispositivos embebidos para evaluar los métodos de `quadprog` para solución de problemas con restricciones de igualdad

Dispositivo	**Arduino MEGA**	**ESP32**	**STM NUCLEO**
Función	**RAM**	**RAM**	**RAM**
quadprog_qp_nullspace	108.6 %*	10.3 %	10.3 %
quadprog_qp_ldlt	128.6 %*	10.0 %	9.3 %
quadprog_qp_linsolve (con LU)	50.2 %	8.7 %	5.8 %
quadprog_qp_linsolve (con SVD)	79.4 %	8.7 %	5.8 %
quadprog_qp_linsolve (con QR)	50.7 %	8.7 %	5.8 %

Nota. Los resultados marcados con asterisco corresponden al uso de memoria obtenido con una medición con las matrices más pequeñas posibles, ya que aún así superaron la capacidad del dispositivo. Elaboración propia.

Cuadro 10. Porcentaje de uso de memoria FLASH en los dispositivos embebidos para evaluar los métodos de `quadprog` para solución de problemas con restricciones de igualdad

Dispositivo	**Arduino MEGA**	**ESP32**	**STM NUCLEO**
Función	**FLASH**	**FLASH**	**FLASH**
quadprog_qp_nullspace	7.5 %	21.3 %	3.9 %
quadprog_qp_ldlt	8.2 %	21.1 %	4.6 %
quadprog_qp_linsolve (con LU)	3.3 %	20.9 %	4.1 %
quadprog_qp_linsolve (con SVD)	3.6 %	20.9 %	4.1 %
quadprog_qp_linsolve (con QR)	3.6 %	20.9 %	4.1 %

Nota. Elaboración propia.

7.4.3. Método de conjunto activo para solución de problemas con restricciones de desigualdad

En cuanto al método de conjunto activo (`quadprog_sqp`), el algoritmo original está enfocado en resolver problemas cuadráticos únicamente con restricciones de desigualdad, el cual asume para ello que las restricciones de igualdad ya fueron cumplidas previamente (algoritmo 5.4 de [2]). Es decir, corresponde a un caso específico, por lo que, para resolver casos generales en los que haya tanto restricciones de igualdad como desigualdad, se modificó el algoritmo a partir de la definición del problema cuadrático (16) incluyendo las condiciones de igualdad explícitamente. Esto se planteó de la siguiente manera para tomar en cuenta las posibles combinaciones de restricciones:

1. Existen tanto condiciones de igualdad y condiciones de desigualdad activas:

$$\begin{bmatrix} \mathbf{Q} & \mathbf{A}_{eq}^{\top} & \mathbf{A}_{w}^{\top} \\ \mathbf{A}_{eq} & 0 & 0 \\ \mathbf{A}_{w} & 0 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{p}_k \\ \boldsymbol{\lambda}_{eq} \\ \boldsymbol{\lambda}_{w} \end{bmatrix} = - \begin{bmatrix} \mathbf{Q}\mathbf{x}_k + \mathbf{c} \\ \mathbf{A}_{eq}\mathbf{x}_k - \mathbf{b}_{eq} \\ \mathbf{A}_{w}\mathbf{x}_k - \mathbf{b}_{w} \end{bmatrix} \tag{42}$$

2. Existen únicamente condiciones de igualdad:

$$\begin{bmatrix} \mathbf{Q} & \mathbf{A}_{eq}^{\top} \\ \mathbf{A}_{eq} & 0 \end{bmatrix} \begin{bmatrix} \mathbf{p}_k \\ \boldsymbol{\lambda}_{e}q \end{bmatrix} = - \begin{bmatrix} \mathbf{Q}\mathbf{x}_k + \mathbf{c} \\ \mathbf{A}_{eq} - \mathbf{b}_{eq} \end{bmatrix} \tag{43}$$

3. Existen únicamente condiciones de desigualdad y por lo menos una está activa:

$$\begin{bmatrix} \mathbf{Q} & \mathbf{A}_{w}^{\top} \\ \mathbf{A}_{w} & 0 \end{bmatrix} \begin{bmatrix} \mathbf{p}_k \\ \boldsymbol{\lambda}_{w} \end{bmatrix} = - \begin{bmatrix} \mathbf{Q}\mathbf{x}_k + \mathbf{c} \\ \mathbf{A}_{w} - \mathbf{b}_{w} \end{bmatrix} \tag{44}$$

4. Existen únicamente condiciones de desigualdad pero todas están desactivadas (problema sin restricciones):

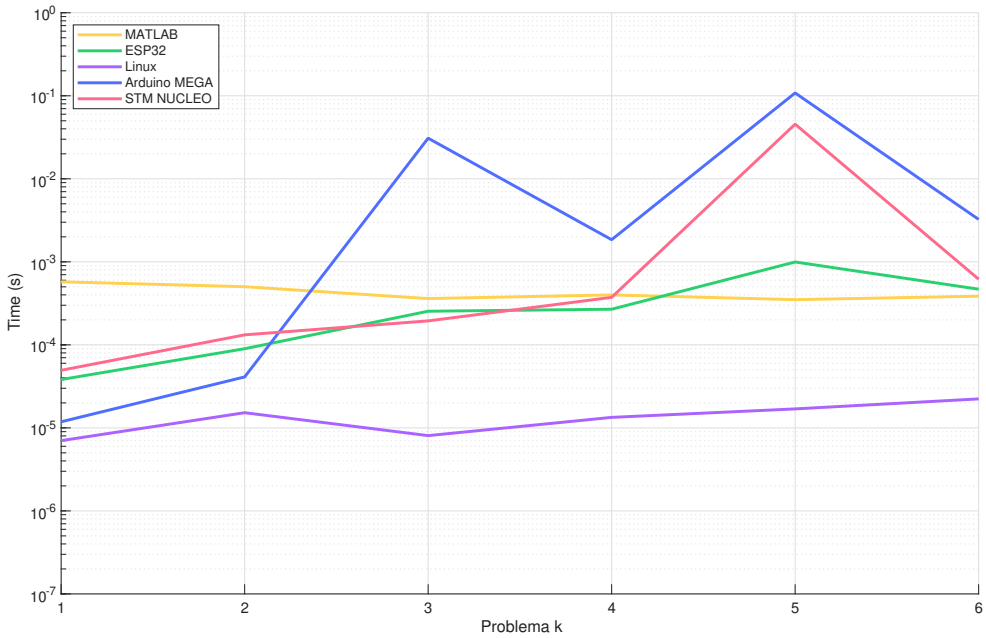
$$\mathbf{Q}\mathbf{p}_k = -(\mathbf{Q}\mathbf{x}_k + \mathbf{c}) \tag{45}$$

En donde, para una iteración dada, $\mathbf{Q}$ y $\mathbf{c}$ son las matrices del costo del problema cuadrático, $\mathbf{A}_{eq}$ y $\mathbf{b}_{eq}$ son la matriz y vector de las restricciones de igualdad, respectivamente. Asímismo, $\mathbf{A}_w$ y $\mathbf{b}_w$ son la matriz y vector de las restricciones de desigualdad activas (conjunto activo o de trabajo), $\mathbf{p}_k$ es la solución propia del sistema KKT, mientras que $\boldsymbol{\lambda}_{eq}$ y $\boldsymbol{\lambda}_w$ son los multiplicadores de Lagrange de las restricciones de igualdad y desigualdad activas, respectivamente.

Dependiendo del caso, el problema podría estar sin restricciones en algunas iteraciones y tener restricciones activas en otras, por lo que la estructura descrita anteriormente permite construir el sistema KKT únicamente con los datos que existen o están activos en el momento, minimizando en cierta medida las dimensiones del sistema a resolver en las iteraciones con menos restricciones. El propósito de esto es reducir el tiempo necesario para la solución del sistema en cada iteración. Por último, cabe resaltar que, actualmente, el método de conjunto activo emplea directamente `linsolve` para la solución del sistema KKT, tanto por simplicidad de la implementación y por ser la opción que menos memoria requiere en los dispositivos (es decir, no se emplea actualmente con el método del espacio nulo ni con la factorización $\mathrm{LDL}^\top$).

En la Gráfica 26 se observó un comportamiento irregular en el tiempo de operación del método de conjunto activo con el Arduino MEGA y la STM NUCLEO, además de que en ambos dispositivos hubo por lo menos un caso con un tiempo mayor al orden de milisegundos, indicando que es un algoritmo más lento en estos dispositivos. En comparación, el tiempo de operación en el ESP32 se mantuvo dentro del rango aceptable y más estable, por lo que sería el dispositivo más adecuado para este algoritmo si el tiempo es crítico.

Figura 26. Evaluación de quadprog_sqp (método de conjunto activo) para resolver problemas con restricciones de igualdad y desigualdad



Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

Uso de memoria del método de conjunto activo

En los cuadros 11 y 12 se incluye el uso de memoria que se midió al evaluar el método de conjunto activo en los distintos dispositivo embebidos. El uso de memoria RAM y FLASH del ESP32 y la NUCLEO, así como en la memoria FLASH del Arduino MEGA se mantuvo similar al de las librerías anteriores, por lo que este método no representó un incremento significativo. En el Arduino MEGA, el uso de memoria RAM fue de 69.7 %, menor al obtenido en las funciones de `linsolve` que estuvo alrededor del 80 %. Por esto mismo, se puede considerar que en todos los dispositivos el uso de memoria fue aceptable, aunque por lo mismo que tienen una mayor capacidad de memoria (1), este método es más apropiado para el ESP32 y la STM NUCLEO para poder tener espacio suficiente y construir aplicaciones basadas en este método, por ejemplo, para el control de modelo predictivo, como se describe en el siguiente capítulo.

Cuadro 11. Porcentaje de uso de memoria RAM en los dispositivos embebidos para evaluar los métodos de `quadprog` para solución de problemas con restricciones de igualdad

Dispositivo	**Arduino MEGA**	**ESP32**	**STM NUCLEO**
Función	**RAM**	**RAM**	**RAM**
quadprog_sqp (active-set)	69.7 %	9.5 %	7.4 %

Nota. Elaboración propia.

Cuadro 12. Porcentaje de uso de memoria RAM en los dispositivos embebidos para evaluar los métodos de `quadprog` para solución de problemas con restricciones de igualdad

Dispositivo	**Arduino MEGA**	**ESP32**	**STM NUCLEO**
Función	**FLASH**	**FLASH**	**FLASH**
quadprog_sqp (active-set)	6.2 %	21.3 %	4.9 %

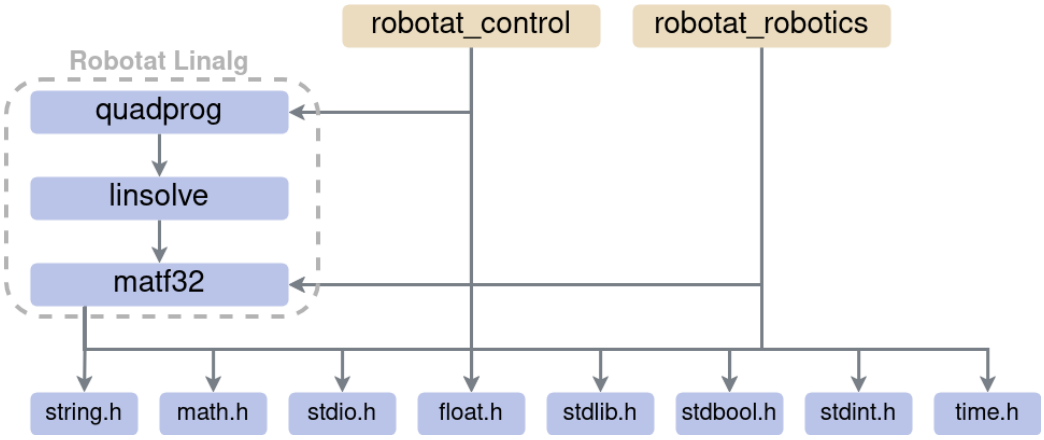
Nota. Elaboración propia.

CAPÍTULO 8

Implementación y validación de librerías para control y robótica

La librería Robotat Linalg se empleó para desarrollar dos nuevas librerías: Robotat Control y Robotat Robotics (al referirse a los archivos del código son `robotat_control` y `robotat_robotics`), para implementar algoritmos de control y robótica, respectivamente, con el propósito de que estas se puedan utilizar en las actividades del laboratorio Robotat en la UVG. Dependiendo de la función, se evaluó su rendimiento como en Robotat Linalg o se emplearon datos de ejemplos representativos para comprobar su correcto funcionamiento. A continuación se describen los resultados obtenidos para cada librería.

Figura 27. Diagrama de dependencias de Robotat Control y Robotat Robotics



Nota. Casi todas las funciones de Robotat Control dependen solamente de `matf32`, siendo la única excepción una de las funciones del control de modelo predictivo en la que se requiere resolver un problema cuadrático y por consecuencia depende de `quadprog`.

8.1. Librería Robotat Control

El desarrollo de esta librería se inició previamente en la Universidad del Valle de Guatemala. Sin embargo, no estaba terminado, por lo que en este trabajo se completaron las rutinas faltantes, se añadieron nuevas rutinas y se validó el funcionamiento de todas estas. A continuación se describe, organizado por temas, los algoritmos de control incluidos, describiendo las funciones más relevantes para cada uno, junto a la evaluación de estas. Para continuar el patrón de Robotat Linalg, se modificaron los nombres de las funciones para iniciar con el prefijo `ctr_`, indicando así que corresponden a la librería de control.

8.1.1. Controlador PID

La librería `robotat_control` incluye una estructura de datos para representar un controlador PID, definiendo los coeficientes del controlador k_P, k_I y k_D, las variables de error e_k, e_{k-1} y e_{k-2} y las variables de entrada en distintos tiempos u_k, u_{k-1} y u_{k-2}. Además, se incluye el método de discretización a utilizar. Las rutinas para el PID incluyen las funciones `ctr_pid_init` y `ctr_pid_set_gains` para inicializar la estructura y asignar valores de k_P, k_I y k_D, así como la función `pid_update` para calcular la siguiente salida controlada a partir de los valores actuales, según el método de discretización seleccionado, que puede ser: Discretización Pura (*Pure Discrete*), que es la más cercana a la definición del PID [15], regla rectangular hacia adelante (*Forward Euler*), rectangular hacia atrás (*Backward Euler*) o regla bilinearlo trapezoidal (Tustin). En este trabajo, se implementaron las ecuaciones de discretización de cada método, según las sustituciones correspondientes para el sistema (24). Estas mismas se emplearon en MATLAB para comparar el rendimiento contra el equivalente exacto.

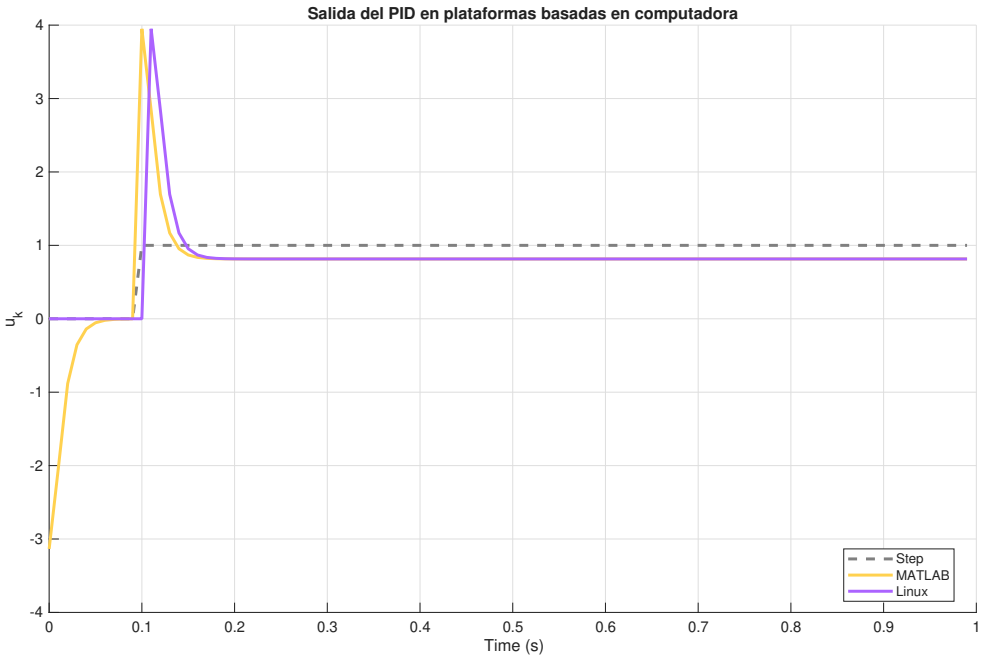
Se graficó la respuesta al escalón unitario de la salida controlada del PID (u_k) contra el tiempo, para cada método de discretización, utilizando los mismos coeficientes del PID en MATLAB y `robotat_control`, esperando un comportamiento similar (Cuadro 13). Para cada método, se ajustaron los coeficientes del controlador hasta obtener un resultado que converge a un valor cercano al escalón (en este caso, 1), demostrando que en todos es posible obtener un resultado adecuado. Para *Forward Euler*, *Backward Euler* y Tustin, la convergencia del resultado fue mejor al emplear `robotat_control` ya que fue más cercano al escalón en comparación con MATLAB, cuyo resultado quedó por debajo del mismo (Figuras 28-35).

Cuadro 13. Coeficientes utilizados para cada discretización del PID

Discretización	K_P	K_I	K_D
Pure Discrete	2.5	0.95	0.5
Forward Euler	1	0.2	0.5
Backward Euler	1	0.2	0.5
Tustin	1	0.2	0.5

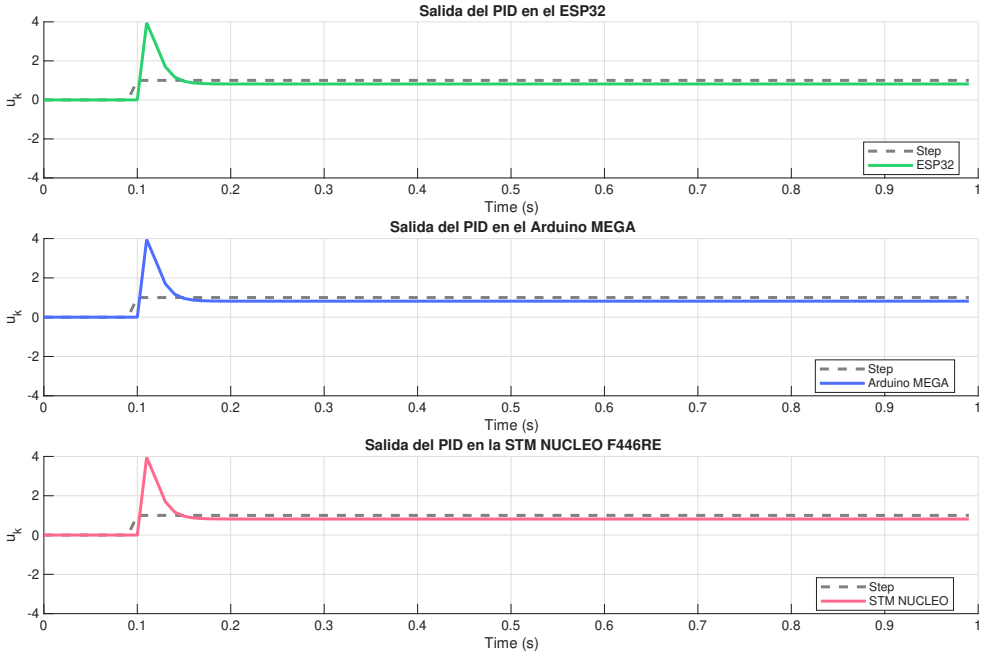
Nota. Salida del PID contra el tiempo de muestreo. Elaboración propia.

Figura 28. Respuesta al escalón de la salida controlada $\mathbf{u}_k$ del PID empleando discretización pura en MATLAB y Linux-amd64



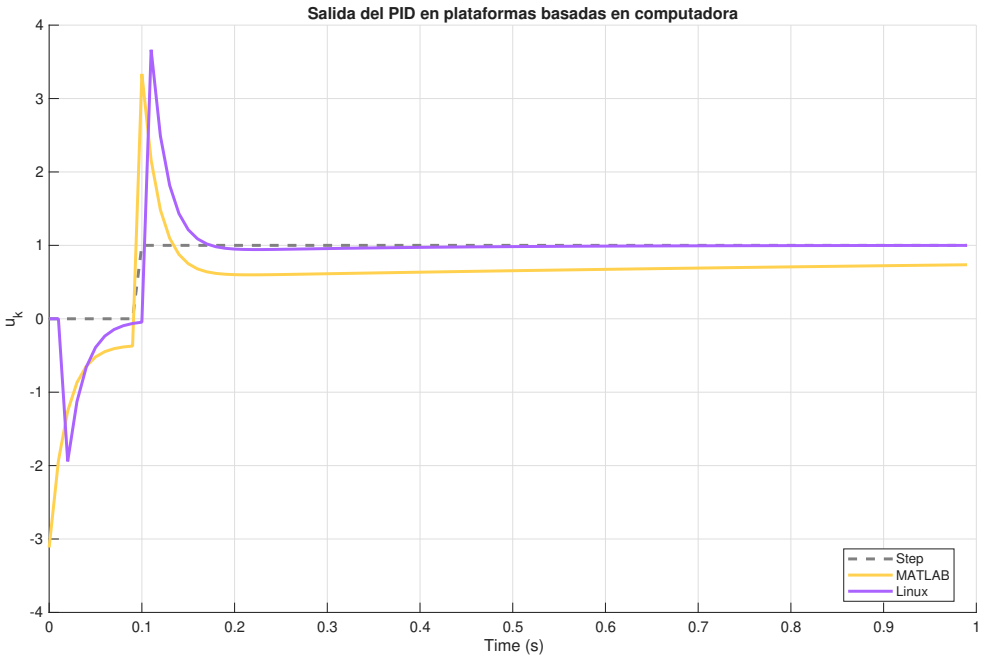
Nota. Salida del PID contra el tiempo de muestreo. Elaboración propia.

Figura 29. Respuesta al escalón de la salida controlada $\mathbf{u}_k$ del PID empleando discretización pura en distintos dispositivos embebidos



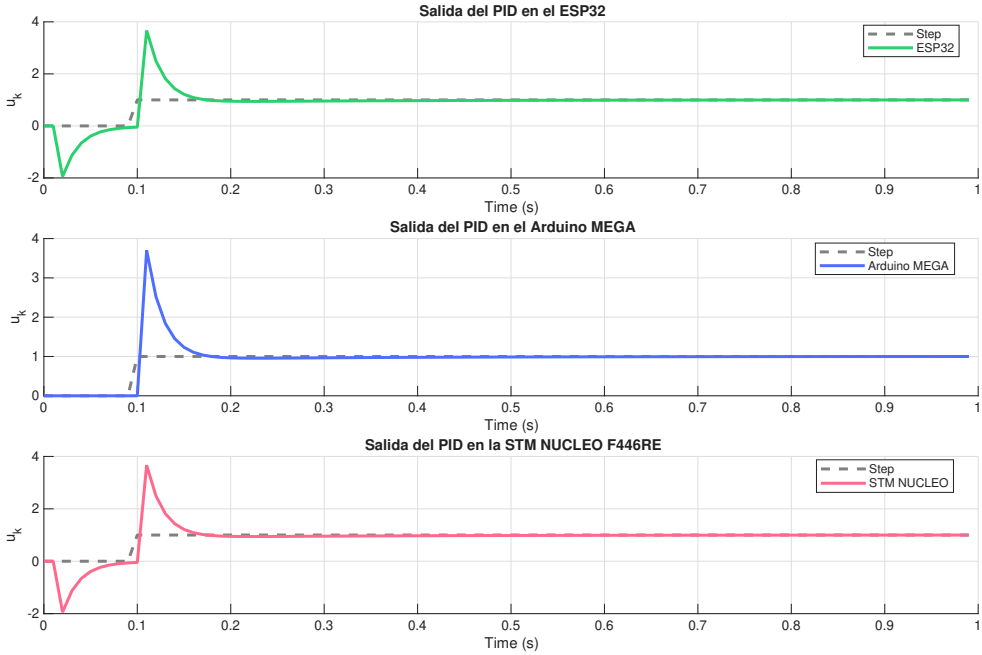
Nota. Salida del PID contra el tiempo de muestreo. Elaboración propia.

Figura 30. Respuesta al escalón de la salida controlada $\mathbf{u}_k$ del PID empleando Forward Euler en MATLAB y Linux-amd64



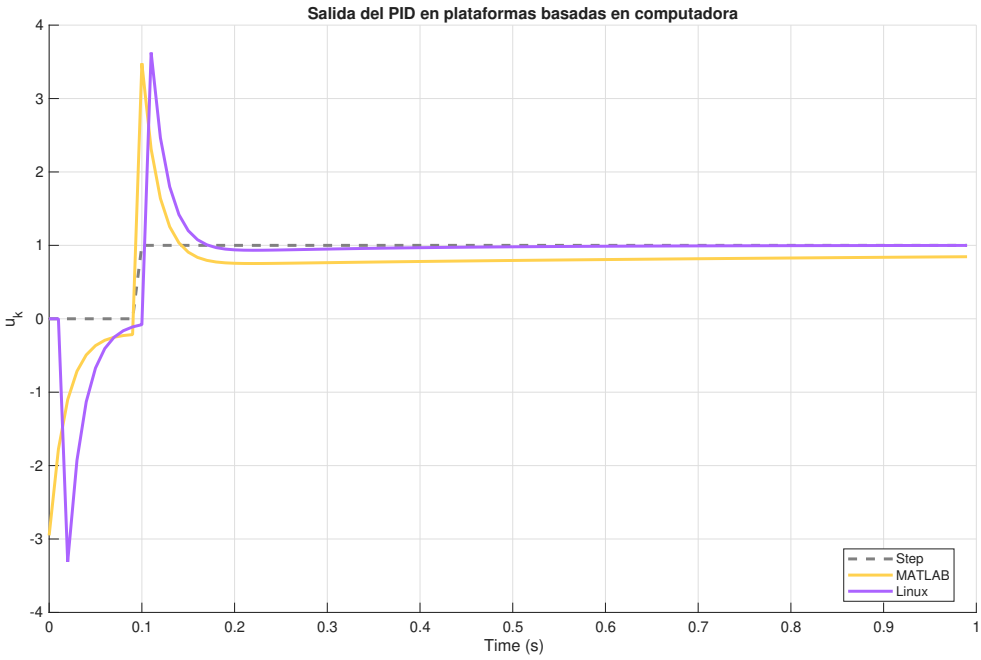
Nota. Salida del PID contra el tiempo de muestreo. Elaboración propia.

Figura 31. Respuesta al escalón de la salida controlada $\mathbf{u}_k$ del PID empleando Forward Euler en distintos dispositivos embebidos



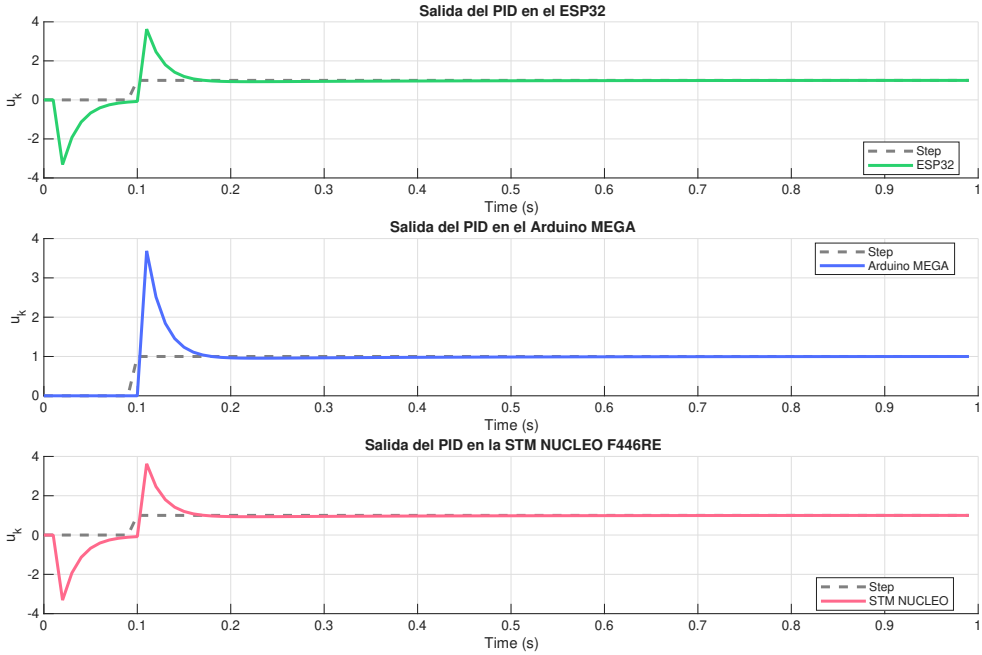
Nota. Salida del PID contra el tiempo de muestreo. Elaboración propia.

Figura 32. Respuesta al escalón de la salida controlada $\mathbf{u}_k$ del PID empleando Backward Euler en MATLAB y Linux



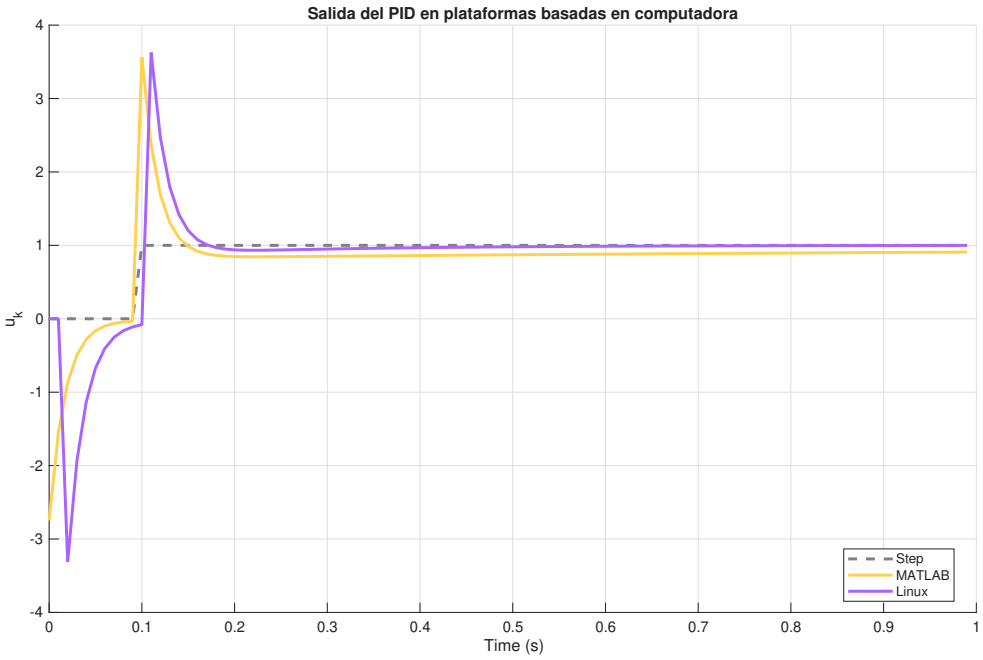
Nota. Salida del PID contra el tiempo de muestreo. Elaboración propia.

Figura 33. Respuesta al escalón de la salida controlada $\mathbf{u}_k$ del PID empleando Backward Euler en distintos dispositivos embebidos



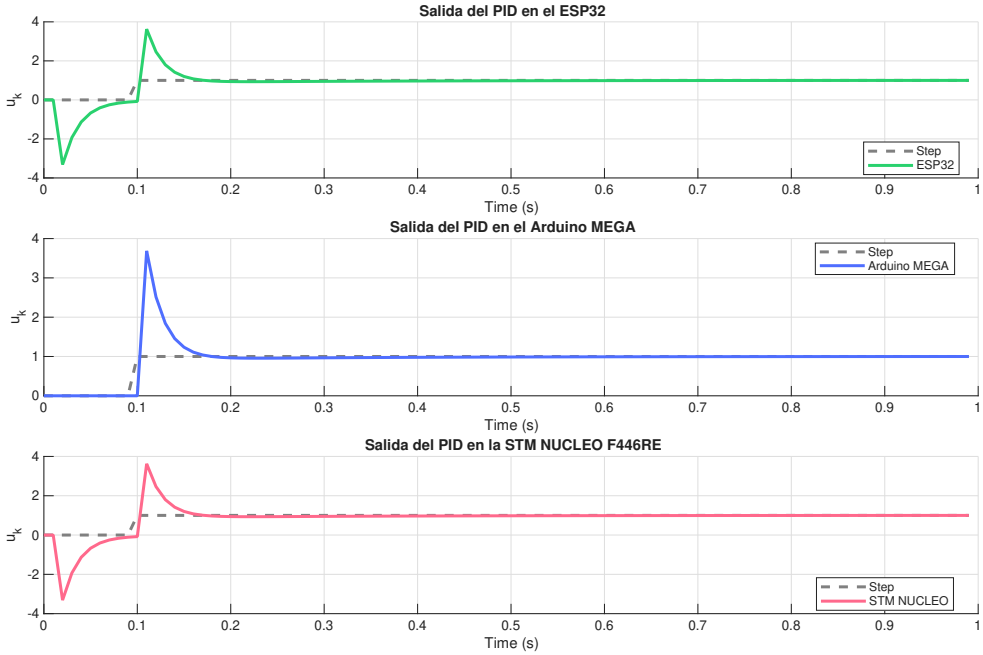
Nota. Salida del PID contra el tiempo de muestreo. Elaboración propia.

Figura 34. Respuesta al escalón de la salida controlada $\mathbf{u}_k$ del PID empleando Tustin en MATLAB y Linux-amd64



Nota. Salida del PID contra el tiempo de muestreo. Elaboración propia.

Figura 35. Respuesta al escalón de la salida controlada $\mathbf{u}_k$ del PID empleando Tustin en distintos dispositivos embebidos



Nota. Salida del PID contra el tiempo de muestreo. Elaboración propia.

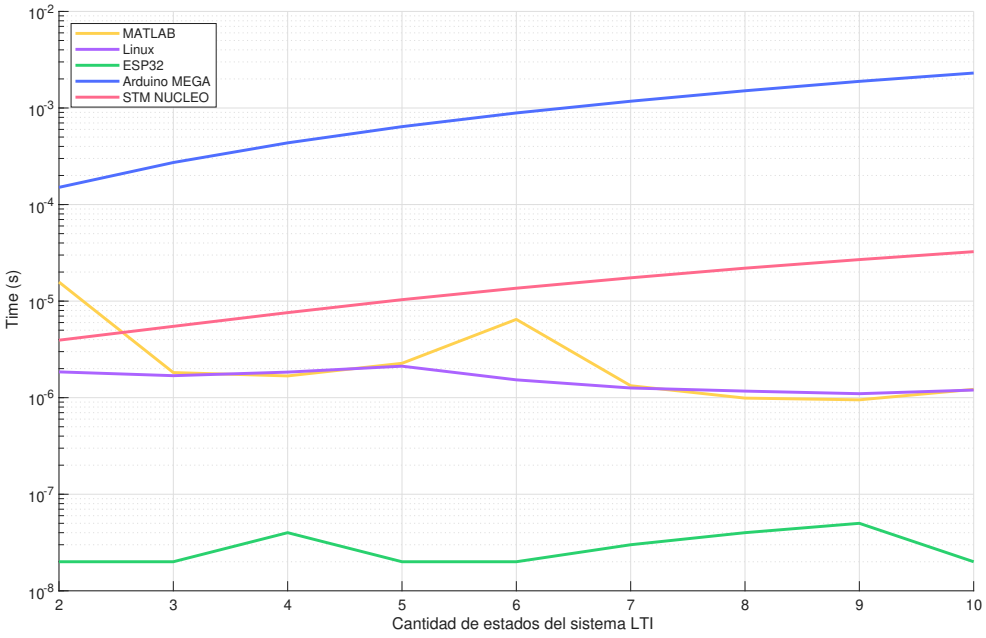
8.1.2. Sistemas lineales

Discretización de sistemas LTI

La librería original incluía una estructura de datos para representar sistemas LTI, así como funciones para inicializarla asignando punteros a los valores correspondientes (`ctr_ss_lti` y `ctr_sys_lti_init`) y la función `ctr_c2d` para discretizar sistemas LTI.

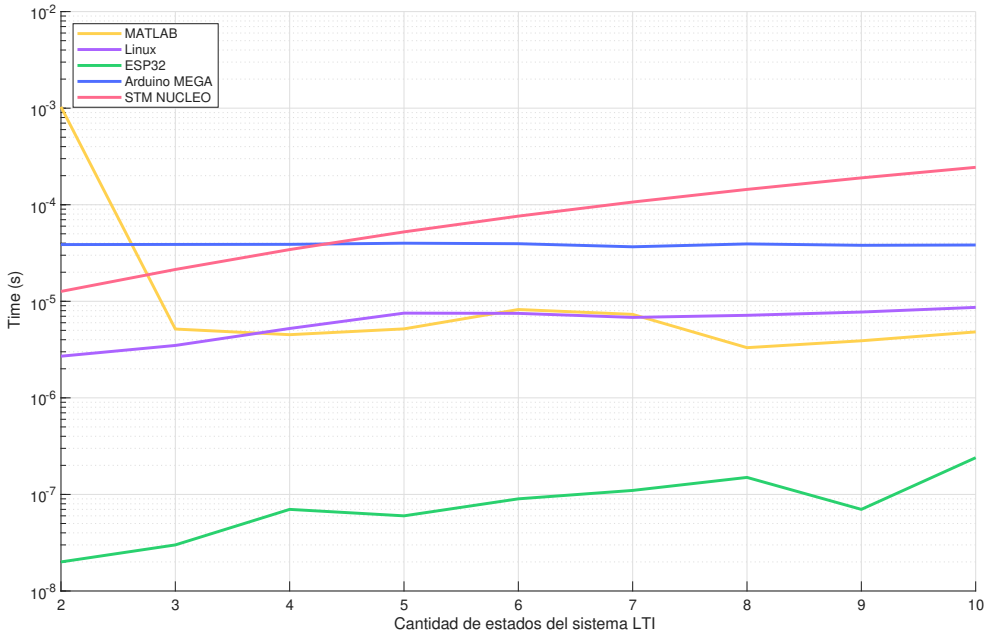
Se implementaron los métodos de *Backward Euler* y Tustin en `ctr_c2d`, que antes contaba únicamente con *Forward Euler*, empleando las sustituciones correspondientes (24). Después, se evaluaron los tres métodos utilizando sistemas desde 2 hasta 10 estados, con matrices **A**, **B** y **C** generadas aleatoriamente con las dimensiones correspondientes, graficando el tiempo promedio de operación por cada sistema (Anexos 36-38). El Arduino MEGA fue el dispositivo con el mayor tiempo de operación al emplear *Forward Euler* y en algunas pruebas de los demás métodos, mientras que la STM NUCLEO tuvo el mayor tiempo con *Backward Euler* y Tustin a partir de sistemas con 5 y 6 estados, respectivamente. Por otro lado, todas las operaciones realizadas tuvieron un tiempo igual o menor al orden de milisegundos, que fue el mayor tiempo que se midió con MATLAB, por lo que se estuvieron dentro de lo esperado (Figuras 36-38). Por otro lado, los tres métodos fueron capaces de generar resultados con exactitud de cinco o más decimales. Aunque al emplear *Backward Euler* y Tustin se observó acumulación de error en algunos casos, lo que puede superar la tolerancia de 1×10^{-5} decimales, y se consideró que podría deberse al calculo de la inversa con `matf32_inv`, por la sensibilidad de esta operación como se describió en el capítulo anterior.

Figura 36. Tiempo de operación de `ctr_c2d` con *Forward Euler*



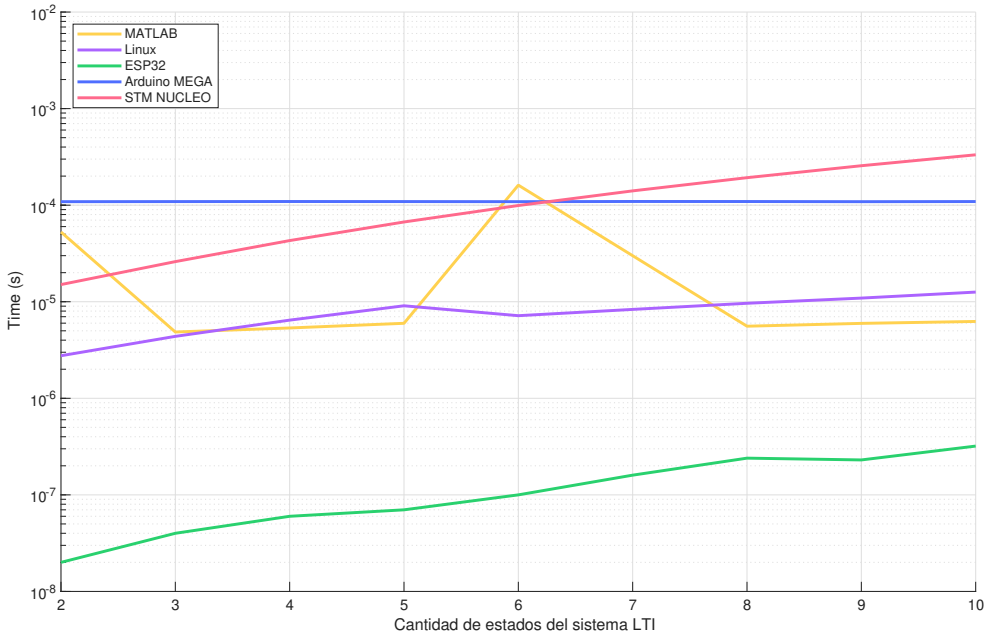
Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

Figura 37. Tiempo de operación de `ctr_c2d` con *Backward Euler*



Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

Figura 38. Tiempo de operación de `ctr_c2d` con Tustin



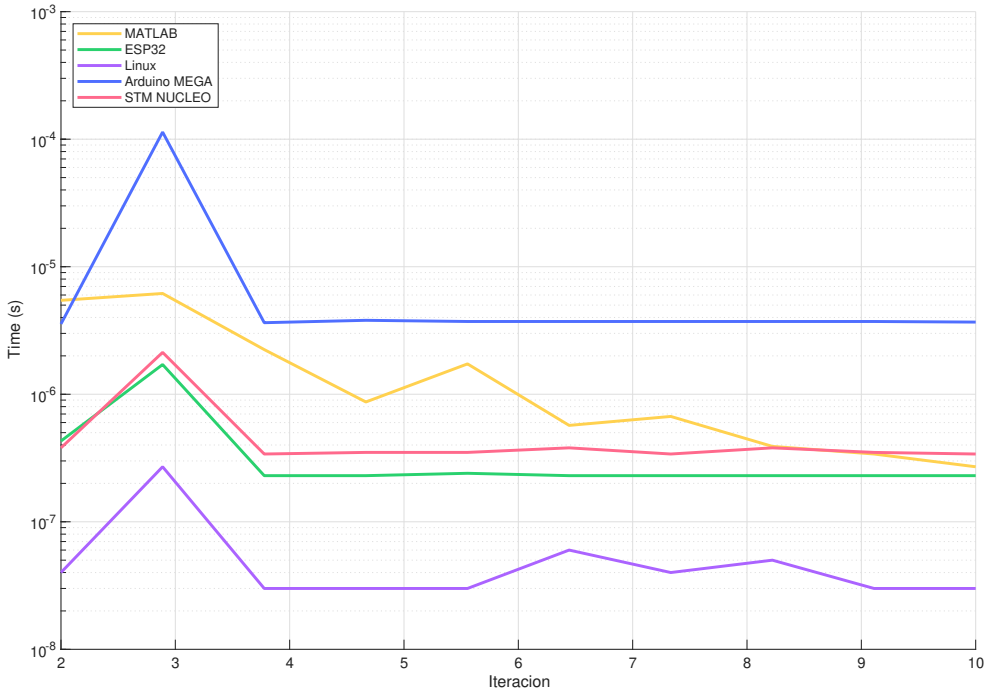
Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

Retroalimenación de estado lineal

Se validó el funcionamiento de `ctr_linear_state_feedback`, que calcula la retroalimentación de estado lineal del sistema (29), empleando vectores de estados $\mathbf{x}_k$ desde 2 hasta 10 estados, con sus respectivas matrices de ganancia $\mathbf{K}$, todas generadas con valores aleatorios. El punto de operación de la entrada utilizada fue $\mathbf{u}_{ss} = 2$ para todas las pruebas.

Para esta función, el Arduino MEGA se consideró el dispositivo menos eficiente en cuanto al tiempo con esta operación ya que superó el tiempo de MATLAB en los sistemas de 3 a 10 estados. Por otro lado, el ESP32 y la STM NUCLEO tuvieron un resultado similar a MATLAB ya que permancieron entre los mismos ordenes de magnitud a lo largo de las distintas pruebas. Aparte de eso, todas las plataformas tuvieron el mayor tiempo en el sistema de 3 estados, indicando que este sistema en específico requirió más tiempo para ser operado, por lo que podría estar relacionado a los datos de entrada empleados para este (Figura 39).

Figura 39. Tiempo de operación de `ctr_linear_state_feedback`



Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

8.1.3. Sistemas no lineales

Linealización de sistemas no lineales

Se validó el funcionamiento de la función de linealización `ctr_linloc` comparándola con su equivalente en MATLAB: `loclin_fast`, que es una función desarrollada y utilizada en los cursos de Sistemas de Control en la UVG. Para esto, se empleó el modelo de un péndulo simple como sistema no lineal, descrito por las siguientes ecuaciones:

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}) = \begin{bmatrix} x_2 \\ -(g/l)sin(x_1) + (1/ml^2)u \end{bmatrix}, \tag{46}$$

en donde m y l corresponden a la masa y longitud del sistema, g es la aceleración de la gravedad, $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ es el vector de estados y $\mathbf{u}$ es el vector de entradas del sistema [15].

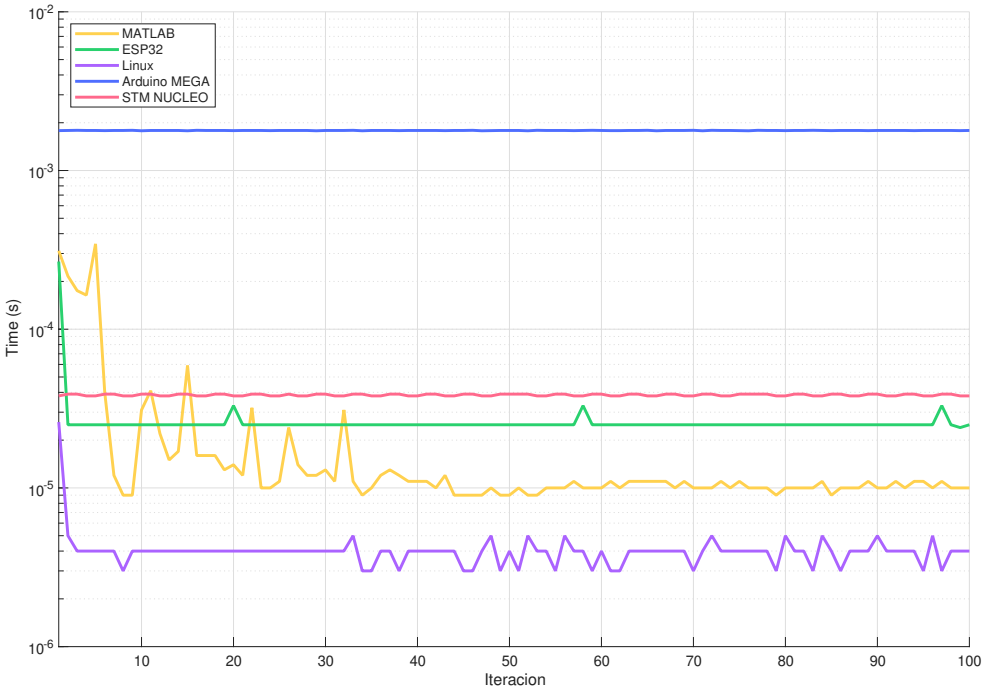
Se graficó el tiempo contra las iteraciones realizadas para la evaluación de esta función. De esta manera, ya que se trata de un sistema específico, se puede visualizar qué tan consistente es el tiempo de operación e identificar además perturbaciones en el mismo. Por ejemplo, en MATLAB el tiempo máximo medido estuvo en el orden de 1×10^{-4} segundos, pero en la gráfica se observó que las primeras mediciones parecieron inestables a diferencia de los demás dispositivos, cuyo tiempo permaneció cercano a constante. Por esto mismo, se tomó en cuenta únicamente el promedio de las mediciones (14-40). Aún así, el tiempo de operación promedio del Arduino MEGA, STM NUCLEO y el ESP32 superó el de la operación en MATLAB, debido a lo cual se consideró que `ctr_linloc` tuvo un rendimiento inferior a MATLAB a pesar de haber logrado una exactitud numérica aceptable dentro de la tolerancia de 1×10^{-5} cifras decimales.

Cuadro 14. Tiempo de operación promedio `ctr_linloc`

Tiempo (s)				
MATLAB	**ESP32**	**Linux-amd64**	**Arduino MEGA**	**STM NUCLEO**
2.43E-05	2.77E-05	4.14E-06	1.79E-03	3.85E-05

Nota. Elaboración propia.

Figura 40. Tiempo de operación de `ctr_linloc` aplicado al modelo de un péndulo simple



Nota. Tiempo de operación promedio contra el número de iteraciones. Elaboración propia.

Cálculo de estado de sistemas no lineales

Se añadió la nueva función `ctr_sys_nonlin_simulate` para calcular el estado siguiente de un sistema no lineal, evaluando la función de la dinámica del mismo escogiendo uno de dos métodos de integración numérica para la linealización, entre *Forward Euler* y Runge-Kutta4 (27-28). Estas mismas ecuaciones se implementaron en MATLAB para comparar el resultado. Además, se empleó el modelo de un péndulo simple para las evaluaciónes de esta función (46).

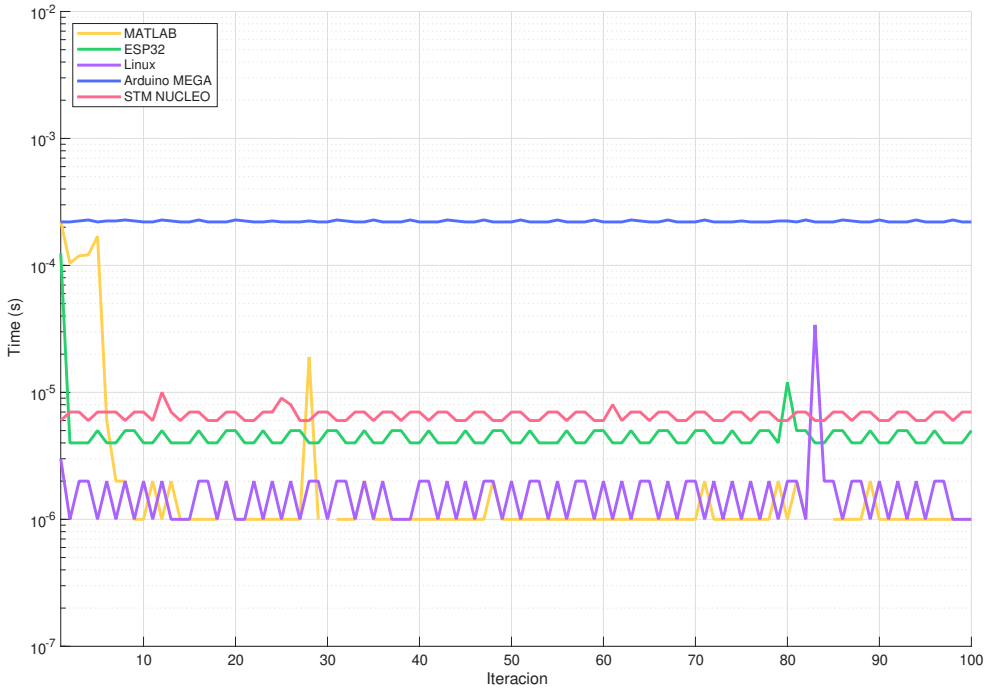
El método de *Forward Euler* fue más rápido que el de Runge-Kutta4 en todas las plataformas, lo cual se encuentra dentro de lo esperado ya que Runge-Kutta4 es una operación más compleja. No obstante, los dispositivos embebidos requirieron más tiempo que MATLAB en la mayoría de mediciones realizadas, por lo que se consideraron menos eficientes que este (Figuras 41-42).

Cuadro 15. Tiempo de operación promedio ctr_sys_nonlin_simulate

Método	Tiempo (s)				
	MATLAB	**ESP32**	**Linux-amd64**	**Arduino MEGA**	**NUCLEO**
FWD EUL	8.62E-06	5.68E-06	1.87E-06	2.22E-04	6.65E-06
RK4	9.52E-06	1.61E-05	2.49E-06	1.00E-03	2.34E-05

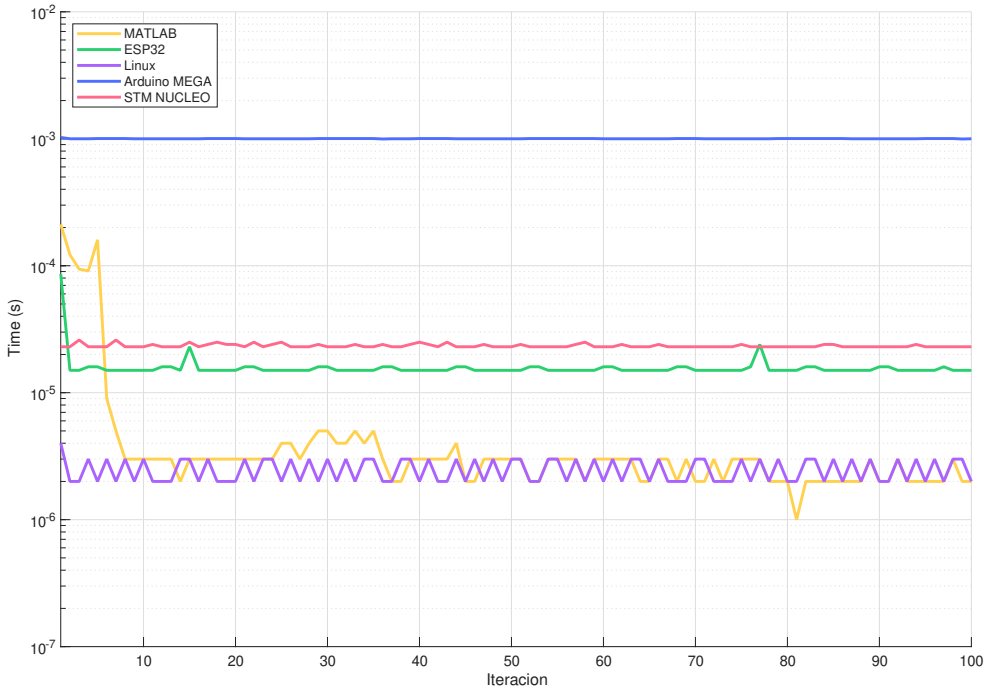
Nota. Notación: FWD EUL corresponde a *Forward Euler*, mientras que RK4 es Runge-Kutta4. Elaboración propia.

Figura 41. Evaluación del tiempo de operación de `ctr_sys_nonlin_simulate` con *Forward Euler*



Nota. Tiempo de operación promedio contra el número de iteraciones. Elaboración propia.

Figura 42. Evaluación del tiempo de operación de `ctr_sys_nonlin_simulate` con Runge-Kutta4



Nota. Tiempo de operación promedio contra el número de iteraciones. Elaboración propia.

8.1.4. Filtro de Kalman

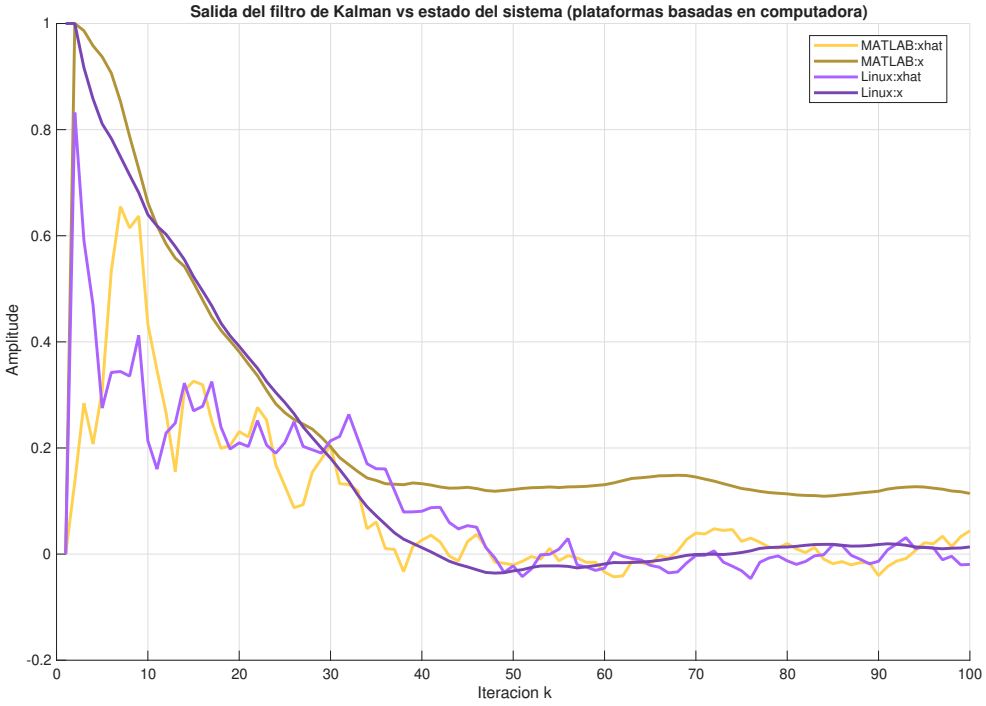
Para validar las funciones que se implementaron previamente en la librería, se implementaron las ecuaciones del filtro de Kalman en MATLAB. Esto debido a que en MATLAB se requieren funciones adicionales para conectar el modelo del filtro a un sistema y simularlo, por lo que no es un equivalente exacto de la implementación en `robotat_control`.

Para validar el funcionamiento, se implementó un sistema LTI con una entrada y un estado, generando ruido con una distribución normal y utilizando las siguientes matrices para el sistema: $\mathbf{A} = \mathbf{1}, \mathbf{B} = \mathbf{1}, \mathbf{C} = \mathbf{1}, \mathbf{D} = \mathbf{0}$ (todas con dimensiones 1×1 para evaluar un sistema sencillo), y calculando la entrada del siguiente estado en función de la corrección del filtro de Kalman: $\mathbf{u} = -0.1\hat{\mathbf{x}}$.

Para analizar los resultados, se graficaron los vectores de estado estimado ($\mathbf{x}$) y real ($\hat{\mathbf{x}}$) del sistema LTI contra el número de iteración correspondiente. El resultado esperado es que tanto $\mathbf{x}$ como $\hat{\mathbf{x}}$ tengan una tendencia hacia cero, demostrando así que el algoritmo converge. Por lo mismo, no se comparó la exactitud numérica ya que lo importante es que el mismo algoritmo alcance la convergencia a cero. Al graficar los resultados, se observó el comportamiento esperado, además, el resultado al emplear `robotat_control` se consideró incluso más estable dado que la trayectoria fue menos irregularidad que la de MATLAB.

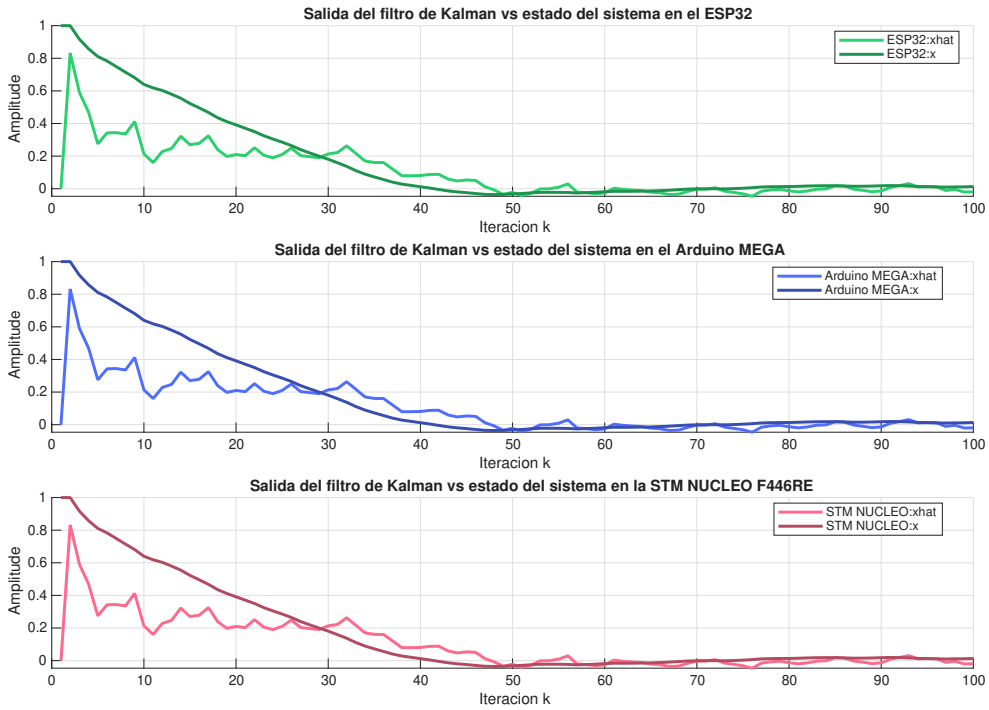
Finalmente, cabe destacar que los resultados fueron iguales o extremadamente similares para todas las plataformas en las que se empleó `robotat_control`, lo cual evidencia que es un algoritmo estable y robusto, y que todos los dispositivos lograron ejecutarlo correctamente (Figuras 43-44).

Figura 43. Vector de estados del sistema LTI con el filtro de Kalman en MATLAB y Linux-amd64



Nota. xhat: estado real obtenido con el Filtro de Kalman, x: estado del sistema estimado con los resultados del Filtro de Kalman. Elaboración propia.

Figura 44. Vector de estados del sistema LTI con el filtro de Kalman en distintos dispositivos embebidos



Nota. xhat: estado real obtenido con el Filtro de Kalman, x: estado del sistema estimado con los resultados del Filtro de Kalman. Elaboración propia.

8.1.5. Modelo de control predictivo (MPC)

Se añadió una estructura de datos en `robotat_control` para representar un control de modelo predictivo de disparo (*shooting*), incluyendo el sistema LTI con el cual trabajar, el programa cuadrático a solucionar, la longitud del horizonte N del MPC, así como las matrices de predicción $\mathbf{M}_x$ u $\mathbf{M}_c$ y de penalización $\mathbf{Q}$, $\mathbf{R}$ y $\mathbf{S}$ del MPC. Se añadieron funciones individuales para calcular las distintas matrices: `ctr_mpc_set_M` para $\mathbf{M}_x$, `ctr_mpc_set_C` para M_c, `ctr_mpc_set_qpQ` y `ctr_mpc_set_qpc` para las matrices del problema cuadrático.

Dado el enfoque para resolver problemas densos y pequeños, el límite de dimensiones para las matrices en Robotat Linalg está configurado actualmente como 10×10 (100 elementos). Debido a eso, en `robotat_control` se implementaron las matrices de predicción del MPC por medio de punteros, calculando una instancia de cada submatriz con la que se construyen las matrices $\mathbf{M}_c$ y $\mathbf{M}_x$ (30). Luego, se asignaron punteros hacia cada submatriz en un arreglo, para ubicarlos en el respectivo número de elemento de lo que sería la matriz $\mathbf{M}_c$ o $\mathbf{M}_c$, de manera que se pueden indexar como una matriz normal. Esto es equivalente a emplear matrices de matrices en MATLAB. De esta manera, el problema se puede escalar a un mayor horizonte siempre que las matrices individuales que conforman a $\mathbf{M}_c$ y $\mathbf{M}_x$ no superen el límite de tamaño establecido.

Adicionalmente, se empleó el mismo algoritmo de conjunto activo de la librería quadprog (`quadprog_sqp`) para la solución del problema cuadrático, en lugar del solucionador cuadrático propio de MATLAB, para verificar que el problema puede solucionarse con este algoritmo específico, como parte de la validación.

Para validar el modelo, se utilizó el siguiente sistema de ejemplo de [17]:

$$\mathbf{A} = \begin{bmatrix} 1.1 & 2 \\ 0 & 0.95 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 0 \\ 0.0787 \end{bmatrix}, \quad \mathbf{C} = \begin{bmatrix} -1 & 1 \end{bmatrix}, \tag{47}$$

utilizando como estado inicial un punto escogido arbitrariamente: $\mathbf{x}_k = [1 \quad 1]$ y matrices de penalización $\mathbf{Q} = \mathbf{C}^\top \mathbf{C}$ y $\mathbf{R} = 0.01$. Por último, se utilizaron las restricciones superior $ub = 10$ e inferior $lb = -10$, para los casos con restricciones (31-32).

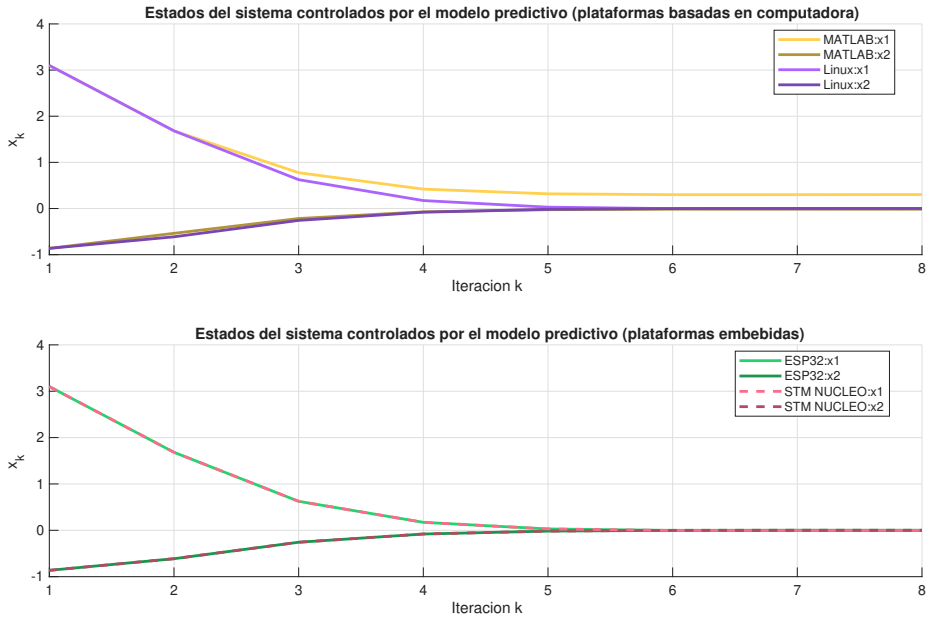
Para analizar los resultados, se graficaron las trayectorias de los dos estados del sistema, de manera que se observó su convergencia hacia cero, validando así que el algoritmo funcionó correctamente tanto en MATLAB como al emplear `robotat_control` (45-51). En el caso del MPC con restricciones únicamente en la entrada, el resultado en MATLAB y las plataformas con `robotat_control` fue el mismo, evidenciado por la superposición de la trayectoria en MATLAB y en Linux (48). Por el contrario, los casos sin restricciones y con restricciones de estado tuvieron una mejor convergencia en `robotat_control` para $\mathbf{x}_1$, dado que en MATLAB este permaneció por encima de cero en lugar de converger a este valor como los demás resultados (Figura 45-51). Por consecuencia, se puede interpretar que el algoritmo en robotat_control tuvo un mejor rendimiento que el de MATLAB.

Además de eso, al graficar la trayectoria del vector de entrada del sistema controlado por el MPC y el costo del problema cuadrático asociado, se vio que convergieron a cero, según lo esperado, por lo que se determinó que el algoritmo estabilizó correctamente el sistema LTI empleado. Esto se puede observar en las Figuras 46, 49 y 52).

Aparte de eso, en las Figuras 53, 50 y 53, se puede observar la trayectoria del costo del problema cuadrático para los tres escenarios del MPC. El resultado obtenido fue significativamente distinto entre MATLAB y `robotat_control`, en los casos con restricciones en el problema, especialmente por el cambio de signo en las trayectorias. Además de esto, la magnitud del costo en `robotat_control` fue mayor que en MATLAB. Estas diferencias se atribuyeron principalmente a que las trayectorias generadas para la solución del problema no fueron numéricamente iguales a MATLAB. A pesar de eso, los resultados se consideraron correctos debido a que las trayectorias de los estados, entradas y costo presentan la misma tendencia a cero que en MATLAB, que es lo más importante de este algoritmo.

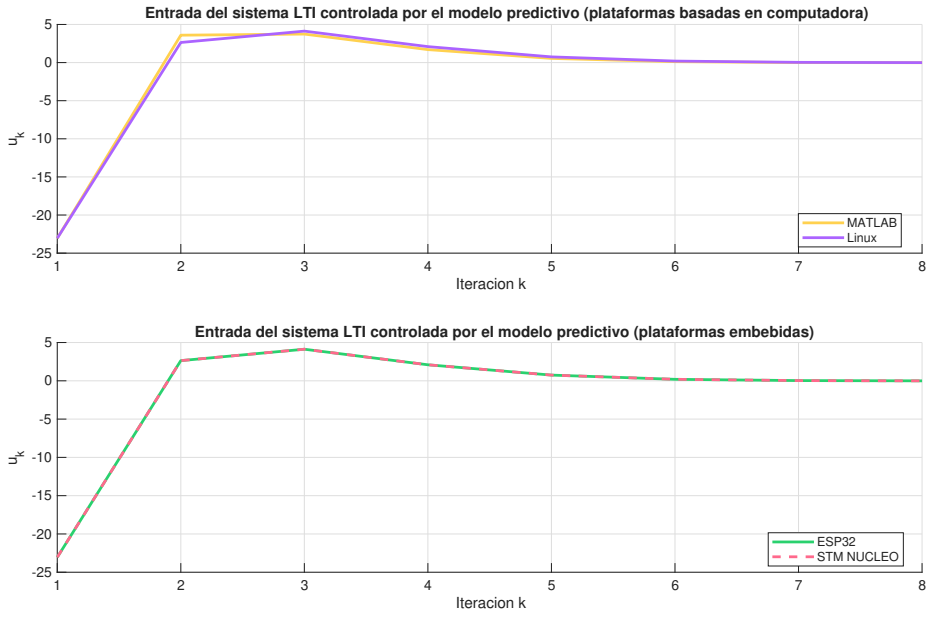
Resultados del MPC sin restricciones

Figura 45. Vector de estados del sistema LTI con un MPC (sin restricciones)



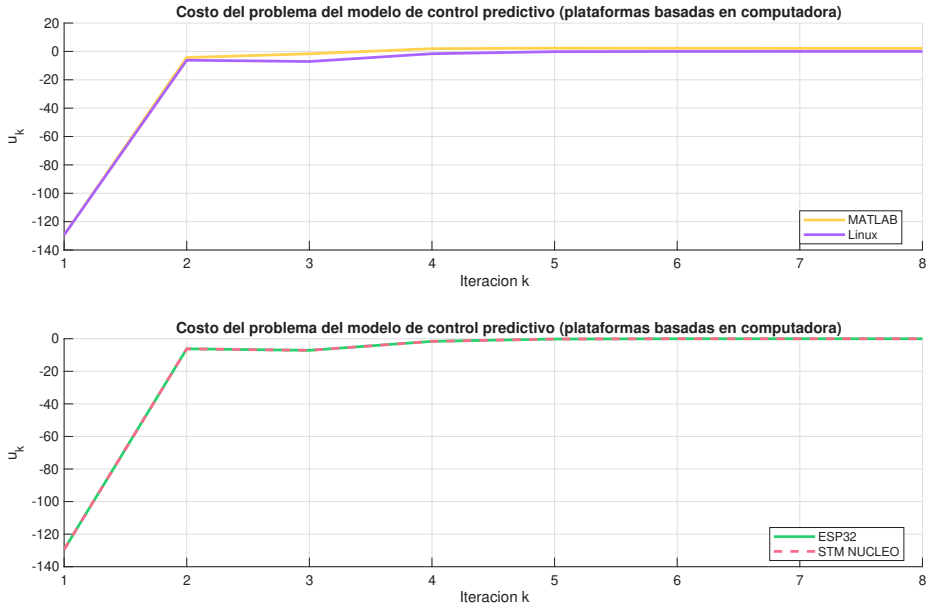
Nota. Estado contra la iteración del horizonte del MPC. Elaboración propia.

Figura 46. Vector de entrada del sistema LTI con un MPC (sin restricciones)



Nota. Entrada contra la iteración del horizonte del MPC. Elaboración propia.

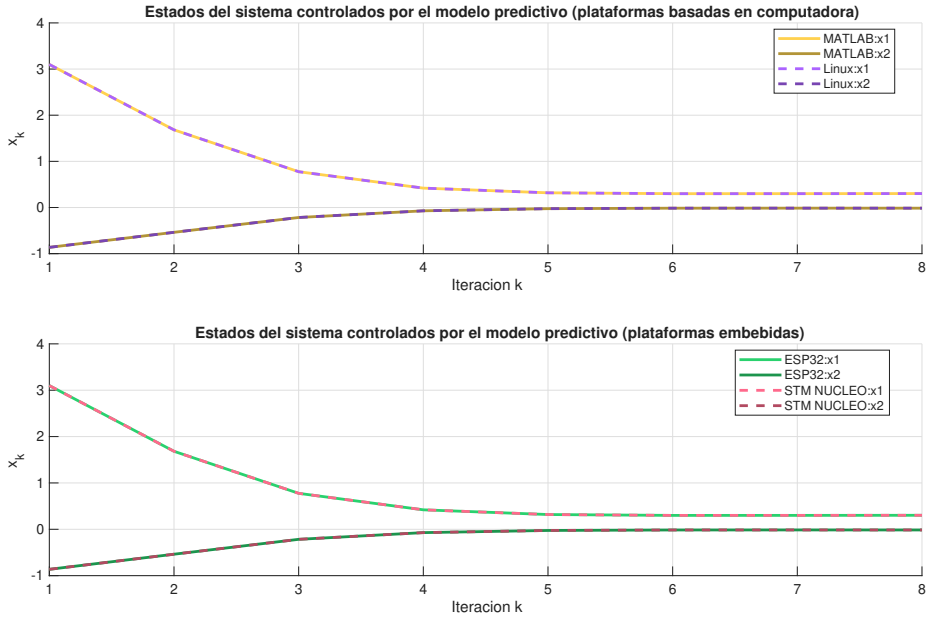
Figura 47. Costo del problema cuadrático del MPC (sin restricciones)



Nota. Costo contra la iteración del horizonte del MPC. Elaboración propia.

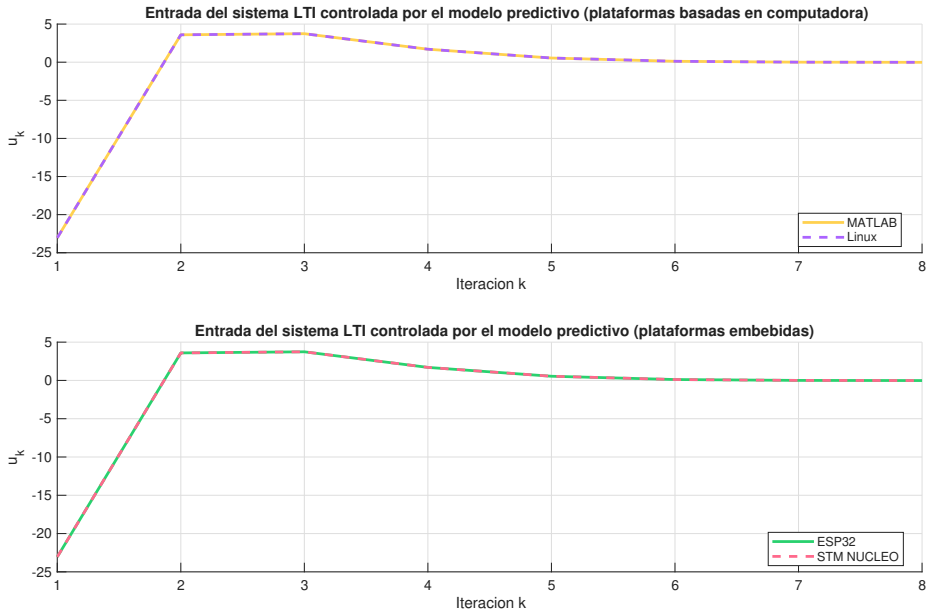
Resultados del MPC con restricciones en la entrada

Figura 48. Vector de estados del sistema LTI con un MPC (con restricciones en la entrada)



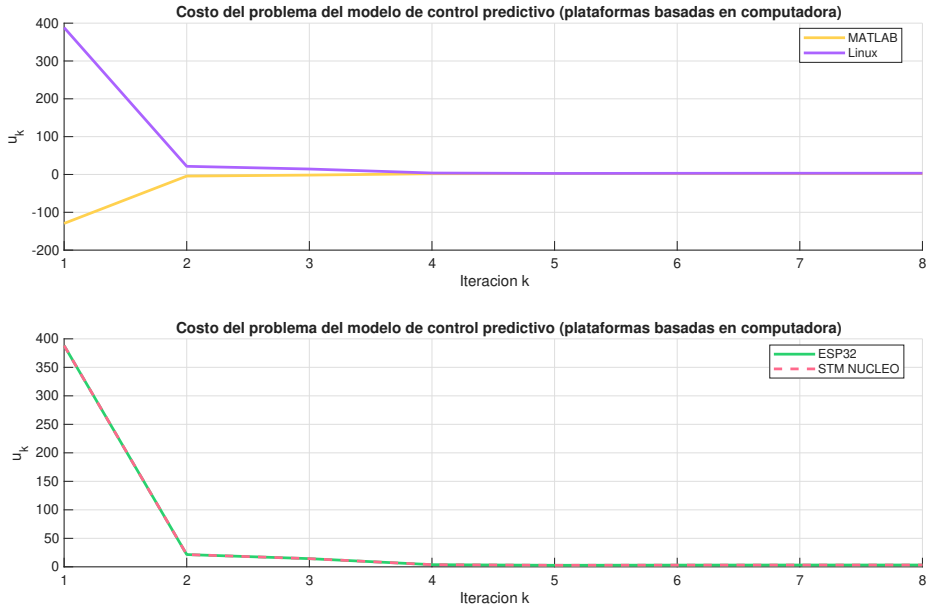
Nota. Estado contra la iteración del horizonte del MPC. Elaboración propia.

Figura 49. Vector de entrada del sistema LTI con un MPC (con restricciones en la entrada)



Nota. Entrada contra la iteración del horizonte del MPC. Elaboración propia.

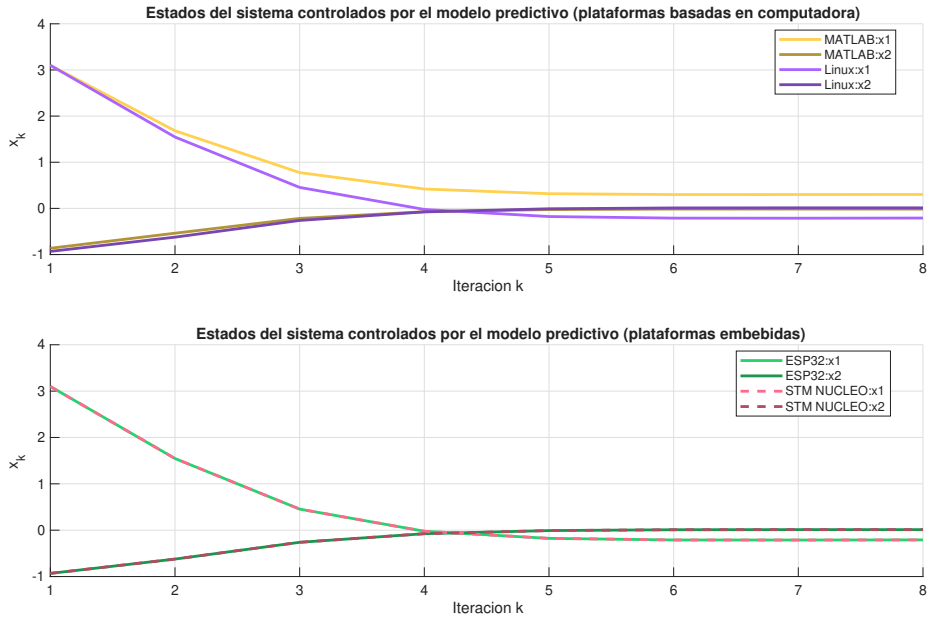
Figura 50. Costo del problema cuadrático del MPC (con restricciones en la entrada)



Nota. Costo contra la iteración del horizonte del MPC. Elaboración propia.

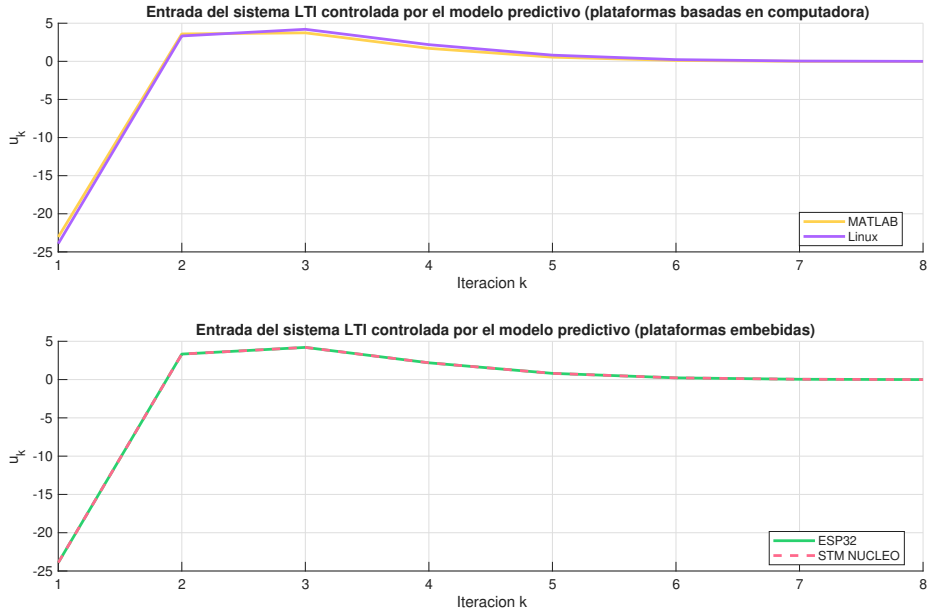
Resultados del MPC con restricciones de estado

Figura 51. Vector de estados del sistema LTI con un MPC (con restricciones en la entrada y estado)



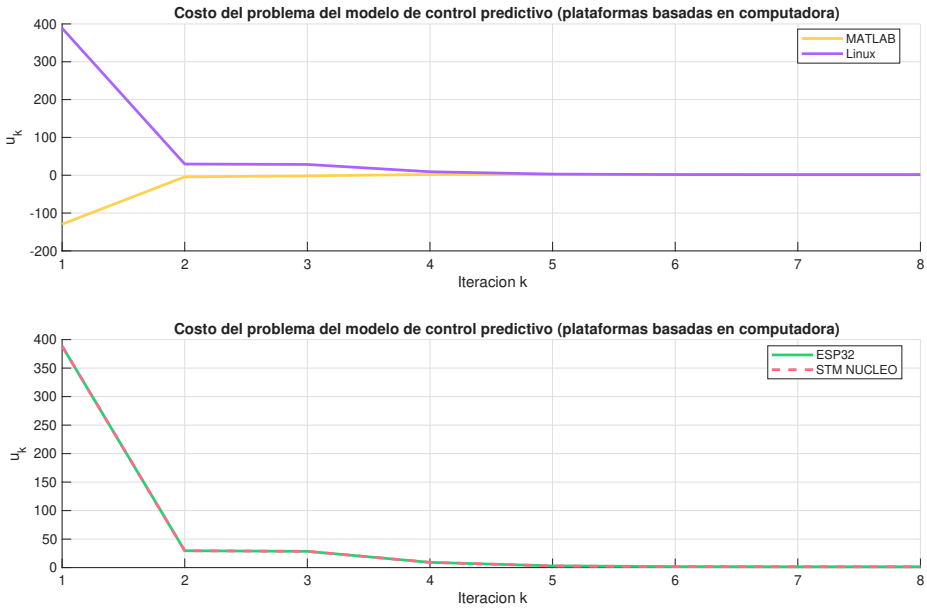
Nota. Estado contra la iteración del horizonte del MPC. Elaboración propia.

Figura 52. Vector de entrada del sistema LTI con un MPC (con restricciones en la entrada y estado)



Nota. Entrada contra la iteración del horizonte del MPC. Elaboración propia.

Figura 53. Costo del problema cuadrático del MPC (con restricciones en la entrada y estado)



Nota. Costo contra la iteración del horizonte del MPC. Elaboración propia.

Por último, la evaluación del uso de memoria con el MPC se incluye en los cuadros 16 y 17. Cabe destacar que el MPC no se logró implementar en el Arduino MEGA dado que la memoria que este algoritmo requiere supera la capacidad de este Arduino (126.9 % en el problema sin restricciones, por lo que no se evaluó la memoria con restricciones ya que tampoco sería posible ejecutarlo). En cambio, tanto en el ESP32 como en la STM NUCLEO, la memoria RAM utilizada fue menor al 15 %, mientras que en la FLASH se utilizó un porcentaje similar que al evaluar las librerías de Robotat Linalg. Debido a esto, se consideró que ambos dispositivos son capaces de ejecutar el MPC de forma eficiente y aún conservan suficiente capacidad para implementar operaciones o algoritmos adicionales.

Cuadro 16. Porcentaje de uso de memoria RAM en los dispositivos embebidos para evaluar los métodos del MPC

Dispositivo	**Arduino MEGA**	**ESP32**	**STM NUCLEO**
Función	**RAM**	**RAM**	**RAM**
MPC sin restricciones	126.9 %	12 %	13.6 %
MPC con restricciones	-	12 %	13.6 %

Nota. Elaboración propia.

Cuadro 17. Porcentaje de uso de memoria RAM en los dispositivos embebidos para evaluar los métodos del MPC

Dispositivo	**Arduino MEGA**	**ESP32**	**STM NUCLEO**
Función	**FLASH**	**FLASH**	**FLASH**
MPC sin restricciones	10.5 %	21.3 %	5.1 %
MPC con restricciones	-	21.4 %	5.2 %

Nota. Elaboración propia.

8.2. Librería Robotat Robotics

Estructura de la librería

Esta librería está basada en la librería para MATLAB: *Robotics Toolbox* de Peter Corke [18]. Se incluyeron diferentes funciones para generar matrices de rotación a partir de un ángulo θ, cuaterniones y con ángulos de Euler y Cardán, acorde a sus definiciones teóricas [18] [19]. Se implementaron matrices de rotación en tres dimensiones, ya que dos dimensiones es un caso específico de esto cuando no hay rotación en uno de los ejes. Asímismo, se incluyeron funciones para aplicar transformaciones homogéneas (37), calcular la inversa de una transformación homogénea y varias funciones de conversión entre matrices, ángulos y cuaterniones (Cuadro 18). Todas las matrices y vectores en en esta librería tienen dimensiones fijas, siendo el tamaño máximo de 4×4.

Aparte de eso, se crearon estructuras de datos para representar marcos de referencia, puntos de coordenadas y cuaterniones, junto a sus respectivas funciones para inicializarlas. Para los marcos de referencia, esto incluye la matriz de transformación homogénea $\mathbf{T}$ (4×4), la matriz de rotación y vector de coordenadas asociados ($\mathbf{R}$ (3×3) y $\mathbf{t}$ (3×1)), además de nombre o número del marco correspondiente. Para los puntos de coordenadas fue similar, incluyendo una matriz (4×1) y el indicador del marco asociado. En el caso de los cuaterniones se asignan los valores escalares del mismo a, b, c y d, a partir de la definición de cuaternión: $q = a + bi + cj + dk$. Finalmente, se añadieron funciones auxiliares para imprimir las estructuras en la terminal, verificar dimensiones de las matrices, evitar divisiones entre cuaterniones nulos y para verificar que los marcos de transformación y puntos de coordenadas se ingresen en el orden correcto antes de operar.

Cuadro 18. Funciones de Robotat Robotics evaluadas y su equivalente en MATLAB

Robotat Robotics	MATLAB	Operación
rob_rotx	rotx	Genera una matriz de rotación en x (3x3).
rob_roty	roty	Genera una matriz de rotación en y (3x3).
rob_rotz	rotz	Genera una matriz de rotación en z (3x3).
rob_trotx	trotx	Genera y aplica una rotación con rob_rotx
rob_troty	troty	Genera y aplica una rotación con rob_roty
rob_trotz	trotz	Genera y aplica una rotación con rob_rotz
rob_apply_transform	homtrans	Aplica una transformación homogénea: ${}^{A}p = {}^{A}T_{B} \cdot {}^{B}p$
rob_inv_transform	inv	Inversa de una transformación homogénea
rob_update_transform	rt2tr	Asigna rotación y coordenadas a una transformación
rob_quat_add	q+q	Suma de cuaterniones
rob_quat_sub	q-q	Resta de cuaterniones
rob_quat_scale	q*c	Multiplicación cuaternión-escalar
rob_quat_mul	q*q	Multiplicación cuaternión-cuaternión
rob_quat_conj	q.conj	Conjugado de un cuaternión
rob_quat_norm	q.norm	Norma de un cuaternión
rob_quat_inv	q.inv	Inversa de un cuaternión
rob_rot2tr	r2t	Conversión: rotación a transformación homogénea
rob_tr2rot	t2r	Conversión: transformación homogénea a rotación
rob_rpy2rot	rpy2r	Conversión: ángulos de Cardán a rotación
rob_rpy2tr	rpy2tr	Conversión: ángulos de Cardán a transformación
rob_eul2rot	eul2r	Conversión: ángulos de Euler a rotación
rob_eul2tr	eul2tr	Conversión: ángulos de Euler a transformación
rob_rot2quat	q.tr2q	Conversión: rotación a cuaternión
rob_tr2quat	q.tr2q	Conversión: transformación homogénea a cuaternión
rob_rpy2quat	q.rpy	Conversión: ángulos de Cardán a cuaternión
rob_eul2quat	q.eul	Conversión: ángulos de Euler a cuaternión
rob_tr2rpy	tr2rpy	Conversión: transformación a ángulos de Cardán
rob_tr2eul	tr2eul	Conversión: transformación a ángulos de Euler
rob_quat2rot	uq.q2r	Conversión: cuaternión a rotación
rob_quat2tr	uq.T	Conversión: cuaternión a transformación homogénea
rob_quat2rpy	uq.torpy	Conversión: cuaternión a ángulos de Cardán
rob_quat2eul	uq.toeul	Conversión: cuaternión a ángulos de Euler

Nota. Elaboración propia.

Resultados de la evaluación de la librería

Se evaluaron las funciones para operaciones con matrices de rotación, transformaciones homogéneas, cuaterniones y conversión entre todas estas representaciones. Su resultado numérico y tiempo de operación se comparó con sus equivalentes de la *Robotics Toolbox*. Para esto, se graficó el tiempo de operación de cada función evaluada y se calculó el tiempo promedio de cada una (Cuadros 19 y 20). Dado que todas las operaciones en esta librería tienen dimmensiones fijas en las matrices y vectores, se buscó evaluar principalmente la estabilidad del tiempo de operación, es decir, qué tan consistente es en distintas mediciones.

Cuadro 19. Tiempo promedio de las funciones de `robotat_robotics` en las plataformas basadas en computadora

Función	Tiempo (s)	
	MATLAB	**Linux-amd64**
rob_transl	2.99E-05	1.18E-06
rob_rotx	9.17E-05	1.50E-06
rob_roty	2.25E-05	1.31E-06
rob_rotz	2.13E-05	1.26E-06
rob_trotx	1.93E-05	1.91E-06
rob_troty	2.02E-05	1.65E-06
rob_trotz	2.08E-05	1.64E-06
rob_apply_transform	2.68E-05	1.15E-06
rob_inv_transform	4.86E-06	2.10E-06
rob_quat_add	1.23E-04	1.08E-06
rob_quat_sub	5.67E-05	1.05E-06
rob_quat_scale	7.88E-05	1.13E-06
rob_quat_mul	1.69E-04	1.17E-06
rob_quat_conj	6.36E-05	1.25E-06
rob_quat_norm	3.17E-05	1.09E-06
rob_quat_inv	8.48E-05	1.22E-06
rob_rot2tr	3.49E-05	1.67E-06
rob_tr2rot	5.97E-05	1.43E-06
rob_rpy2rot	1.80E-02	3.13E-06
rob_rpy2tr	1.65E-03	3.17E-06
rob_update_transform	2.58E-05	1.54E-06
rob_eul2rot	1.06E-04	2.84E-06
rob_eul2tr	1.15E-03	3.09E-06
rob_rot2quat	1.08E-04	1.25E-06
rob_tr2quat	2.61E-05	1.34E-06
rob_rpy2quat	1.42E-03	3.14E-06
rob_eul2quat	2.02E-04	3.29E-06
rob_tr2rpy	1.30E-03	1.58E-06
rob_tr2eul	1.10E-04	1.47E-06
rob_quat2rot	3.68E-05	1.00E-06
rob_quat2tr	9.41E-05	1.26E-06
rob_quat2rpy	2.32E-04	1.32E-06
rob_quat2eul	1.36E-04	1.48E-06

Nota. Elaboración propia.

Cuadro 20. Tiempo promedio de las funciones de `robotat_robotics` en las plataformas embebidas

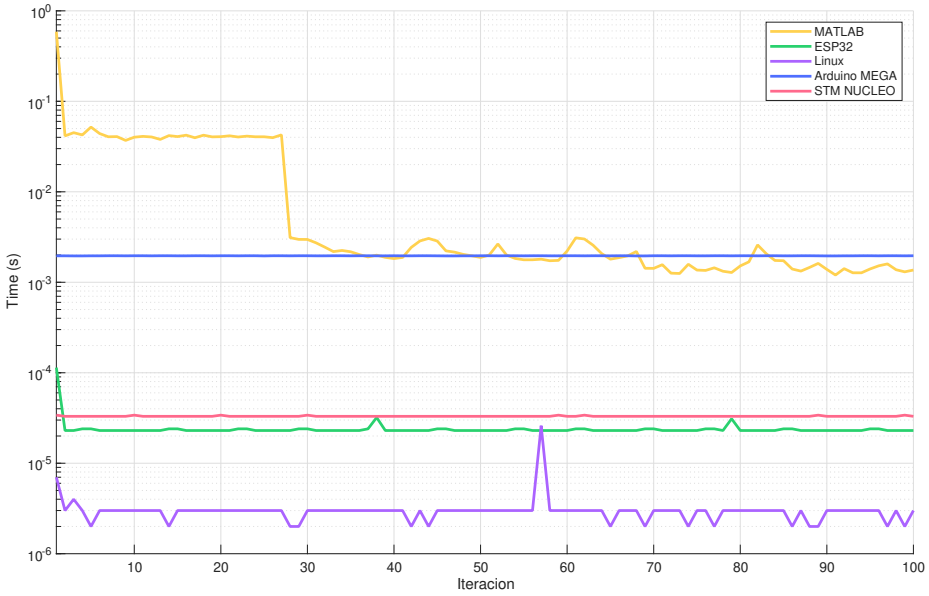
Función	Tiempo (s)		
	ESP32	**Arduino MEGA**	**STM NUCLEO**
rob_transl	1.34E-06	2.92E-05	1.78E-06
rob_rotx	7.17E-06	3.23E-04	8.71E-06
rob_roty	6.26E-06	3.23E-04	8.83E-06
rob_rotz	6.25E-06	3.23E-04	8.82E-06
rob_trotx	7.37E-06	3.66E-04	1.10E-05
rob_troty	7.24E-06	3.66E-04	1.10E-05
rob_trotz	7.20E-06	3.66E-04	1.11E-05
rob_apply_transform	1.20E-06	2.34E-04	1.09E-06
rob_inv_transform	7.08E-06	3.79E-04	1.15E-05
rob_quat_add	9.20E-07	3.33E-05	7.70E-07
rob_quat_sub	9.10E-07	2.88E-05	7.70E-07
rob_quat_scale	9.50E-07	3.33E-05	6.30E-07
rob_quat_mul	1.28E-06	1.93E-04	1.20E-06
rob_quat_conj	8.20E-07	9.60E-06	6.00E-07
rob_quat_norm	5.21E-06	9.96E-05	9.35E-06
rob_quat_inv	6.32E-06	2.46E-04	1.13E-05
rob_rot2tr	1.91E-06	2.49E-05	2.29E-06
rob_tr2rot	1.81E-06	2.54E-05	2.30E-06
rob_rpy2rot	2.43E-05	1.96E-03	3.31E-05
rob_rpy2tr	2.51E-05	2.00E-03	3.53E-05
rob_update_transform	2.67E-06	6.76E-05	3.81E-06
rob_eul2rot	2.42E-05	1.95E-03	3.31E-05
rob_eul2tr	2.51E-05	2.01E-03	3.53E-05
rob_rot2quat	2.66E-05	3.91E-04	4.71E-05
rob_tr2quat	2.68E-05	3.91E-04	4.72E-05
rob_rpy2quat	5.42E-05	2.43E-03	8.75E-05
rob_eul2quat	5.49E-05	2.46E-03	8.87E-05
rob_tr2rpy	5.88E-06	4.63E-04	3.98E-06
rob_tr2eul	6.68E-06	1.36E-04	5.37E-06
rob_quat2rot	3.99E-06	2.45E-04	4.33E-06
rob_quat2tr	4.90E-06	2.68E-04	7.17E-06
rob_quat2rpy	8.19E-06	2.68E-04	7.17E-06
rob_quat2eul	7.78E-06	2.68E-04	7.17E-06

Nota. Elaboración propia.

MATLAB fue la plataforma que alcanzó los tiempos de operación más elevados, siendo estos en el orden de 1×10^{-1} segundos en las primeras mediciones con la función `rpy2r`. Sin embargo, en esa gráfica se observa que el tiempo en MATLAB tuvo irregularidades, así como las demás funciones en MATLAB durante las primeras 30 mediciones de 100, hasta mantenerse en el orden de 1×10^{-3} en el resto, por lo que se utilizó este órden (milisegundos) para comparar el rendimiento de las demás funciones. Bajo esta métrica, se obtuvieron tiempos de operación aceptables con todas las funciones en las demás plataformas, dado que los tiempos más elevados fueron en escala de milisegundos, correspondiente a las funciones `rob_rpy2rot`, `rob_rpy2tr`, `rob_eul2rot`, `rob_eul2tr`, `rob_rpy2quat` y `rob_eul2quat` en el Arduino MEGA. Esto se puede observar en las Figuras 54-59. Mientras que el resto de

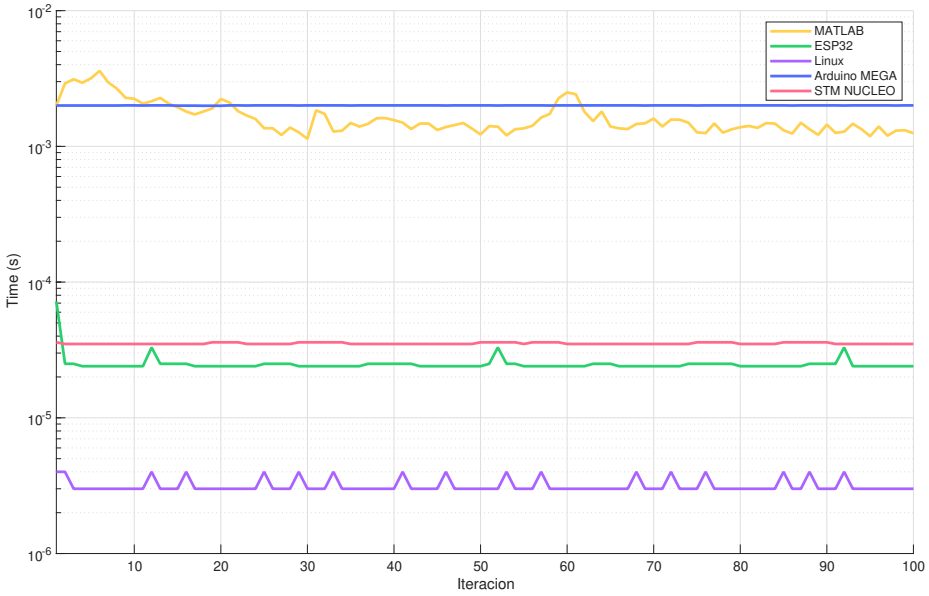
funciones en el Arduino MEGA, ESP32, Linux-amd64 y la STM NUCLEO se mantuvieron por debajo de esto. Aparte de eso, en todas las operaciones con Robotat Robotics se obtuvo una exactitud numérica de cinco decimales o más en todas las pruebas realizadas, validando así el funcionamiento de todas las rutinas e indicando que son robustas.

Figura 54. Evaluación del tiempo de operación de rob_rpy2rot



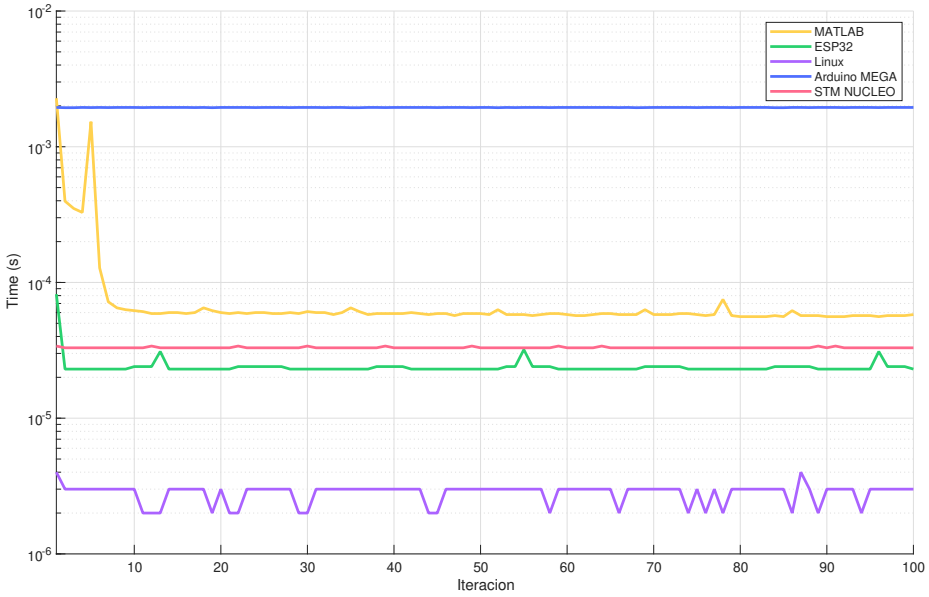
Nota. Tiempo de operación contra el número de iteración. Elaboración propia.

Figura 55. Evaluación del tiempo de operación de rob_rpy2tr



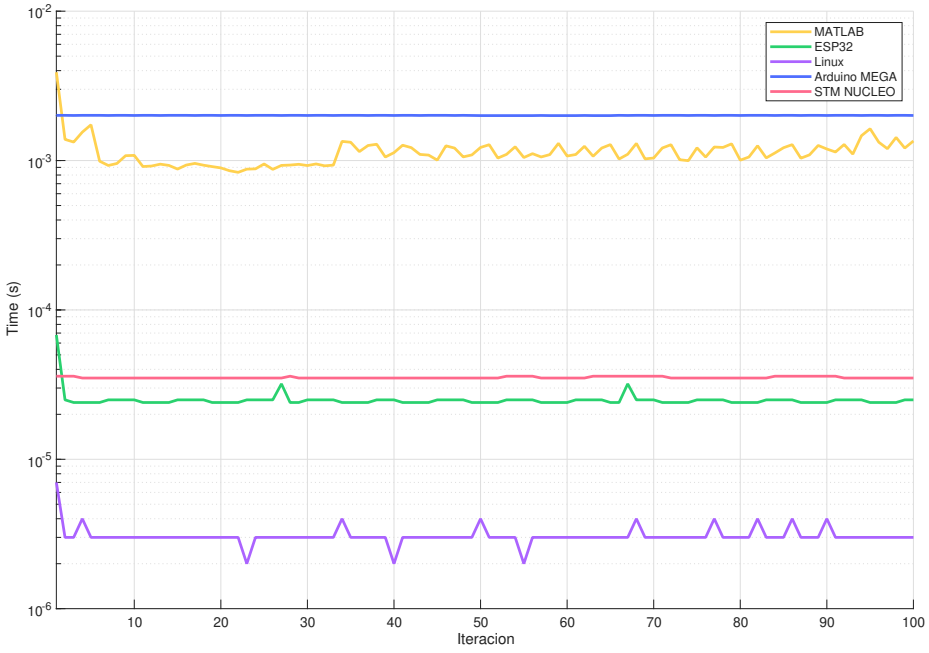
Nota. Tiempo de operación contra el número de iteración. Elaboración propia.

Figura 56. Evaluación del tiempo de operación de rob_eul2rot



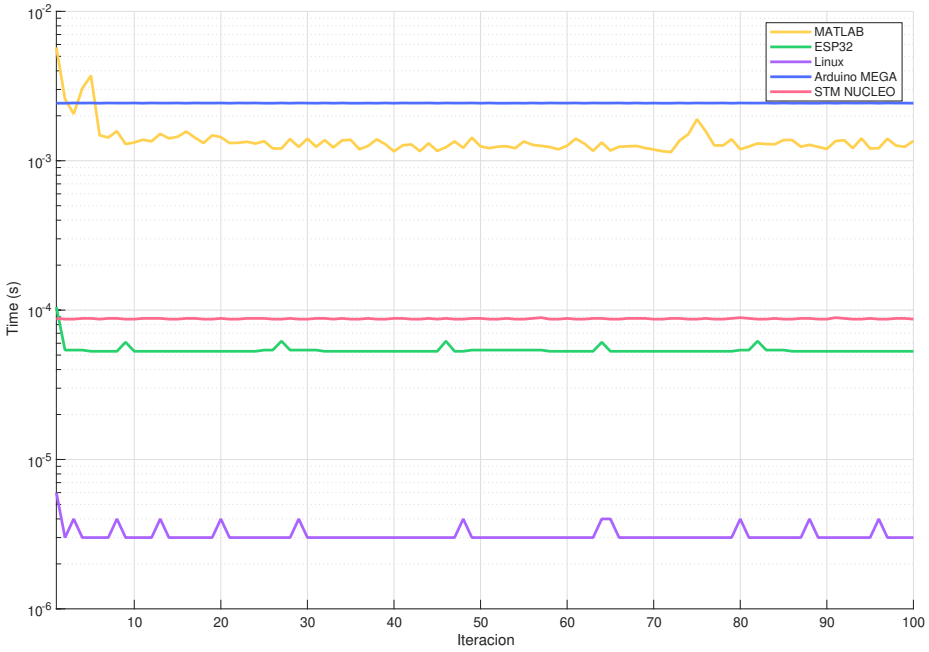
Nota. Tiempo de operación contra el número de iteración. Elaboración propia.

Figura 57. Evaluación del tiempo de operación de rob_eul2tr



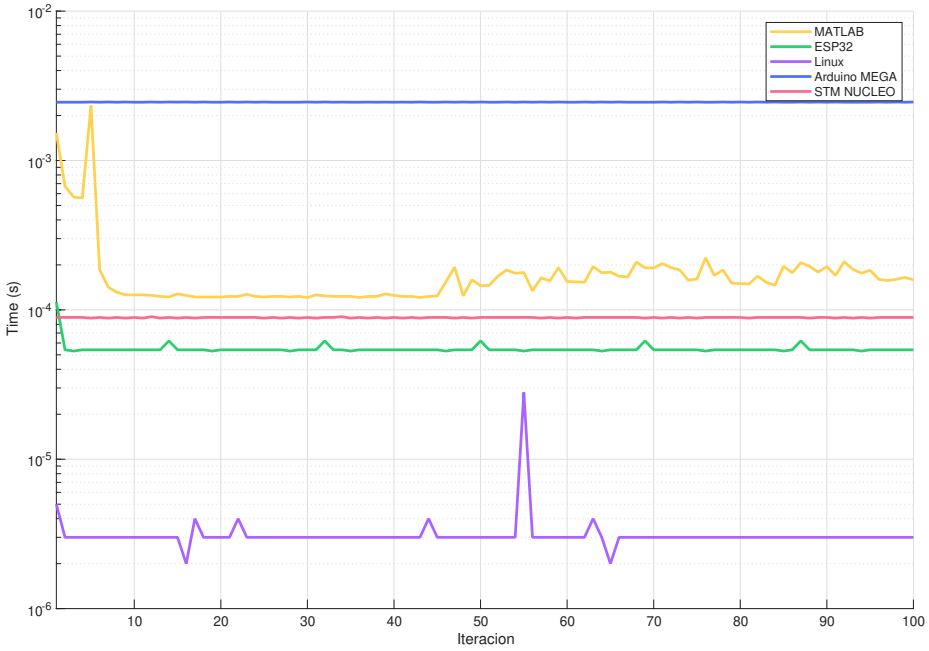
Nota. Tiempo de operación contra el número de iteración. Elaboración propia.

Figura 58. Evaluación del tiempo de operación de rob_rpy2quat



Nota. Tiempo de operación contra el número de iteración. Elaboración propia.

Figura 59. Evaluación del tiempo de operación de rob_eul2quat



Nota. Tiempo de operación contra el número de iteración. Elaboración propia.

Finalmente, en los cuadros 21 y 22 se presenta la evaluación del uso de memoria de Robotat Robotics en los tres dispositivos embebidos. A diferencia de las librerías anteriores, que se midió la memoria empleada por función evaluada, para esta librería se incluyeron todas las evaluaciones de las funciones y sus respectivos datos de entrada (siempre excluyendo los resultados numéricos de MATLAB). El motivo de esto es que el consumo de memoria fue bajo para cada función, por lo que en conjunto la evaluación de todas estas alcanzó porcentajes semejantes a los obtenidos con las funciones de `matf32`. Para la memoria RAM, el Arduino MEGA tuvo el mayor consumo (29.3 %), seguido del ESP32 (7.3 %) y la STM NUCLEO (2.6 %). En cuanto a la memoria FLASH, el ESP32 empleó la mayor cantidad (21.6 %), igual que en las demás librerías, mientras que los otros dispositivos tuvieron un menor porcentaje. A partir de esto, se consideró que Robotat Robotics tiene una alta eficiencia en cuanto al uso de memoria, dando lugar a que se puedan desarrollar aplicaciones complejas en las tres librerías.

Cuadro 21. Porcentaje de uso de memoria RAM en los dispositivos embebidos para evaluar todas las funciones de Robotat Robotics

Arduino MEGA	**ESP32**	**STM NUCLEO**
RAM	**RAM**	**RAM**
29.3 %	7.3 %	2.6 %

Nota. Elaboración propia.

Cuadro 22. Porcentaje de uso de memoria FLASH en los dispositivos embebidos para evaluar todas las funciones de Robotat Robotics

Arduino MEGA	**ESP32**	**STM NUCLEO**
FLASH	**FLASH**	**FLASH**
7.3 %	21.6 %	6 %

Nota. Elaboración propia.

CAPÍTULO 9

Conclusiones

- Se validó el funcionamiento de las rutinas de la librería `matf32`, `linsolve` y `quadprog` de Robotat Linalg, comprobando que es posible obtener en todas estas una exactitud numérica de cinco decimales o más, con respecto de MATLAB, e identificando factores que pueden generar pérdida de exactitud como las características de las matrices utilizadas y posible acumulación de error en los algoritmos que involucran una mayor cantidad de operaciones.
- Se completó la implementación de los métodos de Cholesky y QR en el solucionador lineal, y se añadió la Descomposición de Valores Singulares (SVD) empleando rotaciones de Jacobi para operar casos mal acondicionados según lo requiera el usuario. También se añadió la opción de utilizar QR para solucionar sistemas con matrices cuadradas.
- Se completó la implementación del solucionador cuadrático, validando la utilización de `linsolve` para resolver directamente el sistema KKT asociado a los problemas cuadráticos con restricciones por igualdad, siendo posible utilizar las factorizaciones: LU, SVD, QR. Además, se añadieron y validaron dos métodos recomendados para solución de sistemas KKT como alternativa a la solución directa con `linsolve`. Finalmente, se modificó y validó el método de conjunto activo para solucionar problemas cuadráticos con restricciones de desigualdad y con restricciones de igualdad y desigualdad.
- Se completó la implementación de la librería Robotat Control empleando Robotat Linalg, validando las funciones para controladores PID, sistemas LTI, sistemas no lineales, el filtro de Kalman y, por último, añadiendo y validando un modelo de control predictivo de disparo (*shooting*) en tres escenarios distintos: sin restricciones, con restricciones en el vector de entrada y aplicando restricciones al vector de estados por medio de la entrada.
- Se desarrolló y se validó el funcionamiento de la librería Robotat Robotics empleando la librería `matf32` de Robotat Linalg para operaciones de cálculos de pose, obteniendo tiempos de operación en escala de milisegundos o menos y una exactitud numérica de

cinco o más decimales comparada con la librería *Robotics Toolbox* de Peter Corke para MATLAB.

- Se validó que el ESP32 y la STM NUCLEO F446RE son capaces de ejecutar todas las funciones en Robotat Linalg, Robotat Control y Robotat Robotics, empleando menos del 15 % de la memoria RAM para validar cada función (en ambos dispositivos). Por su parte, el Arduino MEGA es capaz de ejecutar la mayoría de funciones, exceptuando `quadprog_qp_nullspace`, `quadprog_qp_ldlt` y el control de modelo predictivo.

CAPÍTULO 10

Recomendaciones

- Se recomienda emplear estos en las plataformas robóticas del laboratorio Robotat en la UVG. Por ejemplo, en los manipuladores seriales y robots con ruedas, para evaluar el rendimiento de ambas librerías en aplicaciones reales e identificar posibles limitaciones.
- Se recomienda el uso del ESP32 y la STM NUCLEO F446RE dada su mayor capacidad de memoria y velocidad de procesamiento comparados con el Arduino MEGA. Además, en caso se decida trabajar con el Arduino MEGA, se recomienda limitar su uso a una menor cantidad de operaciones y matrices.
- Se recomienda continuar expandiendo las capacidades de la librería Robotat Control, implementando un regulador lineal-cuadrático (LQR), ya que también es utilizado en el curso de Sistemas de Control 2, así como implementando otros tipos de control predictivo.
- Se recomienda implementar funciones para operar matrices de matrices en la librería `matf32`, a partir de generalizar la lógica empleada para las funciones que se implementaron en la generación de las matrices del MPC.
- Se recomienda evaluar la posible implementación de la librería Robotat Linalg para aplicaciones de *Machine Learning*.

CAPÍTULO 11

Referencias

[1] L. V. S. Boyd, *Convex Optimization*. Cambridge University Press, 2004, ISBN: 9781107394001. dirección: https://books.google.com.gt/books/about/Convex_Optimization.html?id=IUZdAAAAQBAJ&redir_esc=y.

[2] A. N. J. Martins, *Engineering Design Optimization*. Cambridge University Press, 2021, ISBN: 9781108988612. dirección: https://books.google.com.gt/books/about/Engineering_Design_Optimization.html?id=FRdSEAAAQBAJ&redir_esc=y.

[3] A. G. P. y T. Ding, «qpSWIFT: A Real-Time Sparse Quadratic Program Solver for Robotic Applications,» *IEEE*, vol. 4, n.º 4, págs. 3355-3362, 2019. DOI: 10.1109/LRA.2019.2926664. dirección: https://ieeexplore.ieee.org/document/8754693.

[4] J. M. y S. Boyd, «CVXGEN: a code generator for embedded convex optimization,» *Optimization and Engineering*, vol. 13, págs. 1-27, 2012. DOI: https://doi.org/10.1007/s11081-011-9176-9. dirección: https://link.springer.com/article/10.1007/s11081-011-9176-9.

[5] C. S. y J. Brembeck, «A QP Solver Implementation for Embedded Systems Applied to Control Allocation,» *computation*, vol. 8, n.º 4, pág. 88, 2020. DOI: https://doi.org/10.3390/computation8040088. dirección: https://www.mdpi.com/2079-3197/8/4/88.

[6] G. B. y B. Stellato, «Embedded code generation using the OSQP solver,» *IEEE*, vol. 56th Annual Conference on Decision and Control (CDC), págs. 1906-1911, 2017. DOI: 10.1109/CDC.2017.8263928. dirección: https://ieeexplore.ieee.org/abstract/document/8263928.

[7] B. S. y G. Banjac, «OSQP: An Operator Splitting Solver for Quadratic Programs,» *IEEE*, vol. UKACC 12th International Conference on Control (CONTROL), 339-339, 2018. DOI: 10.1109/CONTROL.2018.8516834. dirección: https://ieeexplore.ieee.org/document/8516834.

[8] D. A. y A. Bemporad, «A Dual Active-Set Solver for Embedded Quadratic Programming Using Recursive LDLT Updates,» *IEEE*, vol. 67, n.º 8, págs. 4362-4369, 2022. DOI: 10.1109/TAC.2022.3176430. dirección: https://ieeexplore.ieee.org/document/9779534.

[9] J. N. Kutz, *Data Driven Modelling & Scientific Computation: Methods for Complex Systems Big Data*. OUP Oxford, 2013, ISBN: 9780191635878. dirección: https://books.google.com.gt/books?id=L0toAgAAQBAJ.

[10] D. B. Lloyd Trefethen, *Numerical Linear Algebra*. SIAM, 1997, ISBN: 0898719577. dirección: https://books.google.com.gt/books/about/Numerical_Linear_Algebra.html?id=JaPtxOytY7kC&redir_esc=y.

[11] S. W. J. Nocedal, *Numerical Optimization*. Springer Science & Business Media, 2000, ISBN: 9781107394001. dirección: https://books.google.com.gt/books/about/Numerical_Optimization.html?id=w1kJYpOkPykC&redir_esc=y.

[12] C. F. V. L. Gene H. Golub, *Matrix Computations*. JHU Press, 2013, ISBN: 1421407949. dirección: https://books.google.com.gt/books/about/Matrix_Computations.html?id=X5YfsuCWpxMC&redir_esc=y.

[13] J. Demmel, *Applied Numerical Linear Algebra*. SIAM, 1997, ISBN: 1611971446. dirección: https://books.google.com.gt/books/about/Applied_Numerical_Linear_Algebra.html?id=P3bPAgAAQBAJ&redir_esc=y.

[14] J. D. P. Gene F. Franklin, *Feedback Control of Dynamic Systems*. Pearson, 2014, ISBN: 1292068906. dirección: https://books.google.com.gt/books/about/Feedback_Control_of_Dynamic_Systems.html?id=yO2hoAEACAAJ&redir_esc=y.

[15] J. D. P. Gene F. Franklin, *Digital Control of Dynamic Systems*. Addison-Wesley, 1998, ISBN: 0201820544. dirección: https://books.google.com.gt/books/about/Digital_control_of_dynamic_systems.html?id=z6UeAQAAIAAJ&redir_esc=y.

[16] J. F. Epperson, *An Introduction to Numerical Methods and Analysis*. John Wiley Sons, 2013, ISBN: 1118626230. dirección: https://books.google.com.gt/books/about/An_Introduction_to_Numerical_Methods_and.html?id=O1U5W5hzvCoC&redir_esc=y.

[17] M. C. Basil Kouvaritakis, *Model Predictive Control: Classical, Robust and Stochastic*. Springer International Publishing, 2019, ISBN: 3319796895. dirección: https://books.google.com.gt/books/about/Model_Predictive_Control.html?id=w6IcwAEACAAJ&redir_esc=y.

[18] P. Corke, *Robotics, Vision and Control: Fundamental Algorithms in Matlab*. Springer Science & Business Media, 2011, ISBN: 9783642201431. dirección: https://books.google.com.gt/books/about/%5C%5CRobotics_Vision_and_Control.html?id=hdkytqtBcyQC&redir_esc=y.

[19] J. J. Craig, *Introduction to Robotics: Mechanics and Control*. Pearson/Prentice Hall, 2005, ISBN: 0201543613. dirección: https://books.google.com.gt/books/about/Introduction_to_Robotics.html?id=MqMeAQAAIAAJ&redir_esc=y.

[20] Arduino, *Arduino MEGA 2560 Rev3*. 2025. dirección: https://docs.arduino.cc/hardware/mega-2560/.

[21] ESPRESSIF, *ESP32-WROOM-32E ESP32-WROOM-32UE Datasheet Version 2.0*. 2025. dirección: https://documentation.espressif.com/esp32-wroom-32e_esp32-wroom-32ue_datasheet_en.html.

[22] STMicroelectronics, *STM32F446xC/E*. 2021. dirección: https://www.st.com/en/microcontrollers-microprocessors/stm32f446re.html.

[23] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*. Society for Industrial y Applied Mathematics, 1996, ISBN: 0898713552. dirección: https://books.google.com.gt/books/about/Accuracy_and_Stability_of_Numerical_Algo.html?id=BdzhwAEACAAJ&redir_esc=y.

CAPÍTULO 12

Anexos

En este capítulo de anexos se incluyen primero los enlaces al repositorio en Github que contiene el código para las librerías Robotat Linalg, Robotat Control y Robotat Robotics. Después de eso, se incluyen los cuadros de resultados de tiempo generados para durante las evaluaciones de las distintas librerías y que se emplearon para crear las gráficas de cada función evaluada. Asímismo, se incluyen las gráficas de las funciones que se evaluaron, pero que no se consideró prioridad incluirlas en los capítulos, dado que sus resultados se podían explicar de forma resumida, por lo que se colocaron en los anexos para que el lector pueda revisarlas en detalle si lo desea.

12.1. Repositorio del proyecto

Repositorio: https://github.com/andreabp19/robotat-linalg

12.2. Funciones evaluadas de Robotat Linalg organizadas por el tiempo de operación máximo medido en cada plataforma

Cuadro 23. Funciones y operaciones evaluadas en MATLAB, en orden descendente según el tiempo de operación máximo medido

#ID	Función	Tiempo Máximo (s)
1	quadprog (active-set) to compare with quadprog_qp_nullspace	4.23E-03
2	quadprog (active-set) Equality and Inequality Constrained	5.75E-04
3	one_sided_jacobi_svd	4.10E-04
4	\or linsolve (SVD)	5.98E-05
5	quadprog (active-set) Equality Constrained	4.72E-05
6	\or linsolve (QR)	3.30E-05
7	pinv (con SVD)	3.26E-05
8	A+A+A	1.46E-05
9	pinv	1.44E-05
10	A-A-A	1.33E-05
11	\or linsolve (Forward Substitution)	1.23E-05
12	A+A	1.20E-05
13	A*A*A	1.14E-05
14	\or linsolve (LU)	1.09E-05
15	b*A	9.60E-06
16	A-A	8.69E-06
17	qr	7.98E-06
18	lu	7.69E-06
19	\or linsolve (Cholesky)	7.52E-06
20	inv	7.43E-06
21	A*c	6.47E-06
22	chol	6.41E-06
23	\or linsolve (Backward Substitution)	6.23E-06
24	A*A	6.18E-06
25	A’	6.06E-06
26	^	5.90E-06
27	A*b	5.42E-06
28	dot	5.39E-06
29	rowvec*colvec	4.82E-06

Nota. Elaboración propia.

Cuadro 24. Funciones de Robotat Linalg evaluadas en el ESP32, en orden descendente según el tiempo de operación máximo medido

#ID	Función	Tiempo Máximo (s)
1	matf32_jacobi_svd	8.65E-02
2	quadprog_qp_linsolve (SVD)	4.22E-02
3	quadprog_qp_ldlt	3.61E-02
4	quadprog_sqp	9.96E-04
5	quadprog_qp_nullspace	6.10E-04
6	quadprog_qp_linsolve (QR)	5.63E-04
7	matf32_qr	4.79E-04
8	linsolve (QR)	3.94E-04
9	matf32_inv	1.91E-04
10	matf32_arr_mul	1.68E-04
11	linsolve (Cholesky)	1.67E-04
12	quadprog_qp_linsolve (LU)	1.18E-04
13	linsolve (LU)	1.14E-04
14	linsolve (SVD)	8.94E-05
15	matf32_mul	8.29E-05
16	matf32_lu	8.06E-05
17	matf32_cholesky	5.84E-05
18	matf32_pinv (con SVD)	4.16E-05
19	matf32_arr_add	2.72E-05
20	matf32_arr_sub	1.91E-05
21	linsolve (Forward Substitution)	1.82E-05
22	linsolve (Backward Substitution)	1.77E-05
23	matf32_vecposmul	9.25E-06
24	matf32_trans	9.08E-06
25	matf32_scale	8.80E-06
26	matf32_vecpremul	8.76E-06
27	matf32_add	8.72E-06
28	matf32_sub	8.70E-06
29	matf32_vecmul_col_row	7.06E-06
30	matf32_dot	1.49E-06
31	matf32_pinv	3.80E-07
32	matf32_exp	2.50E-07

Nota. Elaboración propia.

Cuadro 25. Funciones de Robotat Linalg evaluadas en el Arduino MEGA, en orden descendente según el tiempo de operación máximo medido

#ID	Función	Tiempo Máximo (s)
1	matf32_pinv (con SVD)	6.05E+00
2	quadprog_qp_linsolve (SVD)	5.58E+00
3	linsolve (SVD)	2.04E+00
4	quadprog_sqp	1.08E-01
5	matf32_pinv	7.12E-02
6	matf32_exp	6.85E-02
7	quadprog_qp_linsolve (QR)	6.67E-02
8	matf32_jacobi_svd	6.13E-02
9	matf32_qr	6.12E-02
10	linsolve (LU)	4.77E-02
11	linsolve (QR)	4.59E-02
12	matf32_arr_mul	4.19E-02
13	matf32_inv	2.88E-02
14	matf32_mul	2.15E-02
15	linsolve (Cholesky)	2.13E-02
16	quadprog_qp_linsolve (LU)	1.48E-02
17	matf32_lu	1.04E-02
18	matf32_cholesky	8.10E-03
19	matf32_arr_add	3.44E-03
20	linsolve (Backward Substitution)	2.34E-03
21	linsolve (Forward Substitution)	2.26E-03
22	matf32_arr_sub	1.97E-03
23	matf32_vecpremul	1.89E-03
24	matf32_vecposmul	1.84E-03
25	matf32_vecmul_col_row	1.27E-03
26	matf32_scale	1.20E-03
27	matf32_add	8.59E-04
28	matf32_sub	7.90E-04
29	matf32_trans	4.41E-04
30	matf32_dot	4.00E-08

Nota. Elaboración propia.

Cuadro 26. Funciones de Robotat Linalg evaluadas en el STM NUCLEO F446RE, en orden descendente según el tiempo de operación máximo medido

#ID	Función	Tiempo Máximo (s)
1	quadprog_qp_linsolve (SVD)	8.41E-02
2	matf32_pinv (con SVD)	8.09E-02
3	quadprog_qp_ldlt	5.58E-02
4	quadprog_sqp	4.55E-02
5	quadprog_qp_linsolve (QR)	1.00E-03
6	quadprog_qp_nullspace	9.77E-04
7	matf32_qr	8.73E-04
8	matf32_jacobi_svd	8.07E-04
9	linsolve (QR)	6.96E-04
10	matf32_pinv	4.07E-04
11	linsolve (LU)	3.97E-04
12	matf32_exp	3.02E-04
13	linsolve (Cholesky)	2.19E-04
14	matf32_inv	2.09E-04
15	quadprog_qp_linsolve (LU)	1.91E-04
16	matf32_arr_mul	1.77E-04
17	matf32_lu	1.10E-04
18	matf32_mul	7.49E-05
19	matf32_cholesky	5.90E-05
20	matf32_arr_add	5.18E-05
21	matf32_arr_sub	4.26E-05
22	matf32_trans	1.92E-05
23	linsolve (Backward Substitution)	1.47E-05
24	matf32_vecposmul	1.45E-05
25	linsolve (Forward Substitution)	1.45E-05
26	matf32_vecpremul	1.44E-05
27	linsolve (SVD)	1.12E-05
28	matf32_vecmul_col_row	9.10E-06
29	matf32_add	8.15E-06
30	matf32_sub	7.59E-06
31	matf32_scale	6.47E-06
32	matf32_dot	9.90E-07

Nota. Elaboración propia.

Cuadro 27. Funciones de Robotat Linalg evaluadas en Linux-amd64, en orden descendente según el tiempo de operación máximo medido

#ID	Función	Tiempo Máximo (s)
1	matf32_jacobi_svd	3.41E-03
2	quadprog_qp_linsolve (SVD)	1.15E-03
3	quadprog_qp_ldlt	1.01E-03
4	matf32_pinv (con SVD)	9.68E-04
5	linsolve (QR)	4.90E-05
6	matf32_exp	4.05E-05
7	quadprog_qp_linsolve (QR)	2.72E-05
8	quadprog_qp_nullspace	2.60E-05
9	quadprog_sqp	2.23E-05
10	matf32_qr	1.93E-05
11	linsolve (Cholesky)	1.78E-05
12	matf32_pinv	1.20E-05
13	linsolve (SVD)	1.12E-05
14	linsolve (LU)	9.61E-06
15	matf32_arr_mul	7.39E-06
16	matf32_inv	7.29E-06
17	quadprog_qp_linsolve (LU)	6.91E-06
18	matf32_lu	4.59E-06
19	matf32_mul	3.85E-06
20	linsolve (Backward Substitution)	3.13E-06
21	linsolve (Forward Substitution)	2.89E-06
22	matf32_cholesky	1.87E-06
23	matf32_arr_add	1.49E-06
24	matf32_add	1.17E-06
25	matf32_sub	1.15E-06
26	matf32_trans	1.14E-06
27	matf32_scale	1.13E-06
28	matf32_vecposmul	1.10E-06
29	matf32_vecpremul	1.10E-06
30	matf32_dot	1.09E-06
31	matf32_vecmul_col_row	1.05E-06
32	matf32_arr_sub	8.80E-07

Nota. Elaboración propia.

12.3. Evaluaciones de matf32: tiempos de operación promedio por tamaño de matriz

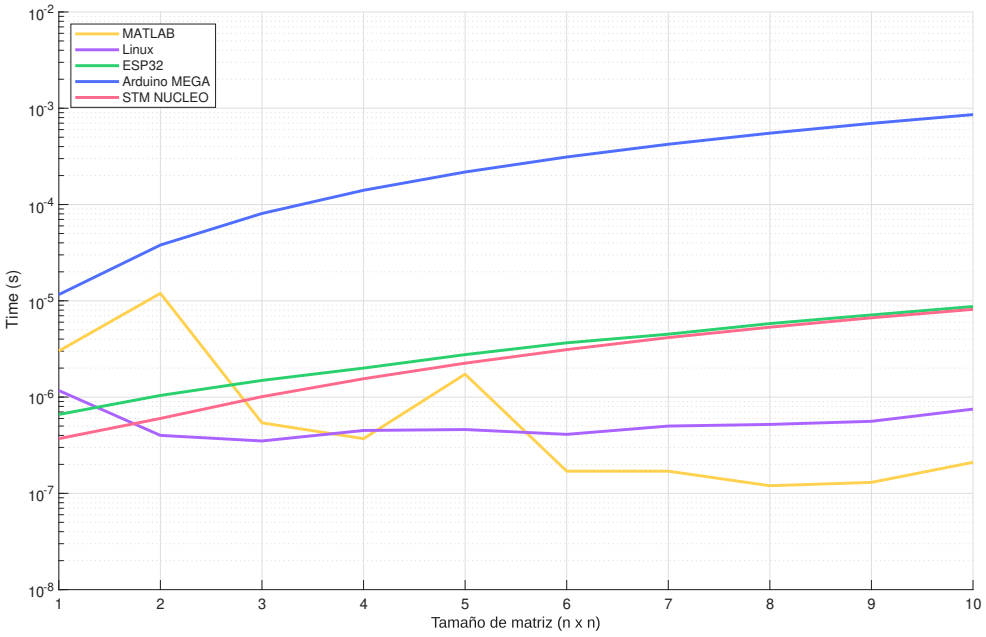
`matf32_add`:

Cuadro 28. Tiempo promedio de operación de `matf32_add` para matrices (n x n)

n	Tiempo (s)				
	MATLAB	**ESP32**	**Linux-amd64**	**Arduino MEGA**	**STM NUCLEO**
1	3.01E-06	6.60E-07	1.17E-06	1.16E-05	3.70E-07
2	1.20E-05	1.04E-06	4.00E-07	3.80E-05	6.00E-07
3	5.40E-07	1.49E-06	3.50E-07	8.07E-05	1.01E-06
4	3.70E-07	2.00E-06	4.50E-07	1.41E-04	1.55E-06
5	1.73E-06	2.76E-06	4.60E-07	2.18E-04	2.25E-06
6	1.70E-07	3.66E-06	4.10E-07	3.12E-04	3.11E-06
7	1.70E-07	4.50E-06	5.00E-07	4.23E-04	4.15E-06
8	1.20E-07	5.80E-06	5.20E-07	5.51E-04	5.32E-06
9	1.30E-07	7.13E-06	5.60E-07	6.96E-04	6.66E-06
10	2.10E-07	8.72E-06	7.50E-07	8.59E-04	8.15E-06

Nota. Elaboración propia.

Figura 60. Evaluación del tiempo de operación de `matf32_add`



Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

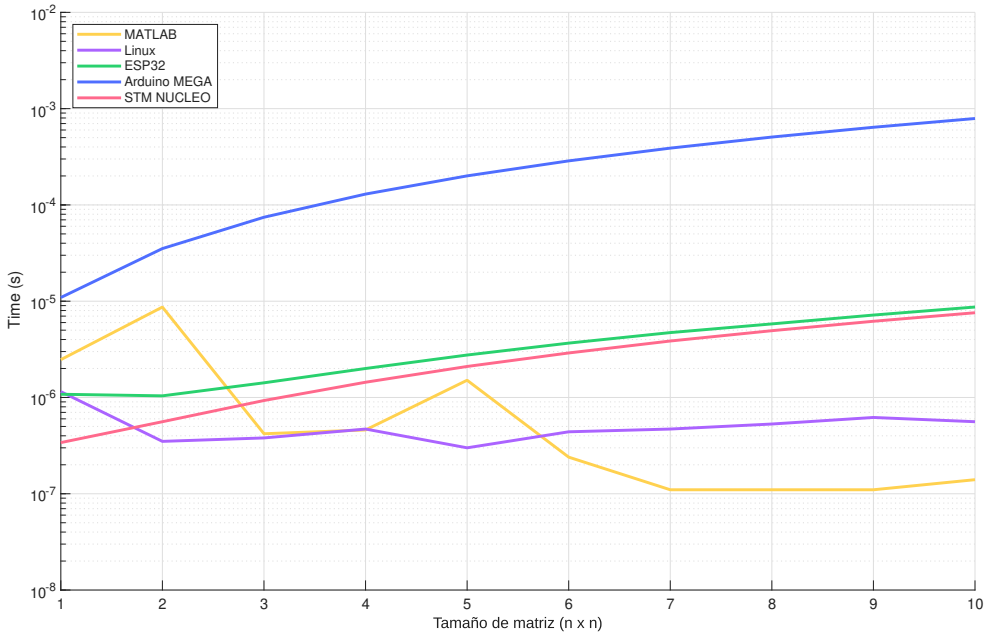
`matf32_sub`:

Cuadro 29. Tiempo promedio de operación de `matf32_sub` para matrices (n x n)

n	Tiempo (s)				
	MATLAB	**ESP32**	**Linux-amd64**	**Arduino MEGA**	**STM NUCLEO**
1	2.47E-06	1.08E-06	1.15E-06	1.09E-05	3.40E-07
2	8.69E-06	1.04E-06	3.50E-07	3.52E-05	5.60E-07
3	4.20E-07	1.42E-06	3.80E-07	7.45E-05	9.30E-07
4	4.60E-07	2.00E-06	4.70E-07	1.30E-04	1.44E-06
5	1.51E-06	2.76E-06	3.00E-07	2.00E-04	2.10E-06
6	2.40E-07	3.67E-06	4.40E-07	2.87E-04	2.90E-06
7	1.10E-07	4.72E-06	4.70E-07	3.89E-04	3.86E-06
8	1.10E-07	5.81E-06	5.30E-07	5.07E-04	4.94E-06
9	1.10E-07	7.18E-06	6.20E-07	6.40E-04	6.19E-06
10	1.40E-07	8.70E-06	5.60E-07	7.90E-04	7.59E-06

Nota. Elaboración propia.

Figura 61. Evaluación del tiempo de operación `matf32_sub`



Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

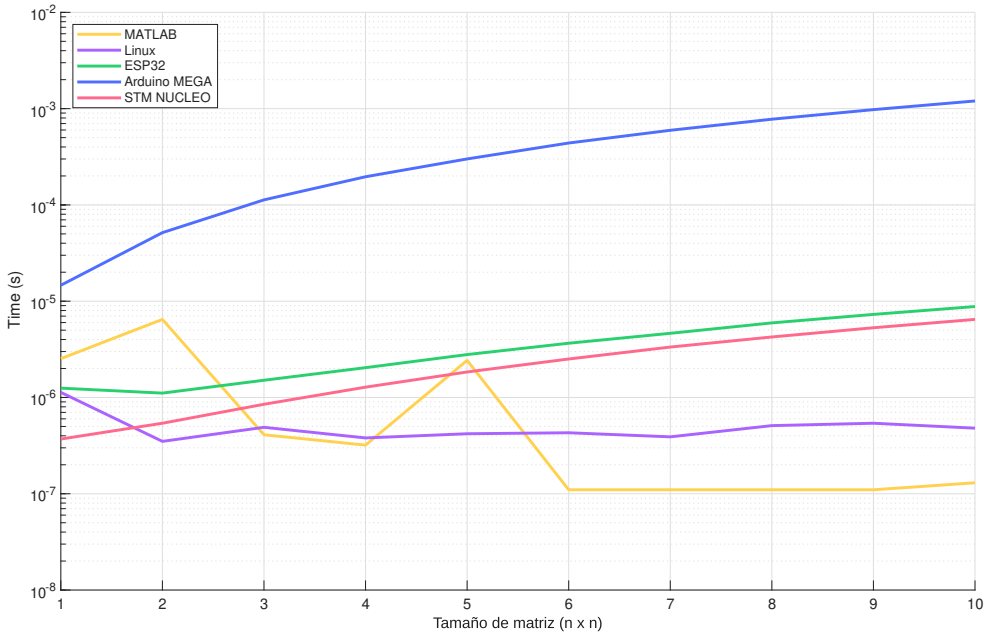
`matf32_scale`:

Cuadro 30. Tiempo promedio de operación de `matf32_scale` para matrices (n x n)

n	Tiempo (s)				
	MATLAB	**ESP32**	**Linux-amd64**	**Arduino MEGA**	**STM NUCLEO**
1	2.53E-06	1.25E-06	1.13E-06	1.46E-05	3.70E-07
2	6.47E-06	1.11E-06	3.50E-07	5.15E-05	5.40E-07
3	4.10E-07	1.51E-06	4.90E-07	1.13E-04	8.50E-07
4	3.20E-07	2.04E-06	3.80E-07	1.96E-04	1.28E-06
5	2.43E-06	2.79E-06	4.20E-07	3.00E-04	1.84E-06
6	1.10E-07	3.66E-06	4.30E-07	4.40E-04	2.51E-06
7	1.10E-07	4.64E-06	3.90E-07	5.96E-04	3.34E-06
8	1.10E-07	5.95E-06	5.10E-07	7.77E-04	4.25E-06
9	1.10E-07	7.29E-06	5.40E-07	9.77E-04	5.30E-06
10	1.30E-07	8.80E-06	4.80E-07	1.20E-03	6.47E-06

Nota. Elaboración propia.

Figura 62. Evaluación del tiempo de operación `matf32_scale`



Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

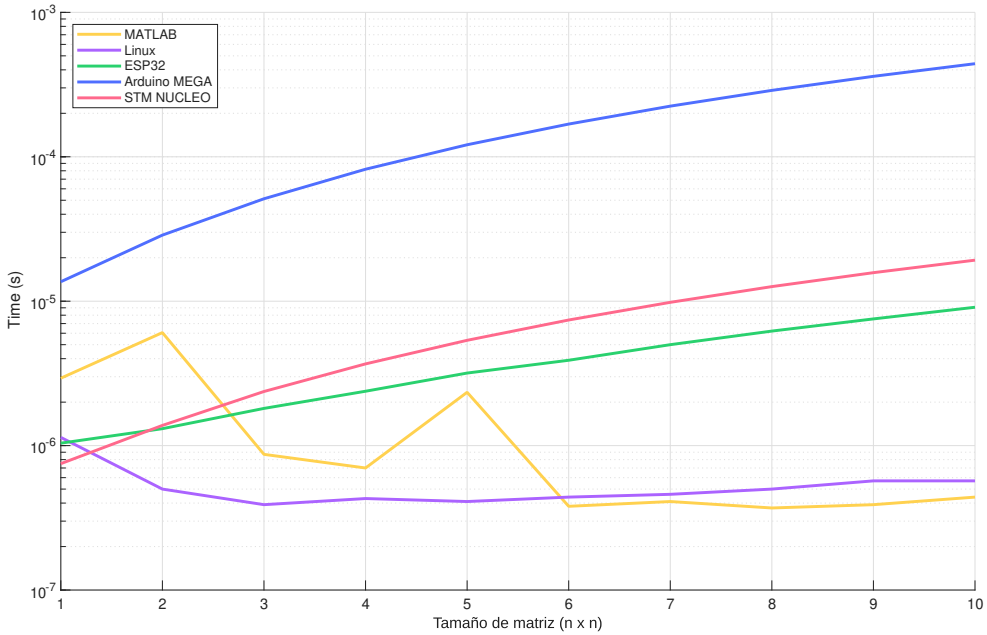
`matf32_trans`:

Cuadro 31. Tiempo promedio de operación de `matf32_trans` para matrices (n x n)

n	Tiempo (s)				
	MATLAB	**ESP32**	**Linux-amd64**	**Arduino MEGA**	**STM NUCLEO**
1	2.93E-06	1.04E-06	1.14E-06	1.36E-05	7.50E-07
2	6.06E-06	1.31E-06	5.00E-07	2.87E-05	1.38E-06
3	8.70E-07	1.81E-06	3.90E-07	5.12E-05	2.37E-06
4	7.00E-07	2.38E-06	4.30E-07	8.20E-05	3.68E-06
5	2.34E-06	3.18E-06	4.10E-07	1.21E-04	5.36E-06
6	3.80E-07	3.90E-06	4.40E-07	1.68E-04	7.41E-06
7	4.10E-07	5.00E-06	4.60E-07	2.24E-04	9.82E-06
8	3.70E-07	6.21E-06	5.00E-07	2.88E-04	1.26E-05
9	3.90E-07	7.54E-06	5.70E-07	3.61E-04	1.58E-05
10	4.40E-07	9.08E-06	5.70E-07	4.41E-04	1.92E-05

Nota. Elaboración propia.

Figura 63. Evaluación del tiempo de operación `matf32_trans`



Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

`matf32_mul`:

Cuadro 32. Tiempo promedio de operación de `matf32_mul` para matrices (n x n)

n	Tiempo (s)				
	MATLAB	**ESP32**	**Linux-amd64**	**Arduino MEGA**	**STM NUCLEO**
1	2.43E-06	1.16E-06	1.19E-06	2.97E-05	6.30E-07
2	6.18E-06	1.82E-06	4.40E-07	1.87E-04	1.39E-06
3	6.70E-07	3.63E-06	5.40E-07	6.18E-04	3.08E-06
4	5.90E-07	6.85E-06	7.50E-07	1.44E-03	6.15E-06
5	2.52E-06	1.21E-05	9.40E-07	2.74E-03	1.09E-05
6	4.70E-07	1.97E-05	1.28E-06	4.75E-03	1.79E-05
7	5.60E-07	3.02E-05	1.81E-06	7.45E-03	2.74E-05
8	5.10E-07	4.39E-05	2.34E-06	1.10E-02	3.99E-05
9	5.60E-07	6.15E-05	3.13E-06	1.57E-02	5.55E-05
10	5.70E-07	8.29E-05	3.85E-06	2.15E-02	7.49E-05

Nota. Elaboración propia.

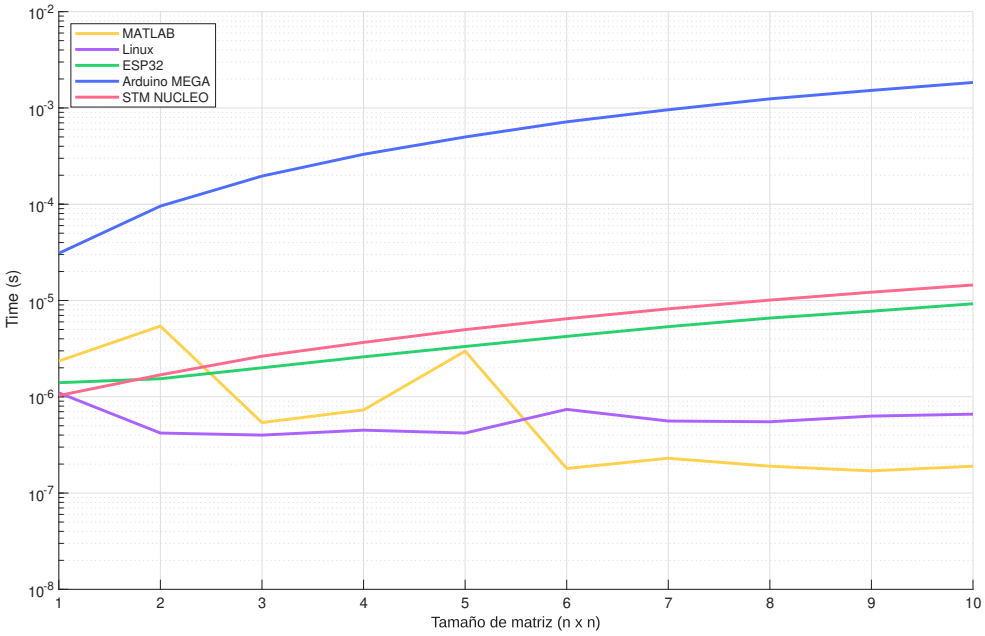
`matf32_vecposmul`:

Cuadro 33. Tiempo promedio de operación de `matf32_vecposmul` para matrices (n x n)

n	Tiempo (s)				
	MATLAB	**ESP32**	**Linux-amd64**	**Arduino MEGA**	**STM NUCLEO**
1	2.35E-06	1.40E-06	1.10E-06	3.09E-05	1.03E-06
2	5.42E-06	1.54E-06	4.20E-07	9.56E-05	1.69E-06
3	5.40E-07	2.00E-06	4.00E-07	1.96E-04	2.64E-06
4	7.30E-07	2.60E-06	4.50E-07	3.30E-04	3.67E-06
5	2.98E-06	3.33E-06	4.20E-07	4.99E-04	4.98E-06
6	1.80E-07	4.24E-06	7.40E-07	7.17E-04	6.47E-06
7	2.30E-07	5.35E-06	5.60E-07	9.58E-04	8.19E-06
8	1.90E-07	6.57E-06	5.50E-07	1.24E-03	1.01E-05
9	1.70E-07	7.74E-06	6.30E-07	1.52E-03	1.22E-05
10	1.90E-07	9.25E-06	6.60E-07	1.84E-03	1.45E-05

Nota. Elaboración propia.

Figura 64. Evaluación del tiempo de operación `matf32_vecposmul`



Nota. Elaboración propia.

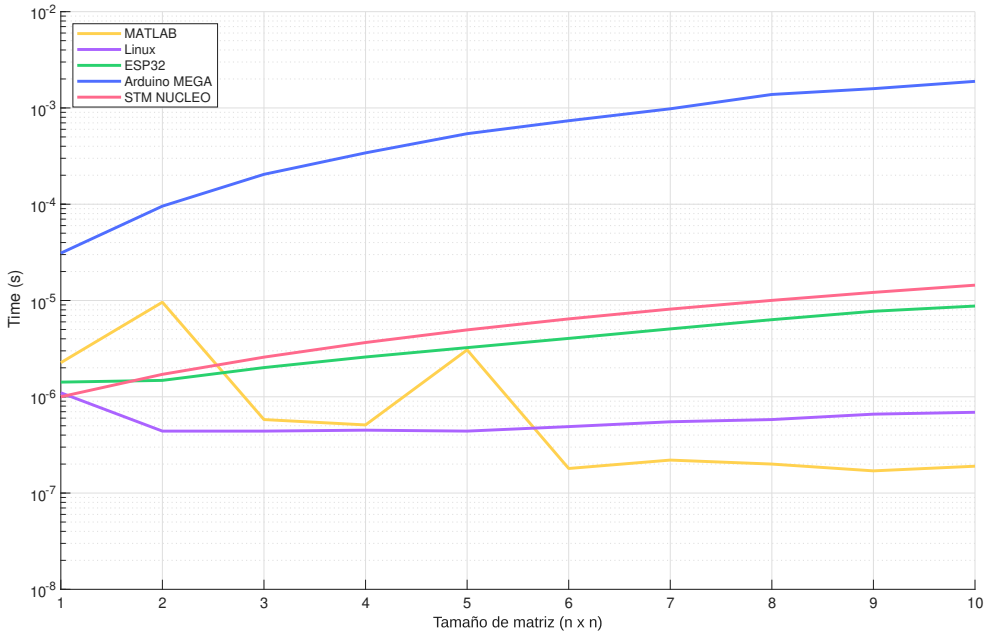
`matf32_vecpremul`:

Cuadro 34. Tiempo promedio de operación de `matf32_vecpremul` para matrices (n x n)

n	Tiempo (s)				
	MATLAB	**ESP32**	**Linux-amd64**	**Arduino MEGA**	**STM NUCLEO**
1	2.26E-06	1.42E-06	1.10E-06	3.10E-05	1.00E-06
2	9.60E-06	1.48E-06	4.40E-07	9.55E-05	1.71E-06
3	5.80E-07	2.01E-06	4.40E-07	2.04E-04	2.58E-06
4	5.10E-07	2.59E-06	4.50E-07	3.41E-04	3.66E-06
5	3.07E-06	3.24E-06	4.40E-07	5.40E-04	4.95E-06
6	1.80E-07	4.04E-06	4.90E-07	7.36E-04	6.44E-06
7	2.20E-07	5.07E-06	5.50E-07	9.80E-04	8.14E-06
8	2.00E-07	6.32E-06	5.80E-07	1.38E-03	1.00E-05
9	1.70E-07	7.73E-06	6.60E-07	1.59E-03	1.21E-05
10	1.90E-07	8.76E-06	6.90E-07	1.89E-03	1.44E-05

Nota. Elaboración propia.

Figura 65. Evaluación del tiempo de operación `matf32_vecpremul`



Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

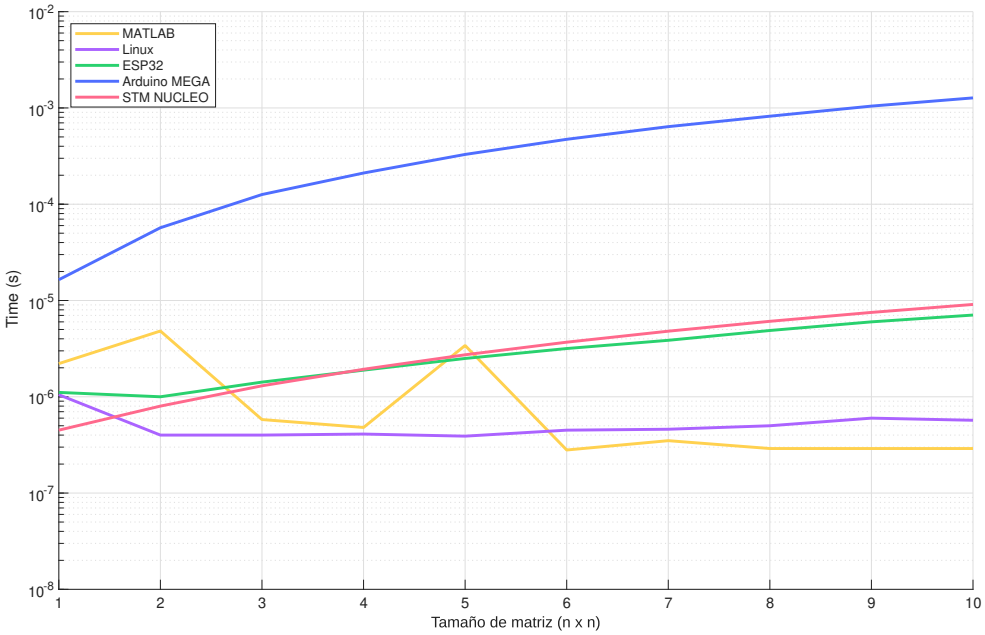
`matf32_vecmul_col_row`:

Cuadro 35. Tiempo promedio de operación de `matf32_vecmul_col_row` para matrices (n x n)

n	Tiempo (s)				
	MATLAB	**ESP32**	**Linux-amd64**	**Arduino MEGA**	**STM NUCLEO**
1	2.21E-06	1.11E-06	1.05E-06	1.64E-05	4.50E-07
2	4.82E-06	1.00E-06	4.00E-07	5.70E-05	8.00E-07
3	5.80E-07	1.42E-06	4.00E-07	1.26E-04	1.30E-06
4	4.80E-07	1.89E-06	4.10E-07	2.11E-04	1.93E-06
5	3.41E-06	2.50E-06	3.90E-07	3.29E-04	2.73E-06
6	2.80E-07	3.17E-06	4.50E-07	4.72E-04	3.69E-06
7	3.50E-07	3.86E-06	4.60E-07	6.40E-04	4.80E-06
8	2.90E-07	4.88E-06	5.00E-07	8.21E-04	6.08E-06
9	2.90E-07	6.00E-06	6.00E-07	1.05E-03	7.52E-06
10	2.90E-07	7.06E-06	5.70E-07	1.27E-03	9.10E-06

Nota. Elaboración propia.

Figura 66. Evaluación del tiempo de operación `matf32_vecmul_col_row`



Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

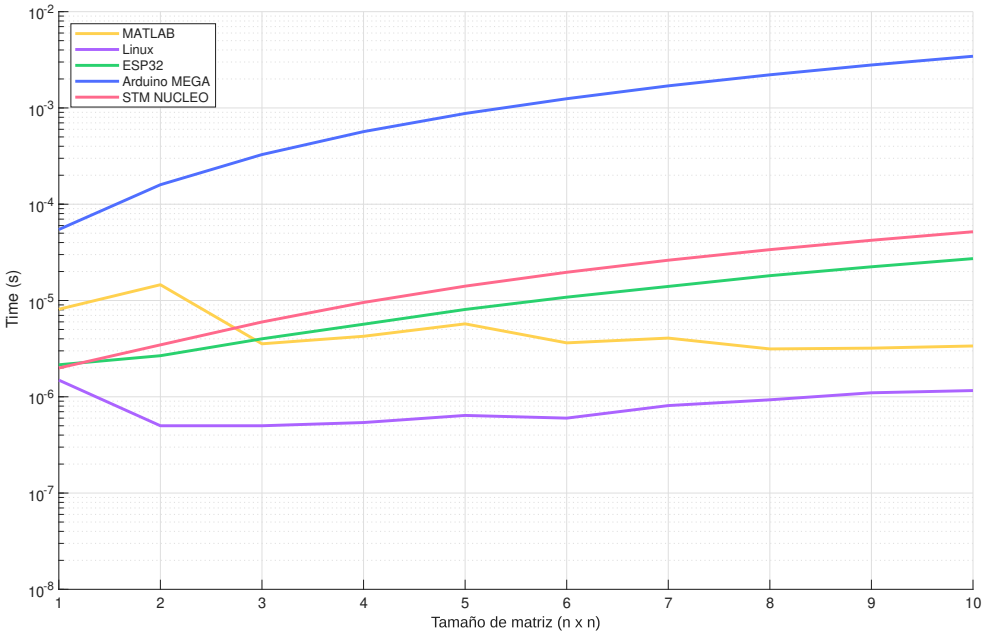
`matf32_arr_add`:

Cuadro 36. Tiempo promedio de operación de `matf32_arr_add` para matrices (n x n)

n	Tiempo (s)				
	MATLAB	**ESP32**	**Linux-amd64**	**Arduino MEGA**	**STM NUCLEO**
1	8.11E-06	2.15E-06	1.49E-06	5.46E-05	1.99E-06
2	1.46E-05	2.67E-06	5.00E-07	1.59E-04	3.46E-06
3	3.55E-06	4.00E-06	5.00E-07	3.28E-04	5.98E-06
4	4.25E-06	5.67E-06	5.40E-07	5.68E-04	9.54E-06
5	5.72E-06	8.07E-06	6.40E-07	8.76E-04	1.41E-05
6	3.63E-06	1.08E-05	6.00E-07	1.25E-03	1.96E-05
7	4.07E-06	1.40E-05	8.10E-07	1.70E-03	2.62E-05
8	3.14E-06	1.81E-05	9.30E-07	2.21E-03	3.37E-05
9	3.20E-06	2.24E-05	1.10E-06	2.79E-03	4.22E-05
10	3.37E-06	2.72E-05	1.16E-06	3.44E-03	5.18E-05

Nota. Elaboración propia.

Figura 67. Evaluación del tiempo de operación `matf32_arr_add`



Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

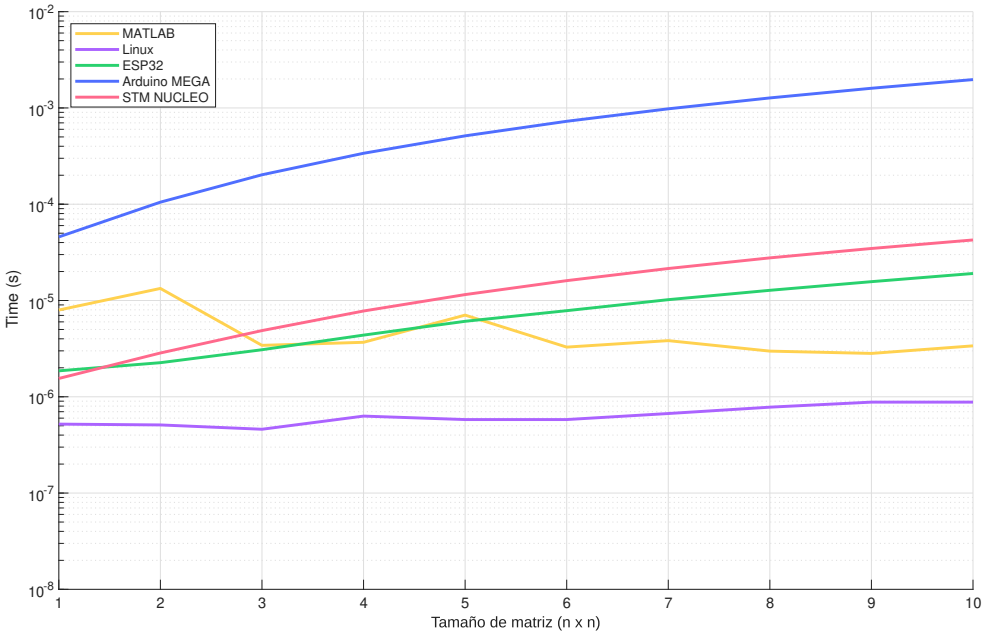
`matf32_arr_sub`:

Cuadro 37. Tiempo promedio de operación de `matf32_arr_sub` para matrices (n x n)

n	Tiempo (s)				
	MATLAB	**ESP32**	**Linux-amd64**	**Arduino MEGA**	**STM NUCLEO**
1	7.96E-06	1.86E-06	5.20E-07	4.58E-05	1.55E-06
2	1.33E-05	2.26E-06	5.10E-07	1.05E-04	2.85E-06
3	3.42E-06	3.08E-06	4.60E-07	2.02E-04	4.87E-06
4	3.68E-06	4.37E-06	6.30E-07	3.38E-04	7.78E-06
5	7.05E-06	6.08E-06	5.80E-07	5.13E-04	1.15E-05
6	3.28E-06	7.84E-06	5.80E-07	7.27E-04	1.61E-05
7	3.83E-06	1.02E-05	6.70E-07	9.79E-04	2.15E-05
8	2.98E-06	1.28E-05	7.80E-07	1.27E-03	2.77E-05
9	2.82E-06	1.57E-05	8.80E-07	1.60E-03	3.47E-05
10	3.38E-06	1.91E-05	8.80E-07	1.97E-03	4.26E-05

Nota. Elaboración propia.

Figura 68. Evaluación del tiempo de operación `matf32_arr_sub`



Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

`matf32_arr_mul`:

Cuadro 38. Tiempo promedio de operación de `matf32_arr_mul` para matrices (n x n)

n	Tiempo (s)				
	MATLAB	**ESP32**	**Linux-amd64**	**Arduino MEGA**	**STM NUCLEO**
1	7.67E-06	2.22E-06	4.20E-07	9.64E-05	2.35E-06
2	1.14E-05	3.76E-06	5.70E-07	4.06E-04	4.67E-06
3	3.93E-06	7.39E-06	7.30E-07	1.25E-03	9.43E-06
4	4.61E-06	1.41E-05	1.06E-06	2.82E-03	1.74E-05
5	6.44E-06	2.48E-05	1.77E-06	5.38E-03	2.94E-05
6	4.19E-06	4.02E-05	2.55E-06	9.26E-03	4.62E-05
7	4.83E-06	6.14E-05	3.26E-06	1.46E-02	6.84E-05
8	4.28E-06	8.92E-05	4.55E-06	2.14E-02	9.71E-05
9	4.33E-06	1.24E-04	6.33E-06	3.06E-02	1.33E-04
10	4.52E-06	1.68E-04	7.39E-06	4.19E-02	1.77E-04

Nota. Elaboración propia.

`matf32_exp`:

Cuadro 39. Tiempo promedio de operación de `matf32_exp` para matrices (n x n)

n	Tiempo (s)				
	MATLAB	**ESP32**	**Linux-amd64**	**Arduino MEGA**	**STM NUCLEO**
1	2.99E-06	1.00E-08	1.54E-06	1.62E-04	3.76E-06
2	5.90E-06	4.00E-08	2.07E-06	7.00E-04	7.93E-06
3	1.20E-06	3.00E-08	3.09E-06	2.11E-03	1.63E-05
4	9.90E-07	3.00E-08	4.60E-06	4.74E-03	3.03E-05
5	3.77E-06	4.00E-08	7.34E-06	8.93E-03	5.10E-05
6	1.06E-06	9.00E-08	1.12E-05	1.53E-02	7.98E-05
7	1.31E-06	1.20E-07	1.65E-05	2.40E-02	1.18E-04
8	1.47E-06	1.30E-07	2.08E-05	3.53E-02	1.67E-04
9	1.38E-06	1.90E-07	2.82E-05	5.02E-02	2.28E-04
10	1.44E-06	2.50E-07	4.05E-05	6.85E-02	3.02E-04

Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

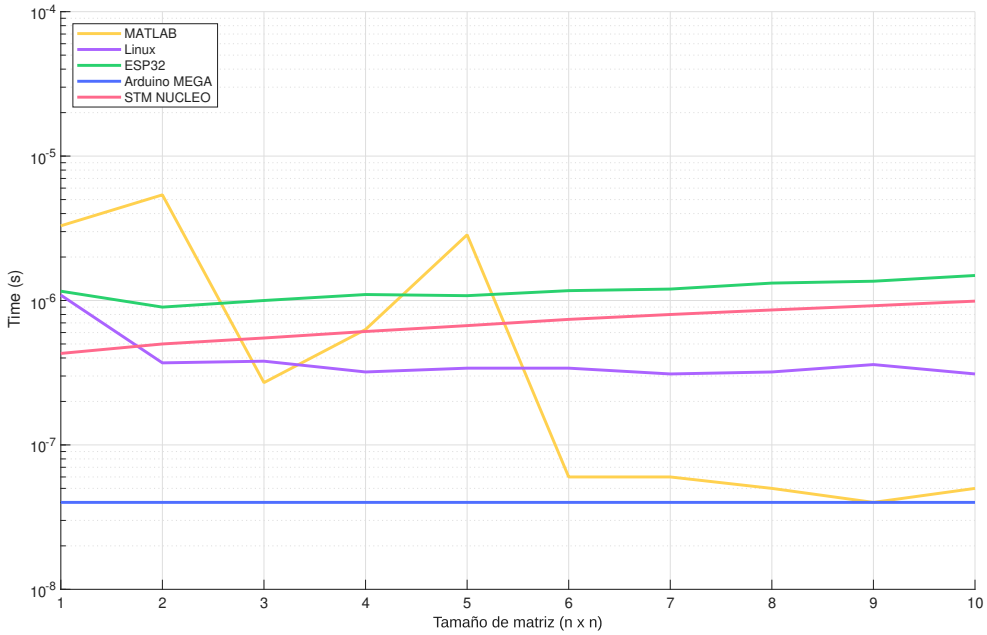
`matf32_dot`:

Cuadro 40. Tiempo promedio de operación de `matf32_dot` para matrices (n x n)

n	Tiempo (s)				
	MATLAB	**ESP32**	**Linux-amd64**	**Arduino MEGA**	**STM NUCLEO**
1	3.29E-06	1.16E-06	1.09E-06	4.00E-08	4.30E-07
2	5.39E-06	9.00E-07	3.70E-07	4.00E-08	5.00E-07
3	2.70E-07	1.00E-06	3.80E-07	4.00E-08	5.50E-07
4	6.30E-07	1.10E-06	3.20E-07	4.00E-08	6.10E-07
5	2.85E-06	1.08E-06	3.40E-07	4.00E-08	6.70E-07
6	6.00E-08	1.17E-06	3.40E-07	4.00E-08	7.40E-07
7	6.00E-08	1.20E-06	3.10E-07	4.00E-08	8.00E-07
8	5.00E-08	1.32E-06	3.20E-07	4.00E-08	8.60E-07
9	4.00E-08	1.36E-06	3.60E-07	4.00E-08	9.20E-07
10	5.00E-08	1.49E-06	3.10E-07	4.00E-08	9.90E-07

Nota. Elaboración propia.

Figura 69. Evaluación del tiempo de operación `matf32_dot`



Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

`matf32_inv`:

Cuadro 41. Tiempo promedio de operación de `matf32_inv` para matrices (n x n)

n	Tiempo (s)				
	MATLAB	**ESP32**	**Linux-amd64**	**Arduino MEGA**	**STM NUCLEO**
1	4.01E-06	2.59E-06	1.40E-06	7.75E-05	3.04E-06
2	7.43E-06	5.52E-06	5.90E-07	3.84E-04	7.11E-06
3	1.83E-06	1.14E-05	7.90E-07	1.08E-03	1.42E-05
4	1.80E-06	2.03E-05	1.26E-06	2.28E-03	2.51E-05
5	3.70E-06	3.40E-05	1.54E-06	4.15E-03	4.04E-05
6	1.32E-06	5.24E-05	2.44E-06	6.87E-03	6.08E-05
7	1.30E-06	7.65E-05	3.29E-06	1.05E-02	8.74E-05
8	1.42E-06	1.07E-04	4.62E-06	1.53E-02	1.20E-04
9	1.73E-06	1.45E-04	6.59E-06	2.13E-02	1.61E-04
10	2.11E-06	1.91E-04	7.29E-06	2.88E-02	2.09E-04

Nota. Elaboración propia.

`matf32_pinv`, calculando $A^+ = (A^\top A)^{-1}A^\top$:

Cuadro 42. Tiempo promedio de operación de `matf32_pinv` con la ecuación $A^+ = (A^\top A)^{-1}A^\top$ para matrices (n x n)

n	Tiempo (s)				
	MATLAB	**ESP32**	**Linux-amd64**	**Arduino MEGA**	**STM NUCLEO**
1	3.71E-06	2.00E-08	1.67E-06	1.99E-04	5.59E-06
2	1.44E-05	5.00E-08	2.33E-06	8.29E-04	1.26E-05
3	3.10E-06	3.00E-08	3.39E-06	2.38E-03	2.50E-05
4	7.75E-06	5.00E-08	5.54E-06	5.20E-03	4.43E-05
5	6.34E-06	7.00E-08	4.78E-06	9.62E-03	7.27E-05
6	4.58E-06	1.30E-07	4.00E-06	1.63E-02	1.12E-04
7	5.82E-06	1.60E-07	4.61E-06	2.53E-02	1.63E-04
8	5.28E-06	1.90E-07	6.58E-06	3.71E-02	2.28E-04
9	5.79E-06	2.90E-07	9.03E-06	5.23E-02	3.09E-04
10	7.61E-06	3.80E-07	1.20E-05	7.12E-02	4.07E-04

Nota. Elaboración propia.

`matf32_pinv` empleando la SVD:

Cuadro 43. Tiempo promedio de operación de `matf32_pinv` empleando la SVD para matrices (n x n)

n	Tiempo (s)				
	MATLAB	**ESP32**	**Linux-amd64**	**Arduino MEGA**	**STM NUCLEO**
1	1.10E-05	3.00E-08	2.71E-06	7.51E-04	1.68E-05
2	1.34E-05	2.40E-07	2.67E-05	1.97E-02	3.61E-04
3	1.14E-05	8.20E-07	9.73E-05	8.03E-02	1.31E-03
4	1.43E-05	1.90E-06	1.89E-04	1.96E-01	3.05E-03
5	2.15E-05	3.49E-06	1.66E-04	4.93E-01	5.70E-03
6	2.23E-05	6.01E-06	1.98E-04	7.00E-01	9.51E-03
7	3.14E-05	9.33E-06	2.94E-04	1.54E+00	1.76E-02
8	3.21E-05	1.40E-05	4.19E-04	2.46E+00	4.50E-02
9	2.88E-05	2.29E-05	6.00E-04	4.15E+00	5.49E-02
10	3.26E-05	4.16E-05	9.68E-04	6.05E+00	8.09E-02

Nota. Elaboración propia.

`matf32_cholesky`:

Cuadro 44. Tiempo promedio de operación de `matf32_cholesky` para matrices (n x n)

n	Tiempo (s)				
	MATLAB	**ESP32**	**Linux-amd64**	**Arduino MEGA**	**STM NUCLEO**
2	6.41E-06	3.32E-06	4.30E-07	2.41E-04	3.29E-06
3	1.76E-06	5.90E-06	5.20E-07	5.33E-04	5.92E-06
4	7.70E-07	9.58E-06	6.00E-07	9.90E-04	9.59E-06
5	3.34E-06	1.44E-05	7.10E-07	1.62E-03	1.45E-05
6	6.00E-07	2.06E-05	8.70E-07	2.43E-03	2.06E-05
7	1.19E-06	2.79E-05	1.14E-06	3.48E-03	2.80E-05
8	6.50E-07	3.68E-05	1.41E-06	4.77E-03	3.68E-05
9	7.40E-07	4.68E-05	1.79E-06	6.30E-03	4.72E-05
10	8.10E-07	5.84E-05	1.87E-06	8.10E-03	5.90E-05

Nota. Elaboración propia.

`matf32_qr`:

Cuadro 45. Tiempo promedio de operación de `matf32_qr` para matrices (n x n)

n	Tiempo (s)				
	MATLAB	**ESP32**	**Linux-amd64**	**Arduino MEGA**	**STM NUCLEO**
2	7.98E-06	1.29E-05	9.80E-07	1.37E-03	3.04E-05
3	3.91E-06	2.60E-05	1.65E-06	3.13E-03	6.09E-05
4	2.98E-06	4.95E-05	2.63E-06	5.85E-03	1.07E-04
5	7.20E-06	8.31E-05	4.28E-06	9.98E-03	1.71E-04
6	2.54E-06	1.29E-04	6.08E-06	1.57E-02	2.56E-04
7	4.72E-06	1.89E-04	8.91E-06	2.37E-02	3.66E-04
8	3.30E-06	2.67E-04	1.26E-05	3.32E-02	5.04E-04
9	3.60E-06	3.63E-04	1.77E-05	4.59E-02	6.72E-04
10	4.02E-06	4.79E-04	1.93E-05	6.12E-02	8.73E-04

Nota. Elaboración propia.

`matf32_lu`:

Cuadro 46. Tiempo promedio de operación de `matf32_lu` para matrices (n x n)

n	Tiempo (s)				
	MATLAB	**ESP32**	**Linux-amd64**	**Arduino MEGA**	**STM NUCLEO**
2	7.69E-06	4.67E-06	1.56E-06	1.86E-04	3.56E-06
3	3.47E-06	8.27E-06	3.43E-06	5.13E-04	9.16E-06
4	1.73E-06	1.35E-05	3.00E-06	1.01E-03	1.66E-05
5	4.76E-06	1.41E-05	3.01E-06	1.76E-03	2.62E-05
6	1.09E-06	2.67E-05	3.76E-06	2.79E-03	3.79E-05
7	1.76E-06	3.71E-05	4.53E-06	3.91E-03	4.36E-05
8	1.55E-06	4.93E-05	3.81E-06	5.79E-03	6.86E-05
9	1.64E-06	5.79E-05	4.59E-06	7.51E-03	7.34E-05
10	1.64E-06	8.06E-05	4.56E-06	1.04E-02	1.10E-04

Nota. Elaboración propia.

`matf32_jacobi_svd`:

Cuadro 47. Tiempo promedio de operación de `matf32_jacobi_svd` para matrices (n x n)

n	Tiempo (s)				
	MATLAB	**ESP32**	**Linux-amd64**	**Arduino MEGA**	**STM NUCLEO**
2	1.55E-05	2.41E-04	3.02E-06	1.98E-04	3.56E-06
3	3.43E-05	7.62E-04	1.39E-05	8.14E-04	1.31E-05
4	5.66E-05	2.10E-03	4.66E-05	1.99E-03	3.05E-05
5	1.01E-04	4.63E-03	1.30E-04	5.01E-03	5.68E-05
6	1.68E-04	9.37E-03	2.96E-04	7.12E-03	9.48E-05
7	1.86E-04	1.77E-02	6.08E-04	1.57E-02	1.75E-04
8	2.51E-04	3.15E-02	1.10E-03	2.50E-02	4.49E-04
9	3.22E-04	5.34E-02	1.85E-03	4.21E-02	5.48E-04
10	4.10E-04	8.65E-02	3.41E-03	6.13E-02	8.07E-04

Nota. Elaboración propia.

12.4. Evaluaciones de `linsolve`: tiempos de operación promedio por tamaño de matriz

Cuadro 48. Tiempo promedio de operación de `linsolve` usando *Forward Substitution* para matrices (n x n)

n	Tiempo (s)				
	MATLAB	**ESP32**	**Linux-amd64**	**Arduino MEGA**	**STM NUCLEO**
8.58E-06	2.09E-06	1.33E-06	6.80E-05	1.46E-06	3.56E-06
1.23E-05	2.29E-06	1.31E-06	1.53E-04	2.05E-06	1.31E-05
1.24E-06	3.35E-06	1.54E-06	2.79E-04	2.82E-06	3.05E-05
1.31E-06	4.58E-06	1.60E-06	4.44E-04	3.83E-06	5.68E-05
2.52E-06	6.22E-06	1.74E-06	6.44E-04	5.06E-06	9.48E-05
1.66E-06	8.03E-06	2.89E-06	8.92E-04	6.48E-06	1.75E-04
8.90E-07	1.01E-05	2.70E-06	1.17E-03	8.19E-06	4.49E-04
1.57E-06	1.21E-05	2.56E-06	1.50E-03	1.00E-05	5.48E-04
7.10E-07	1.49E-05	2.21E-06	1.87E-03	1.22E-05	8.07E-04

Nota. Elaboración propia.

Cuadro 49. Tiempo promedio de operación de `linsolve` usando *Backward Substitution* para matrices (n x n)

n	Tiempo (s)				
	MATLAB	**ESP32**	**Linux-amd64**	**Arduino MEGA**	**STM NUCLEO**
2	4.26E-06	1.41E-06	1.22E-06	6.80E-05	1.46E-06
3	6.23E-06	2.66E-06	1.29E-06	1.73E-04	2.32E-06
4	1.06E-06	3.08E-06	1.52E-06	3.01E-04	3.06E-06
5	1.59E-06	4.67E-06	1.60E-06	4.68E-04	4.05E-06
6	2.09E-06	6.08E-06	2.11E-06	6.74E-04	5.29E-06
7	8.70E-07	7.80E-06	2.42E-06	9.29E-04	6.74E-06
8	7.90E-07	9.89E-06	2.44E-06	1.22E-03	8.37E-06
9	1.53E-06	1.22E-05	2.68E-06	1.55E-03	1.03E-05
10	6.50E-07	1.48E-05	3.13E-06	1.92E-03	1.24E-05

Nota. Elaboración propia.

Cuadro 50. Tiempo promedio de operación de `linsolve` usando Cholesky para matrices (n x n)

n	Tiempo (s)				
	MATLAB	**ESP32**	**Linux-amd64**	**Arduino MEGA**	**STM NUCLEO**
2	4.31E-06	1.54E-06	1.23E-06	6.77E-05	1.49E-06
3	7.52E-06	1.19E-05	2.58E-06	9.21E-04	1.74E-05
4	1.77E-06	1.97E-05	3.38E-06	1.78E-03	2.78E-05
5	1.80E-06	3.02E-05	4.58E-06	3.03E-03	4.17E-05
6	2.93E-06	4.47E-05	6.27E-06	4.73E-03	6.04E-05
7	1.80E-06	6.21E-05	9.71E-06	6.90E-03	8.27E-05
8	2.37E-06	8.29E-05	1.14E-05	9.66E-03	1.10E-04
9	5.33E-06	1.07E-04	1.78E-05	1.30E-02	1.41E-04
10	2.29E-06	1.35E-04	1.23E-05	1.68E-02	1.78E-04

Nota. Elaboración propia.

Cuadro 51. Tiempo promedio de operación de `linsolve` usando QR para matrices (n x n)

n	Tiempo (s)				
	MATLAB	**ESP32**	**Linux-amd64**	**Arduino MEGA**	**STM NUCLEO**
2	3.30E-05	1.17E-06	1.18E-06	5.53E-05	1.32E-06
3	9.51E-06	1.79E-05	3.83E-06	1.01E-03	2.63E-05
4	4.08E-06	3.24E-05	6.28E-06	2.71E-03	5.72E-05
5	3.80E-06	5.85E-05	1.06E-05	5.45E-03	1.05E-04
6	6.00E-06	9.50E-05	1.61E-05	9.58E-03	1.71E-04
7	5.19E-06	1.45E-04	2.70E-05	1.55E-02	2.61E-04
8	1.01E-05	2.10E-04	3.59E-05	2.33E-02	3.76E-04
9	4.43E-06	2.93E-04	4.90E-05	3.35E-02	5.20E-04
10	4.58E-06	3.94E-04	3.53E-05	4.59E-02	6.96E-04

Nota. Elaboración propia.

Cuadro 52. Tiempo promedio de operación de `linsolve` usando LU para matrices (n x n)

n	Tiempo (s)				
	MATLAB	**ESP32**	**Linux-amd64**	**Arduino MEGA**	**STM NUCLEO**
2	2.90E-05	1.73E-06	1.35E-06	6.73E-05	1.49E-06
3	1.09E-05	1.09E-05	2.45E-06	1.30E-03	2.60E-05
4	1.48E-06	1.42E-05	3.07E-06	2.74E-03	4.30E-05
5	1.90E-06	1.97E-05	4.94E-06	4.89E-03	6.06E-05
6	3.38E-06	3.05E-05	5.79E-06	8.17E-03	8.83E-05
7	3.58E-06	3.94E-05	7.29E-06	1.29E-02	1.32E-04
8	2.06E-06	5.27E-05	9.61E-06	1.91E-02	1.86E-04
9	5.10E-06	6.55E-05	9.40E-06	2.66E-02	2.42E-04
10	2.18E-06	8.68E-05	8.10E-06	3.64E-02	3.23E-04

Nota. Elaboración propia.

Cuadro 53. Tiempo promedio de operación de `linsolve` usando SVD para matrices (n x n)

n	Tiempo (s)				
	MATLAB	**ESP32**	**Linux-amd64**	**Arduino MEGA**	**STM NUCLEO**
2	2.80E-04	1.53E-06	4.40E-07	4.78E-02	4.03E-04
3	5.98E-05	1.00E-05	1.18E-06	1.23E-03	2.33E-05
4	2.33E-05	1.32E-05	1.66E-06	2.68E-03	4.01E-05
5	1.88E-05	1.76E-05	2.50E-06	4.99E-03	6.37E-05
6	1.69E-05	2.63E-05	3.50E-06	8.37E-03	9.53E-05
7	1.68E-05	3.95E-05	4.87E-06	1.29E-02	1.32E-04
8	1.78E-05	5.52E-05	6.41E-06	1.89E-02	1.81E-04
9	2.29E-05	6.88E-05	8.51E-06	2.04E+00	2.38E-02
10	1.44E-05	8.94E-05	1.12E-05	3.59E-02	3.04E-04

Nota. Elaboración propia.

12.5. Evaluaciones de quadprog: tiempos de operación promedio por tamaño de matriz

12.5.1. Solución de programas cuadráticos con restricciones de igualdad

Cuadro 54. Tiempo promedio de operación de `quadprog_qp_linsolve` usando LU

n	Tiempo (s)				
	MATLAB	**ESP32**	**Linux-amd64**	**Arduino MEGA**	**STM NUCLEO**
2	2.57E-05	1.34E-05	2.34E-06	6.44E-04	1.86E-05
3	3.04E-05	2.81E-05	4.67E-06	2.11E-03	4.62E-05
4	2.83E-05	4.64E-05	1.88E-06	4.64E-03	7.67E-05
5	2.79E-05	7.31E-05	3.74E-06	8.67E-03	1.20E-04
6	4.72E-05	1.18E-04	6.91E-06	1.48E-02	1.91E-04

Nota. Elaboración propia.

Cuadro 55. Tiempo promedio de operación de `quadprog_qp_linsolve` usando QR

n	Tiempo (s)				
	MATLAB	**ESP32**	**Linux-amd64**	**Arduino MEGA**	**STM NUCLEO**
2	2.57E-05	2.97E-05	4.82E-06	1.83E-03	4.65E-05
3	3.04E-05	7.72E-05	8.16E-06	7.01E-03	1.39E-04
4	2.83E-05	1.72E-04	7.75E-06	1.79E-02	3.12E-04
5	2.79E-05	3.29E-04	1.16E-05	3.71E-02	5.92E-04
6	4.72E-05	5.63E-04	2.72E-05	6.67E-02	1.00E-03

Nota. Elaboración propia.

Cuadro 56. Tiempo promedio de operación de `quadprog_qp_linsolve` usando SVD

n	Tiempo (s)				
	MATLAB	**ESP32**	**Linux-amd64**	**Arduino MEGA**	**STM NUCLEO**
2	2.57E-05	2.27E-04	2.96E-05	2.06E-02	3.71E-04
3	3.04E-05	1.89E-03	2.22E-04	2.04E-01	3.02E-03
4	2.83E-05	6.06E-03	2.79E-04	1.05E+00	1.43E-02
5	2.79E-05	1.76E-02	5.68E-04	2.01E+00	2.98E-02
6	4.72E-05	4.22E-02	1.15E-03	5.58E+00	8.41E-02

Nota. Elaboración propia.

Cuadro 57. Tiempo promedio de operación de `quadprog_qp_ldlt`

n	Tiempo (s)			
	MATLAB	**ESP32**	**Linux-amd64**	**STM NUCLEO**
2	2.46E-03	2.38E-05	1.45E-06	3.65E-05
3	2.20E-03	4.30E-05	2.38E-06	6.92E-05
4	2.55E-03	7.28E-05	3.66E-06	1.19E-04
5	3.57E-03	1.15E-04	5.18E-06	1.89E-04
6	3.13E-03	1.74E-04	7.16E-06	2.84E-04

Nota. Elaboración propia.

Cuadro 58. Tiempo promedio de operación de `quadprog_qp_nullspace`

n	Tiempo (s)			
	MATLAB	**ESP32**	**Linux-amd64**	**STM NUCLEO**
2	2.57E-05	3.44E-05	5.96E-06	4.75E-05
3	3.04E-05	1.05E-04	1.67E-05	1.61E-04
4	2.83E-05	6.02E-03	2.91E-04	9.21E-03
5	2.79E-05	1.34E-02	1.92E-05	2.31E-02
6	4.72E-05	3.61E-02	1.01E-03	5.58E-02

Nota. Elaboración propia.

12.5.2. Problemas cuadráticos con restricciones de desigualdad

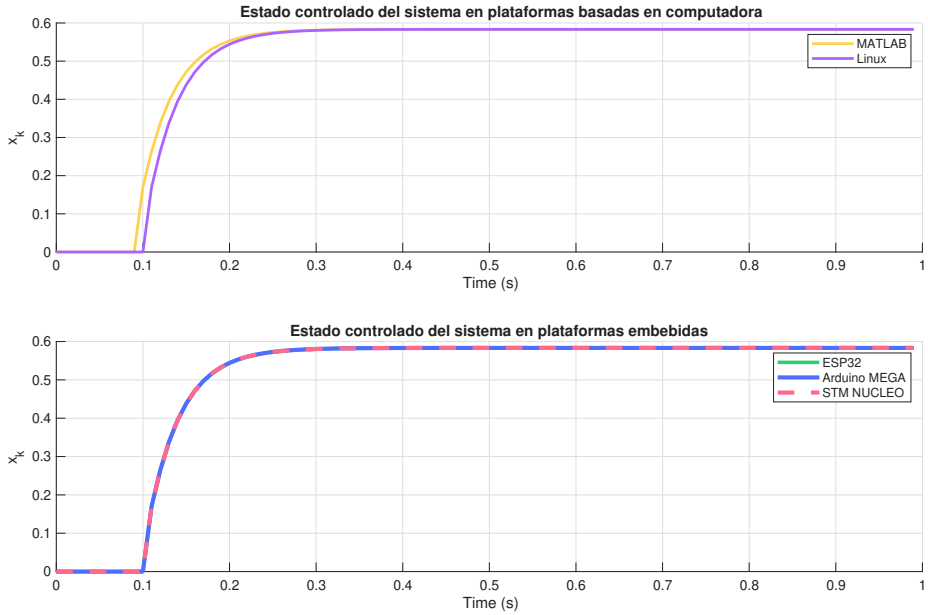
Cuadro 59. Tiempo promedio de operación de `quadprog_sqp` (método de conjunto activo) para resolver problemas con restricciones de igualdad y desigualdad

n	Tiempo (s)				
	MATLAB	**ESP32**	**Linux-amd64**	**Arduino MEGA**	**STM NUCLEO**
2	2.57E-05	2.27E-04	2.96E-05	2.06E-02	3.71E-04
3	3.04E-05	1.89E-03	2.22E-04	2.04E-01	3.02E-03
4	2.83E-05	6.06E-03	2.79E-04	1.05E+00	1.43E-02
5	2.79E-05	1.76E-02	5.68E-04	2.01E+00	2.98E-02
6	4.72E-05	4.22E-02	1.15E-03	5.58E+00	8.41E-02

Nota. Elaboración propia.

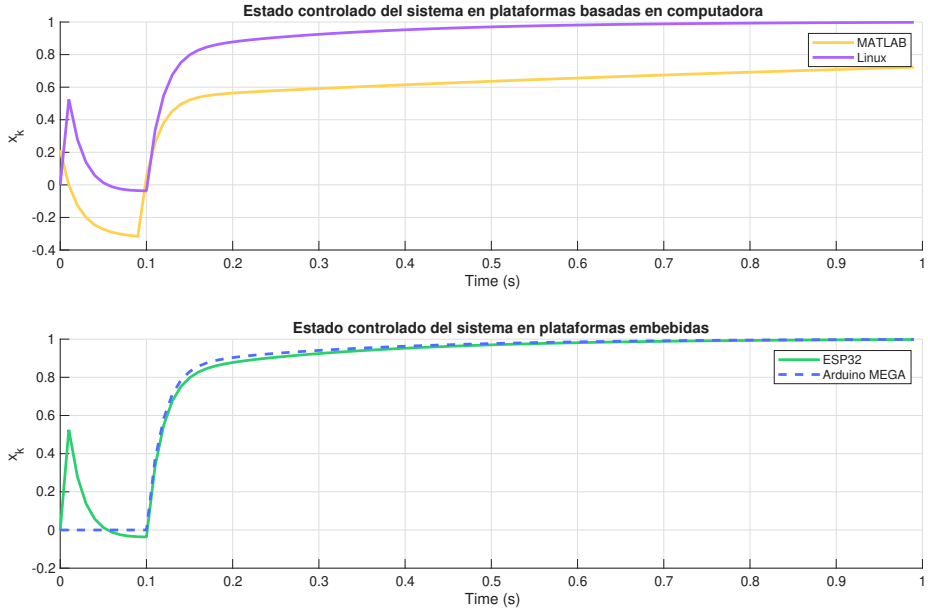
12.6. Evaluaciones de Robotat Control: controlador PID

Figura 70. Vector de estado del sistema LTI controlado por un PID empleando discretización pura



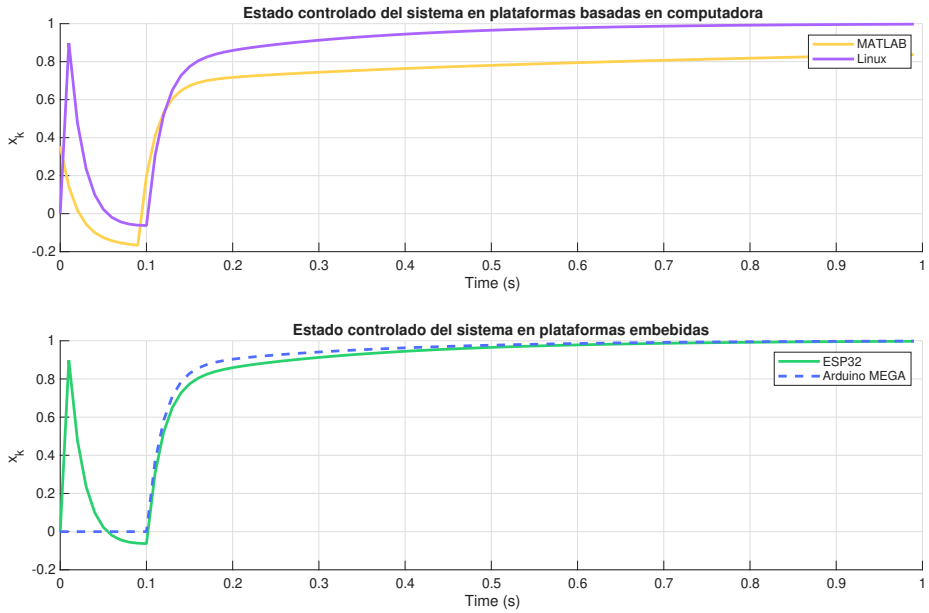
Nota. Estado contra el tiempo de muestreo. Elaboración propia.

Figura 71. Vector de estado del sistema LTI controlado por un PID empleando Forward Euler



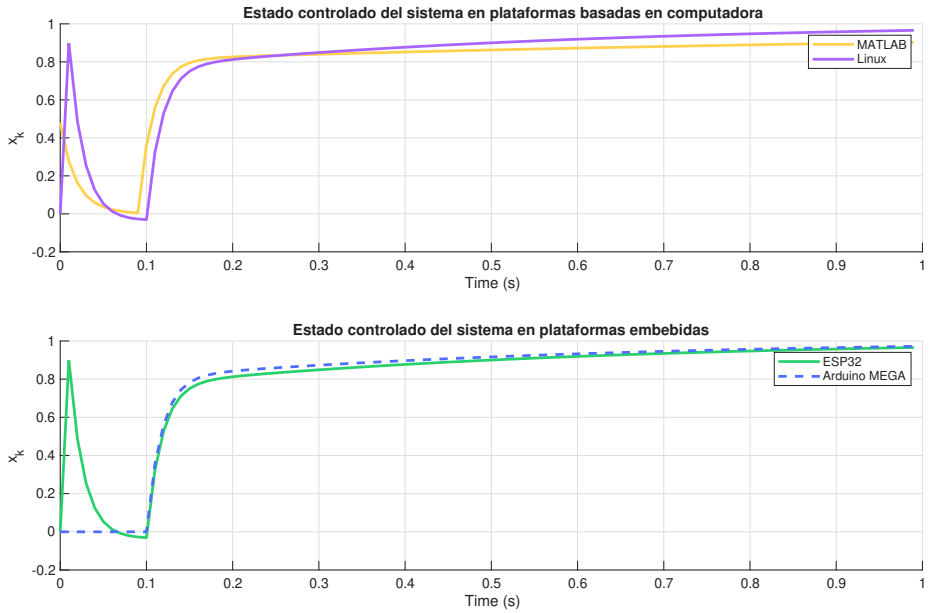
Nota. Estado contra el tiempo de muestreo. Elaboración propia.

Figura 72. Vector de estado del sistema LTI controlado por un PID empleando Backward Euler



Nota. Estado contra el tiempo de muestreo. Elaboración propia.

Figura 73. Vector de estado del sistema LTI controlado por un PID empleando Tustin



Nota. Estado contra el tiempo de muestreo. Elaboración propia.

12.7. Evaluaciones de Robotat Control: discretización de sistemas lineales invariantes en el tiempo (LTI)

Discretización LTI con *Forward Euler*

Cuadro 60. Tiempo promedio de operación de `ctr_c2d` con *Forward Euler*

n	Tiempo (s)				
	MATLAB	**ESP32**	**Linux-amd64**	**Arduino MEGA**	**STM NUCLEO**
2	1.58E-05	1.69E-06	2.00E-08	2.73E-04	5.48E-06
3	1.82E-06	1.84E-06	4.00E-08	4.35E-04	7.60E-06
4	1.68E-06	2.12E-06	2.00E-08	6.41E-04	1.04E-05
5	2.27E-06	1.53E-06	2.00E-08	8.89E-04	1.36E-05
6	6.47E-06	1.26E-06	3.00E-08	1.18E-03	1.74E-05
7	1.33E-06	1.17E-06	4.00E-08	1.51E-03	2.19E-05
8	9.90E-07	1.10E-06	5.00E-08	1.89E-03	2.69E-05
9	9.50E-07	1.20E-06	2.00E-08	2.30E-03	3.25E-05

Nota. Elaboración propia.

Discretización LTI con *Backward Euler*

Cuadro 61. Tiempo promedio de operación de `ctr_c2d` con *Backward Euler*

n	Tiempo (s)				
	MATLAB	**ESP32**	**Linux-amd64**	**Arduino MEGA**	**STM NUCLEO**
2	1.03E-03	3.49E-06	3.00E-08	3.88E-05	2.14E-05
3	5.16E-06	5.23E-06	7.00E-08	3.89E-05	3.43E-05
4	4.51E-06	7.54E-06	6.00E-08	4.00E-05	5.23E-05
5	5.18E-06	7.50E-06	9.00E-08	3.95E-05	7.62E-05
6	8.22E-06	6.81E-06	1.10E-07	3.67E-05	1.07E-04
7	7.32E-06	7.16E-06	1.50E-07	3.93E-05	1.44E-04
8	3.31E-06	7.74E-06	7.00E-08	3.80E-05	1.90E-04
9	3.90E-06	8.65E-06	2.40E-07	3.83E-05	2.44E-04

Nota. Elaboración propia.

Discretización LTI con Tustin

Cuadro 62. Tiempo promedio de operación de `ctr_c2d` con Tustin

n	Tiempo (s)				
	MATLAB	**ESP32**	**Linux-amd64**	**Arduino MEGA**	**STM NUCLEO**
2	5.27E-05	4.38E-06	4.00E-08	1.09E-04	2.60E-05
3	4.86E-06	6.45E-06	6.00E-08	1.09E-04	4.30E-05
4	5.35E-06	9.08E-06	7.00E-08	1.09E-04	6.70E-05
5	5.98E-06	7.17E-06	1.00E-07	1.09E-04	9.92E-05
6	1.61E-04	8.33E-06	1.60E-07	1.09E-04	1.41E-04
7	3.00E-05	9.64E-06	2.40E-07	1.09E-04	1.93E-04
8	5.58E-06	1.09E-05	2.30E-07	1.09E-04	2.56E-04
9	5.97E-06	1.26E-05	3.20E-07	1.09E-04	3.32E-04

Nota. Elaboración propia.

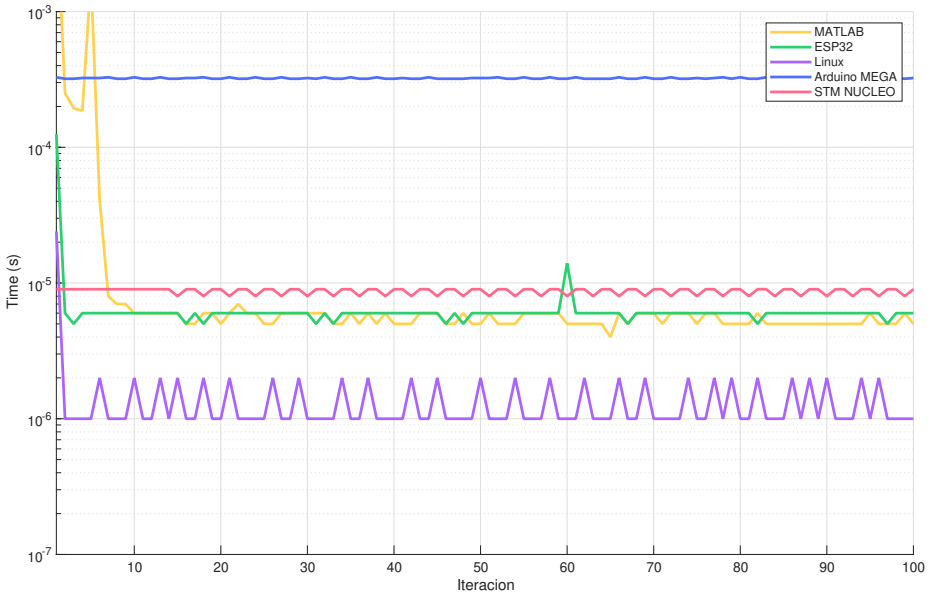
12.8. Evaluaciones de Robotat Control: retroalimentación de estado lineal

Cuadro 63. Tiempo de operación promedio `ctr_linear_state_feedback`.

n	Tiempo (s)				
	MATLAB	**ESP32**	**Linux-amd64**	**Arduino MEGA**	**STM NUCLEO**
2	6.17E-06	1.71E-06	2.70E-07	1.14E-04	2.13E-06
3	2.24E-06	2.30E-07	3.00E-08	3.64E-06	3.40E-07
4	8.70E-07	2.30E-07	3.00E-08	3.80E-06	3.50E-07
5	1.73E-06	2.40E-07	3.00E-08	3.72E-06	3.50E-07
6	5.70E-07	2.30E-07	6.00E-08	3.72E-06	3.80E-07
7	6.70E-07	2.30E-07	4.00E-08	3.72E-06	3.40E-07
8	3.90E-07	2.30E-07	5.00E-08	3.72E-06	3.80E-07
9	3.40E-07	2.30E-07	3.00E-08	3.72E-06	3.50E-07
10	2.70E-07	2.30E-07	3.00E-08	3.68E-06	3.40E-07

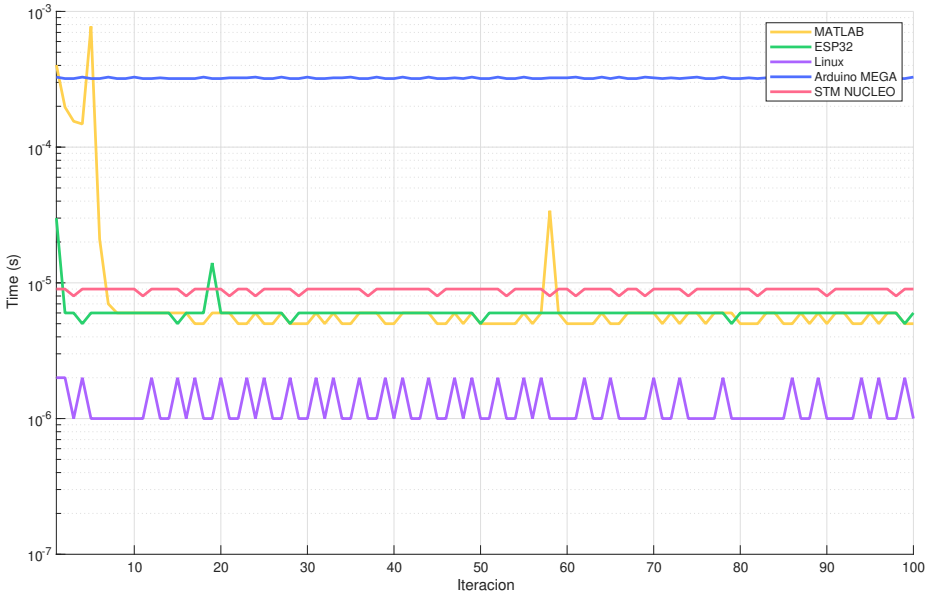
12.9. Evaluaciones de Robotat Robotics

Figura 74. Evaluación del tiempo de operación `rob_rotx`



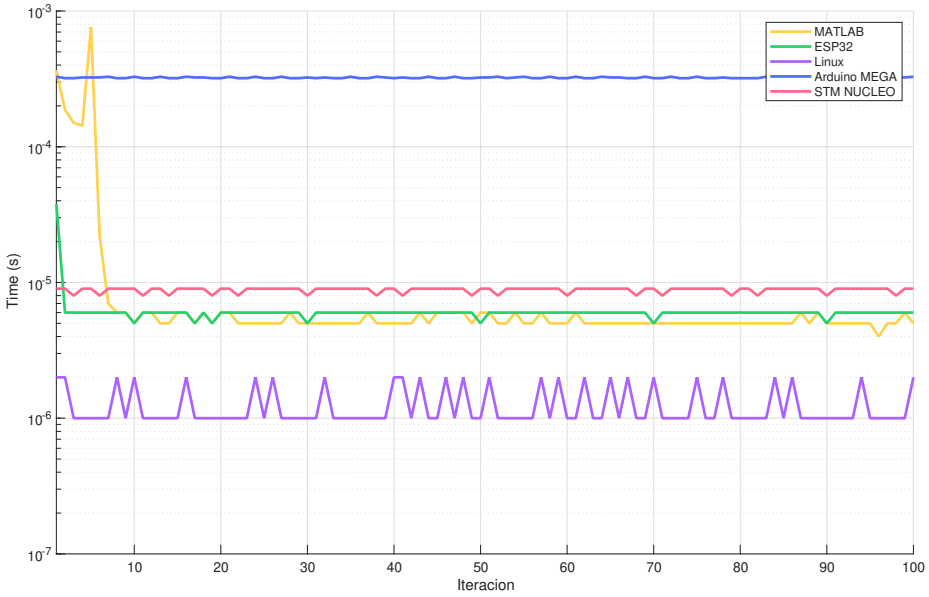
Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

Figura 75. Evaluación del tiempo de operación `rob_roty`



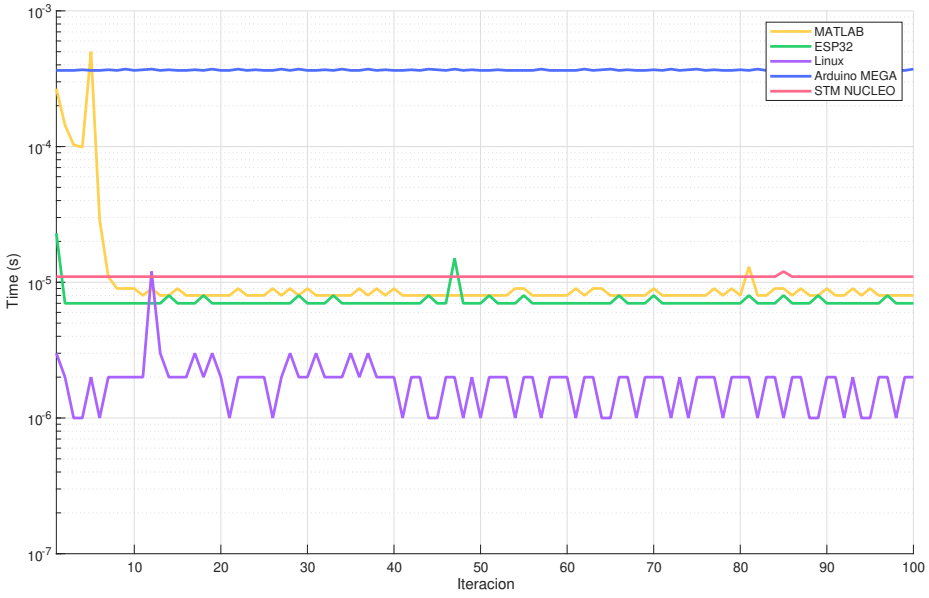
Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

Figura 76. Evaluación del tiempo de operación `rob_rotz`



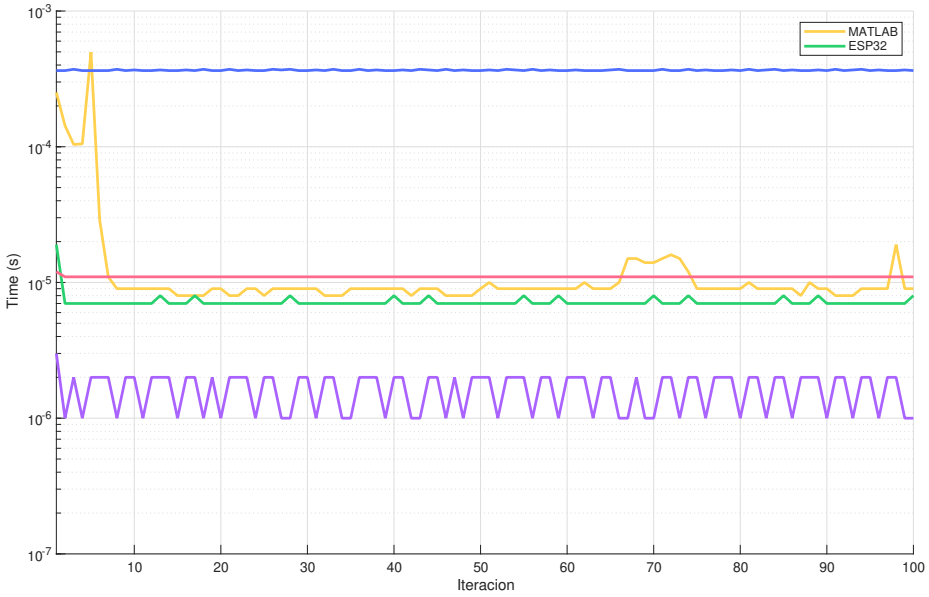
Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

Figura 77. Evaluación del tiempo de operación `rob_trotx`



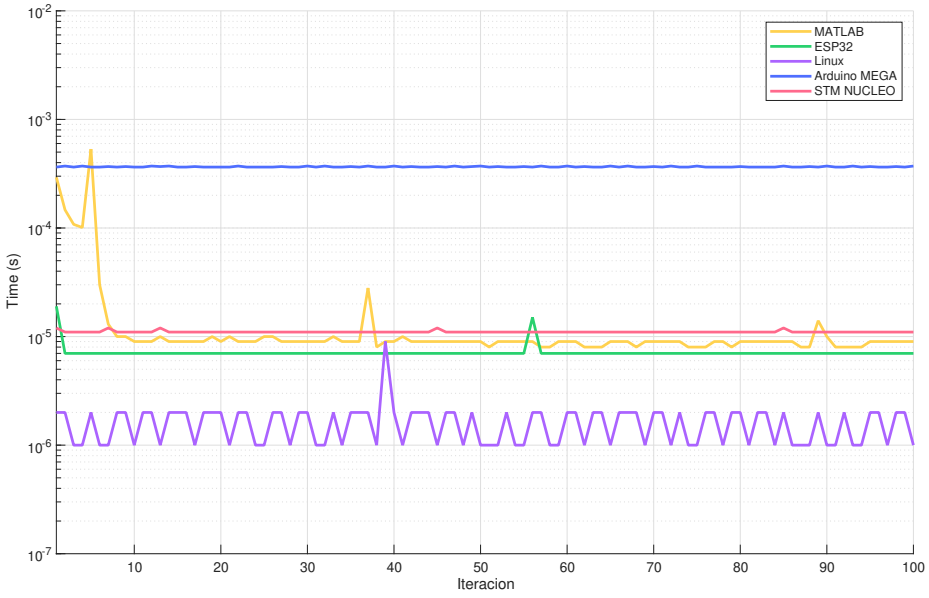
Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

Figura 78. Evaluación del tiempo de operación `rob_troty`



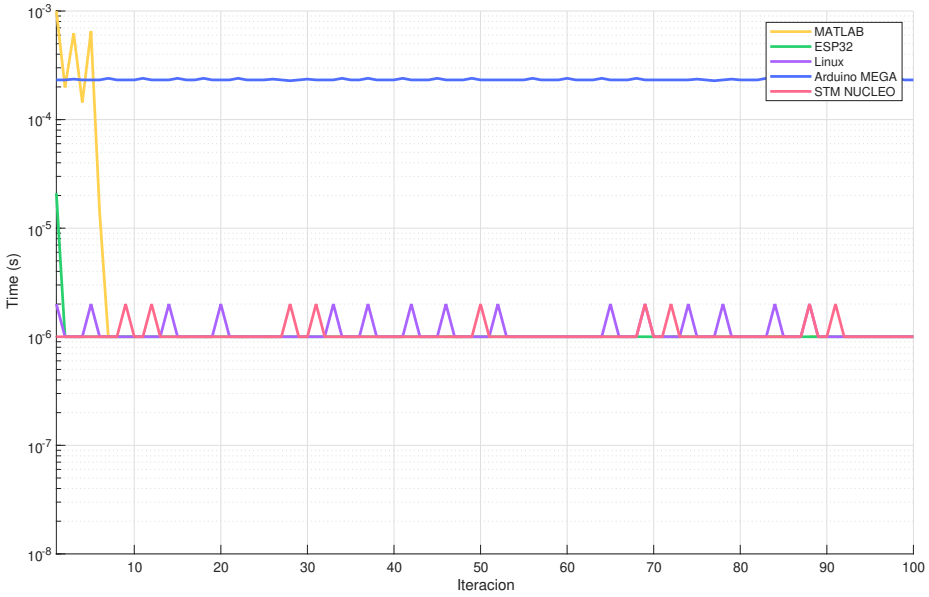
Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

Figura 79. Evaluación del tiempo de operación `rob_trotz`



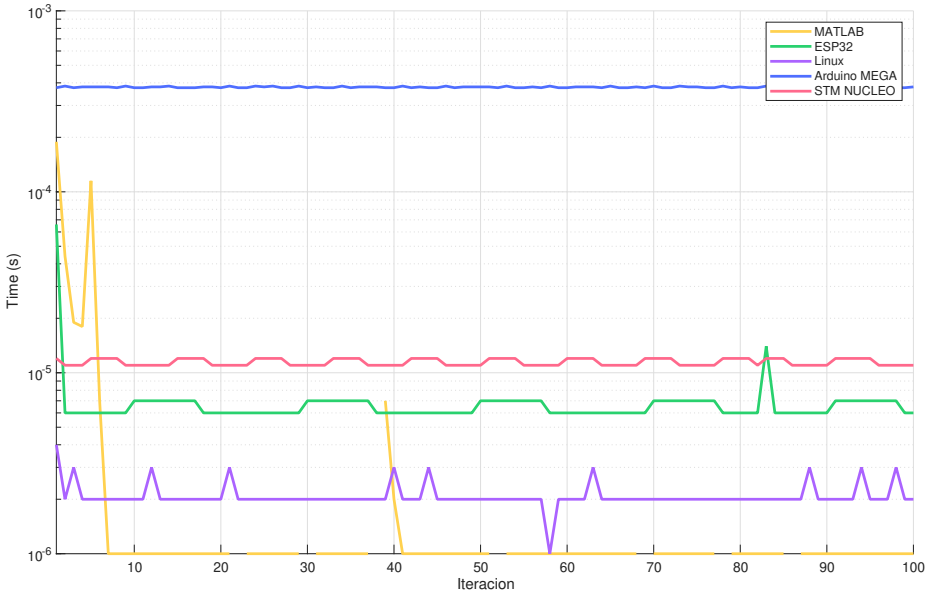
Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

Figura 80. Evaluación del tiempo de operación `rob_apply_transform`



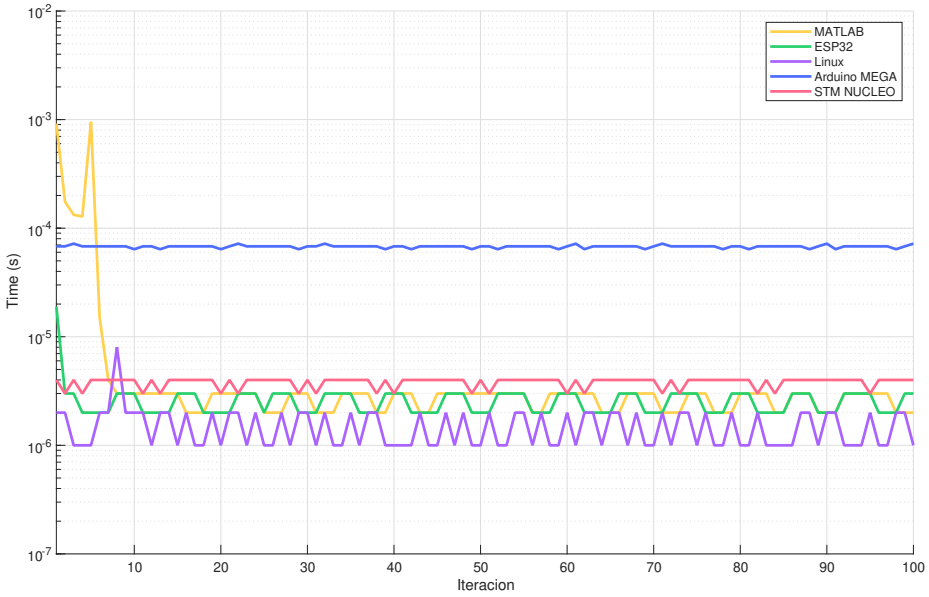
Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

Figura 81. Evaluación del tiempo de operación `rob_inv_transform`



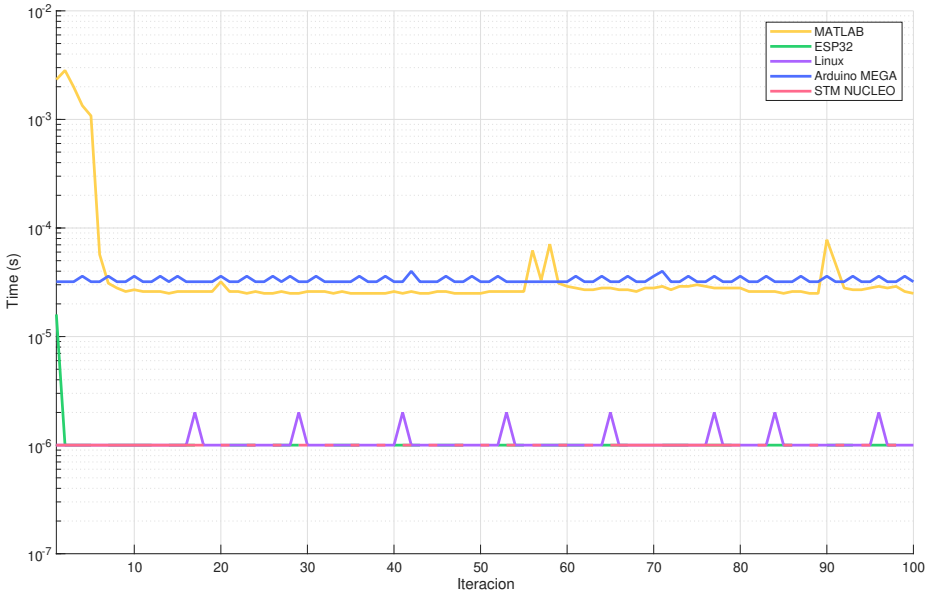
Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

Figura 82. Evaluación del tiempo de operación `rob_update_transform`



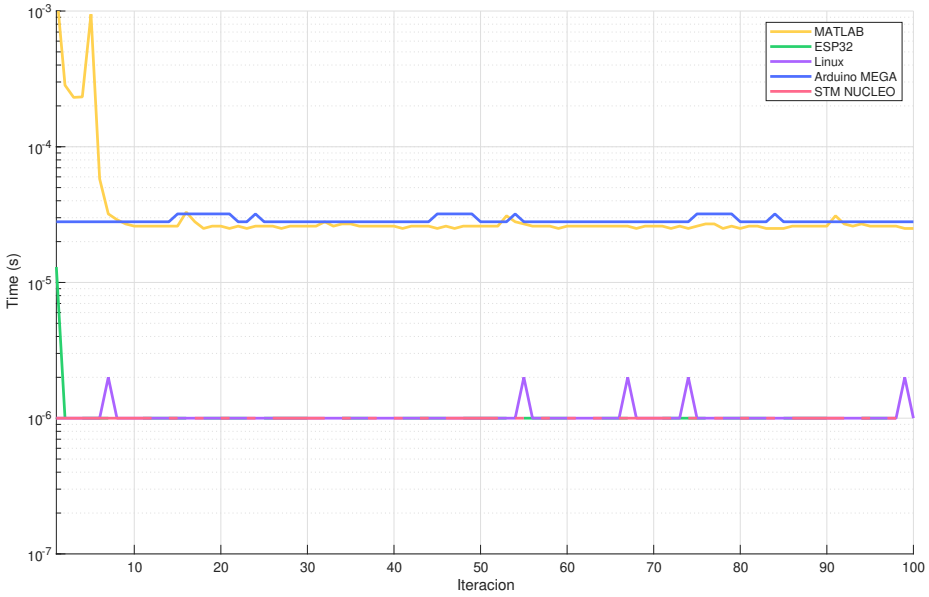
Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

Figura 83. Evaluación del tiempo de operación `rob_quat_add`



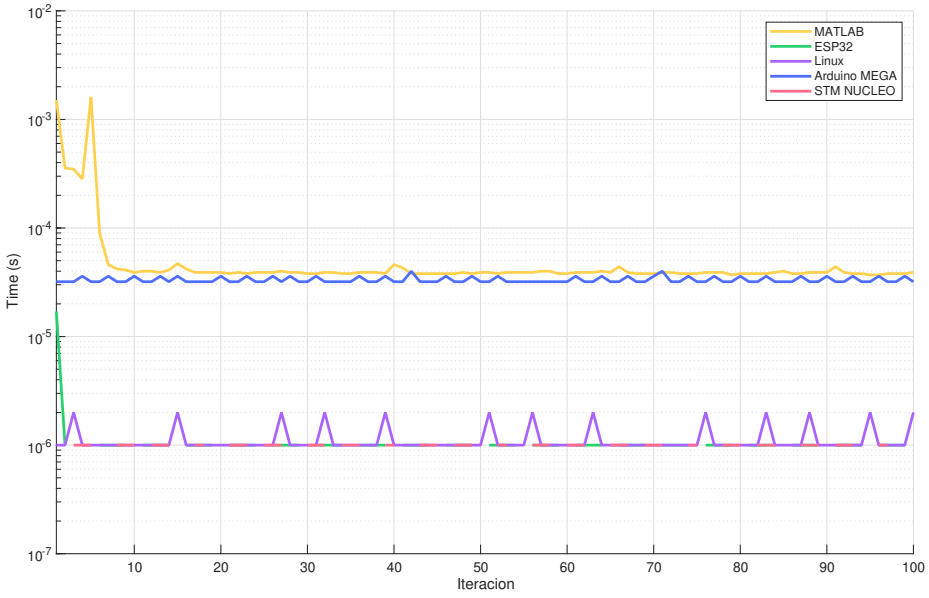
Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

Figura 84. Evaluación del tiempo de operación `rob_quat_sub`



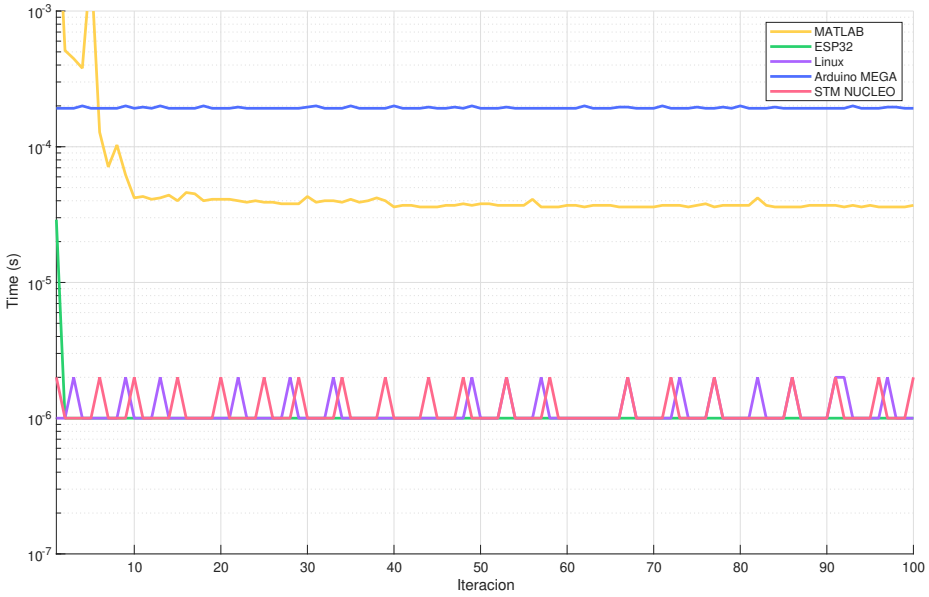
Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

Figura 85. Evaluación del tiempo de operación `rob_quat_scale`



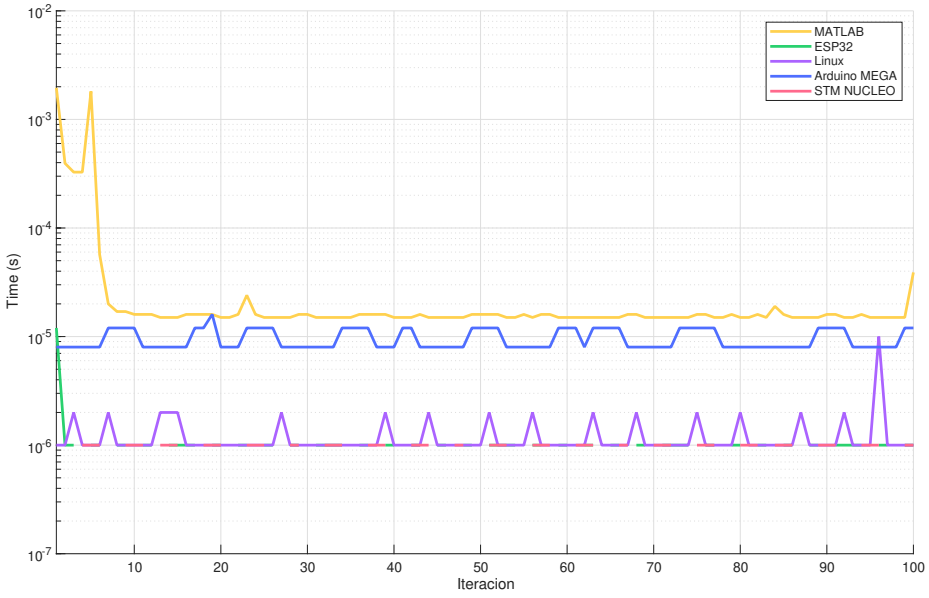
Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

Figura 86. Evaluación del tiempo de operación `rob_quat_mul`



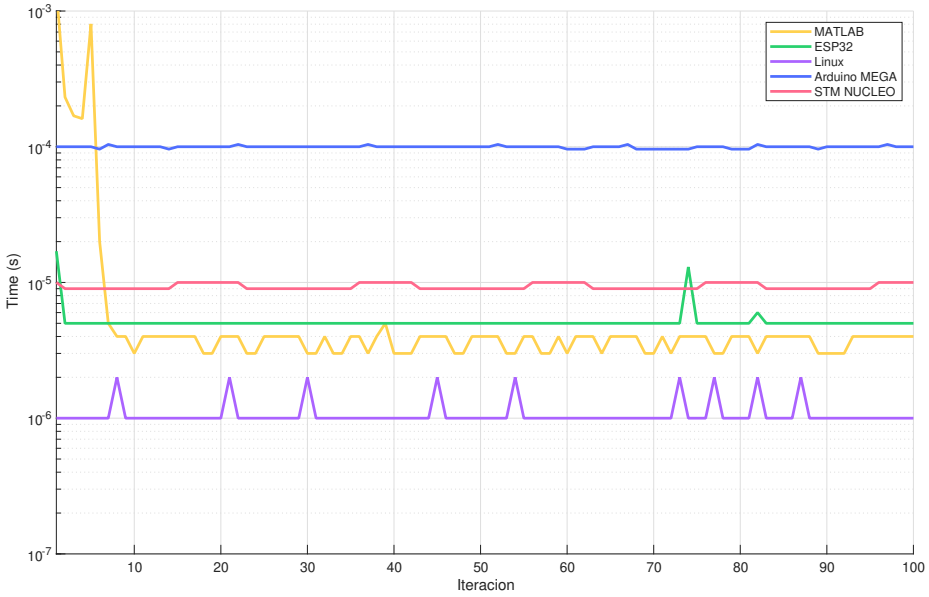
Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

Figura 87. Evaluación del tiempo de operación `rob_quat_conj`



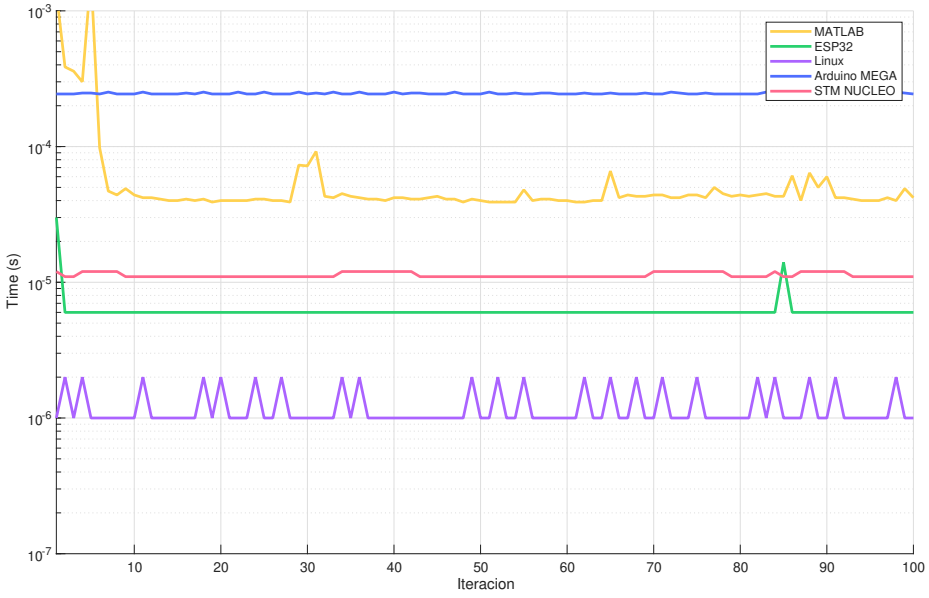
Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

Figura 88. Evaluación del tiempo de operación `rob_quat_norm`



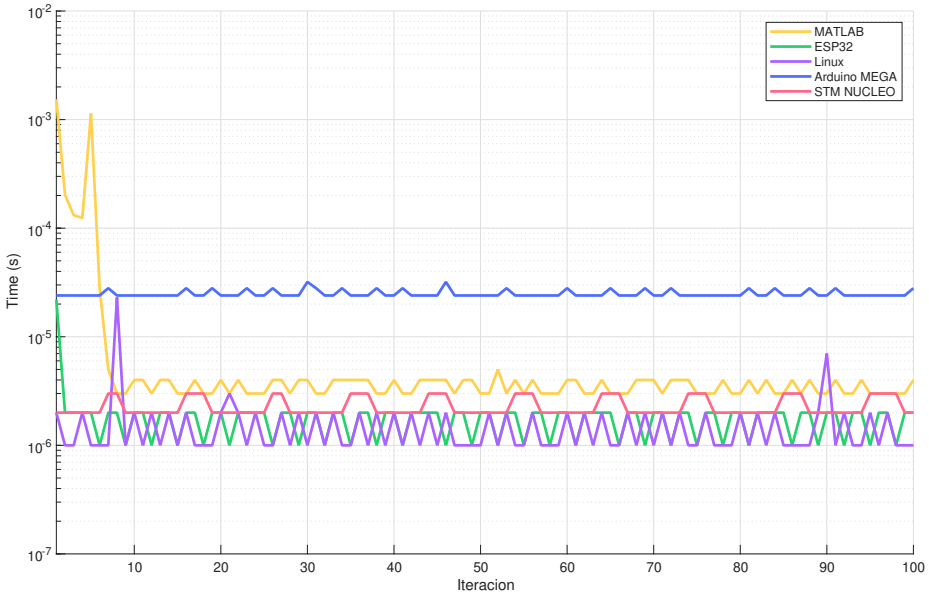
Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

Figura 89. Evaluación del tiempo de operación `rob_quat_inv`



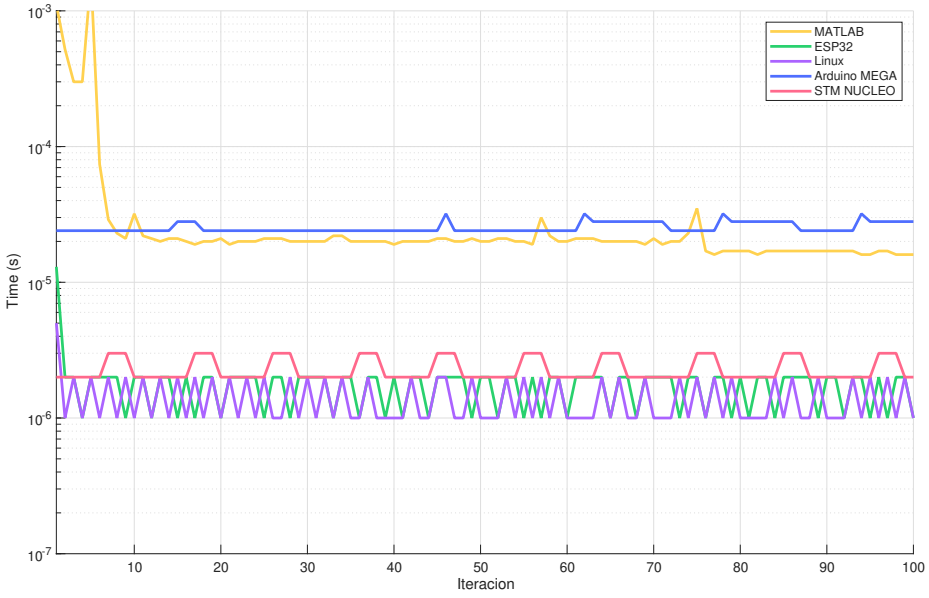
Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

Figura 90. Evaluación del tiempo de operación `rob_rot2tr`



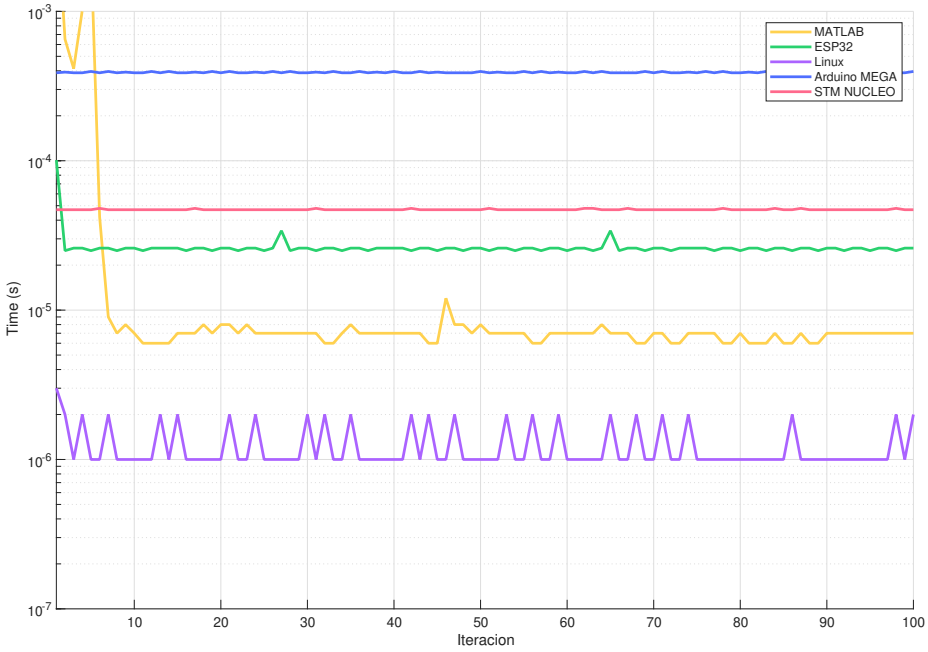
Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

Figura 91. Evaluación del tiempo de operación `rob_tr2rot`



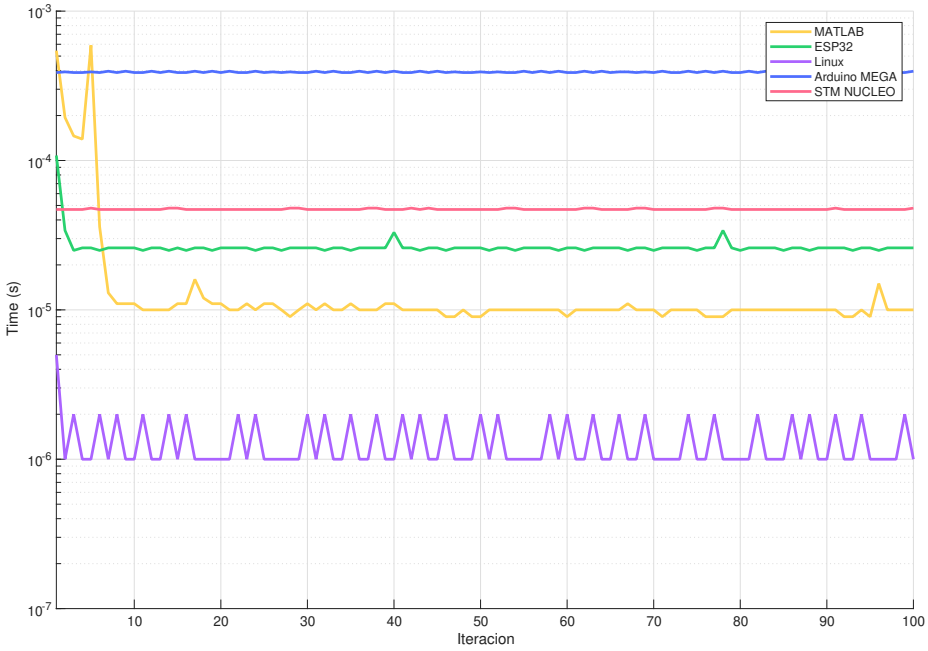
Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

Figura 92. Evaluación del tiempo de operación `rob_rot2quat`



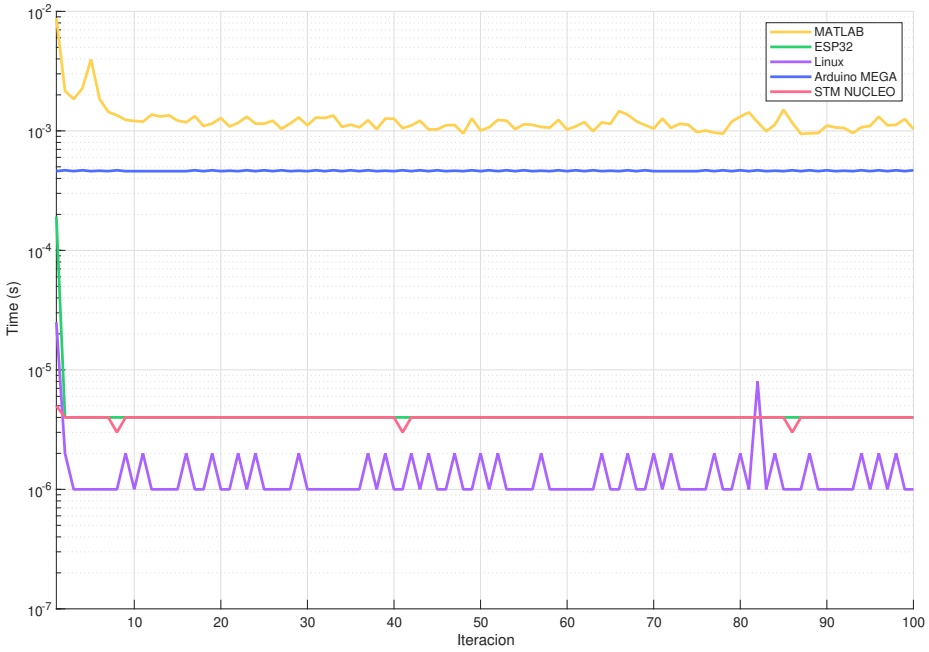
Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

Figura 93. Evaluación del tiempo de operación `rob_tr2quat`



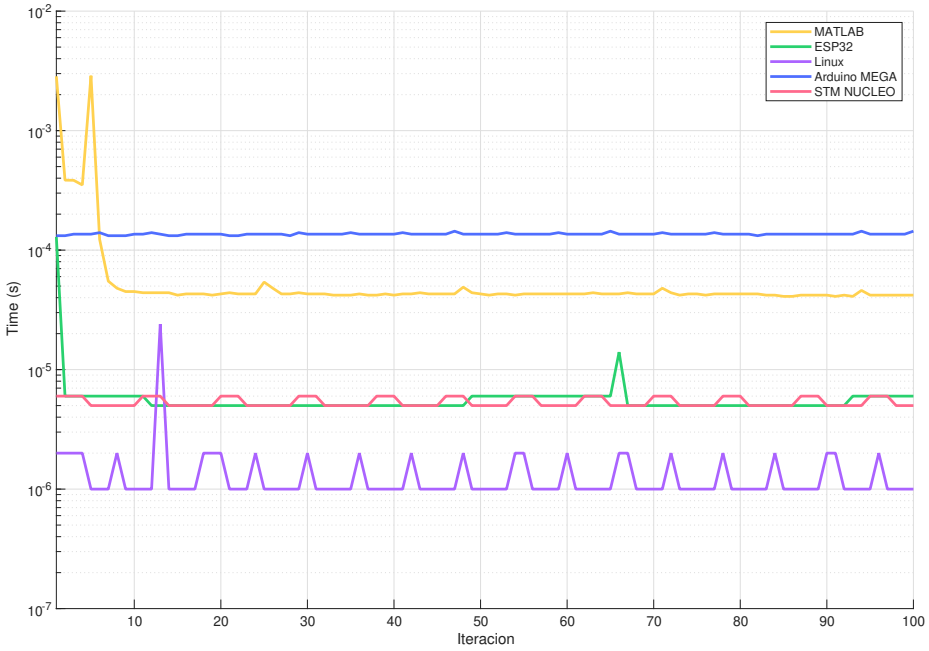
Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

Figura 94. Evaluación del tiempo de operación `rob_tr2rpy`



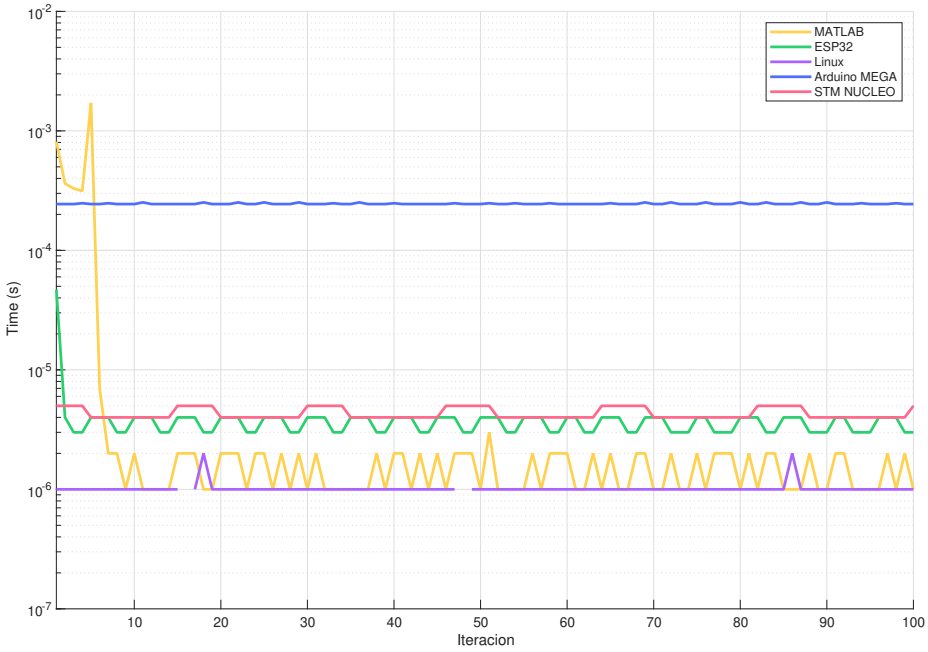
Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

Figura 95. Evaluación del tiempo de operación `rob_tr2eul`



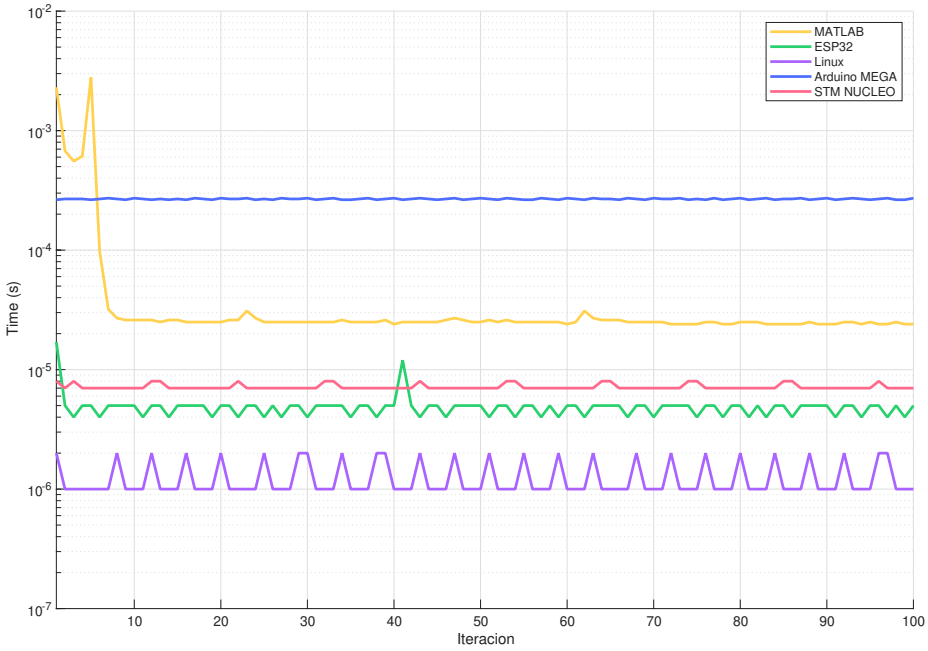
Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

Figura 96. Evaluación del tiempo de operación `rob_quat2rot`



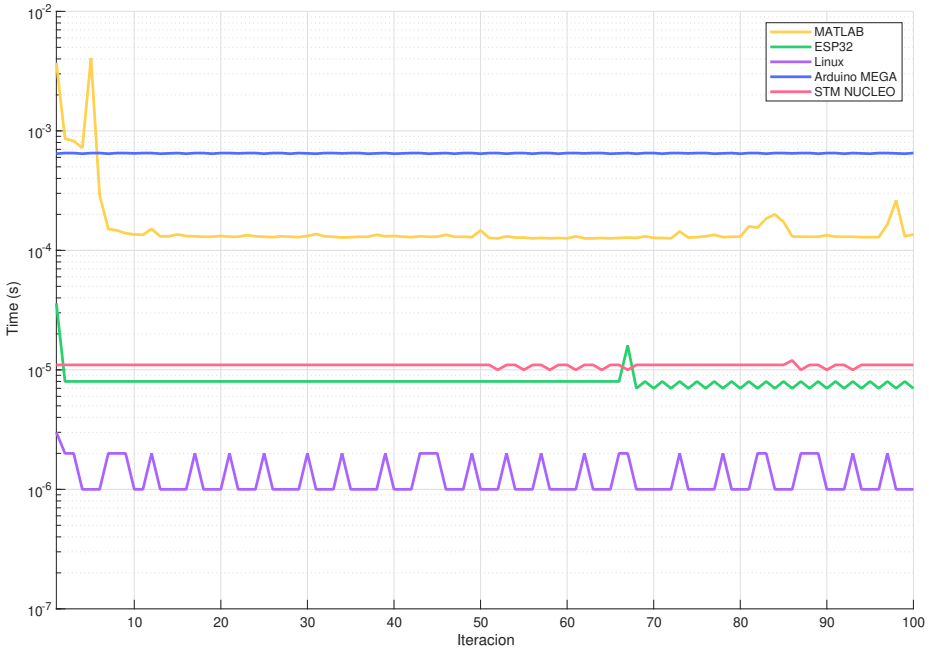
Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

Figura 97. Evaluación del tiempo de operación `rob_quat2tr`



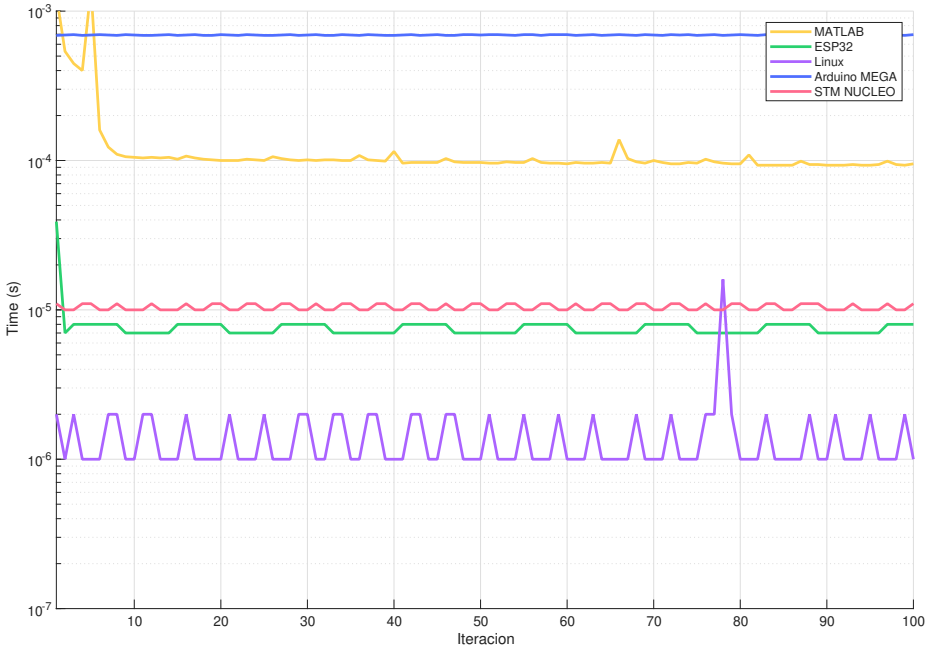
Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

Figura 98. Evaluación del tiempo de operación `rob_quat2rpy`



Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.

Figura 99. Evaluación del tiempo de operación `rob_quat2eul`



Nota. Tiempo de operación promedio contra el tamaño de la matriz. Elaboración propia.