UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE ELECTRÓNICA
VALPARAÍSO - CHILE

IMPLEMENTACIÓN DE ALGORITMOS DE CONTROL PREDICTIVO POR MODELO EN FPGA UTILIZANDO TÉCNICAS DE SÍNTESIS DE ALTO NIVEL

ANDREW GEORGE MORRISON TORRES

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

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

MARZO 2022

Agradecimientos

Me gustaría agradecer a mi profesor guía, Gonzalo Carvajal, por la paciencia que tuvo conmigo y la ayuda que me dio. También quiero dar las gracias a Angel Cedeño por su trabajo previo que sirvió como base para esta memoria, por su disponibilidad y entusiasmo para discutir, y adicionalmente, por responder mis dudas durante este proceso.

Durante el desarrollo de esta memoria recibí apoyo financiero por medio del proyecto FONDECYT Regular 1211676.

Resumen

El presente informe reporta los principales resultados del trabajo de memoria de título orientado a la exploración de técnicas y herramientas para la implementación de algoritmos Control Predictivo por Modelo (*Model Predictive Control* o MPC) en *Field Programmable Gate Arrays* (FPGAs) utilizando el paradigma de Síntesis de Alto Nivel (*High-Level Synthesis* o HLS). Específicamente, se utiliza como base un código ya implementado y validado en MATLAB de un lazo MPC que utiliza el algoritmo *Primal Dual Interior Point* (PDIP) para el control de un motor DC, el cual fue reestructurado y portado a un versión equivalente en C++ . A partir del código en C++ , se exploran optimizaciones y herramientas para su ejecución eficiente en procesadores embebidos, y también para la generación asistida de códigos en Lenguaje de Descripción de Hardware (*Hardware Description Language* o HDL) mediante HLS.

Para comparar y validar las técnicas consideradas, se realizaron distintos experimentos para obtener estimaciones cuantitativas y cualitativas en términos de tiempos de ejecución y complejidad de diseño e implementación en FPGA del lazo MPC de referencia, en relación a un implementación de software ejecutada en un microprocesador embebido. El principal resultado de este trabajo es la documentación de técnicas y procedimientos prácticos para facilitar la exploración de tecnologías y herramientas emergentes para la implementación de MPC en sistemas con dinámicas rápidas.

Palabras Clave. Control predictivo por modelo (MPC), formulación densa de MPC, *high level synthesis* (HLS), C++ , OpenBLAS, *field-programmable gate array* (FPGA).

Abstract

This report shows the main results of the final engineering project oriented to the exploration of tools and techniques for the implementation of Model Predictive Control algorithms (MPC) in Field-Programmable Gate Arrays (FPGAs) using the High-Level Synthesis (HLS) paradigm. Specifically, a code already implemented and validated in MATLAB of an MPC loop that uses the Primal Dual Interior Point (PDIP) algorithm for the control of a DC motor is used as a base, which was restructured and ported to an equivalent version in C++ . From the C++ code, optimizations and tools are explored for its efficient execution in embedded processors, and also for the assisted generation of Hardware Description Language (HDL) codes through HLS.

To compare and validate the considered techniques, different experiments were carried out to obtain quantitative and qualitative estimates in terms of execution times and complexity of design and implementation in FPGA of the reference MPC loop, in relation to a software implementation executed in an embedded microprocessor. The main result of this work is the documentation of practical techniques and procedures to facilitate the exploration of emerging technologies and tools for the implementation of MPC in systems with fast dynamics.

Key Words. Model predictive control (MPC), dense MPC, high level synthesis (HLS), C++ , OpenBLAS, field-programmable gate array (FPGA).

Índice general

1. Introducción **8**
1.1. Motivación y Contexto . . . 8
1.2. Planteamiento del Problema . . . 10
1.3. Alcances y Contribuciones . . . 11
1.4. Organización del Informe . . . 11

2. Antecedentes **13**
2.1. Fundamentos de MPC . . . 13
2.1.1. Definición del problema de control . . . 14
2.1.2. Formulación densa . . . 16
2.1.3. Seguimiento de referencia . . . 18
2.1.4. Restricción de desigualdad . . . 20
2.1.5. Planteamiento final del problema QP . . . 20
2.1.6. Resolución del problema QP . . . 21
2.1.7. Resultado final . . . 21
2.2. Ejemplo MPC Aplicado a Motor DC . . . 22
2.2.1. Modelo del motor y descripción del lazo de control. . . . 22
2.3. Código MATLAB . . . 22
2.3.1. Uso del código MATLAB. . . . 23
2.3.2. Oportunidades de aceleración en hardware. . . . 27

3. Implementaciones de IterMPC **29**
3.1. Implementación MPC_CPP . . . 29
3.1.1. Implementación del algoritmo . . . 30
3.1.2. Parámetros . . . 30
3.2. Implementación MPC_OpenBLAS . . . 31
3.2.1. Implementación del algoritmo . . . 31
3.3. Implementación MPC_HLS . . . 32
3.3.1. Pragmas . . . 32
3.3.2. Implementación del algoritmo . . . 35
3.3.3. Flujo de trabajo con Vitis . . . 40

4. Evaluación Experimental **41**
4.1. Metodología . . . 41
4.2. Implementaciones MPC_CPP y MPC_OpenBLAS . . . 43
4.2.1. Caracterización de tiempos de ejecución de iterMPC . . . 43

4.3. Implementaciones MPC_OpenBLAS y MPC_HLS 47
4.3.1. Caracterización de tiempos de ejecución de MPC 48

5. Conclusiones y Trabajo Futuro **51**
5.1. Conclusiones 51
5.2. Trabajo Futuro 52

Índice de tablas

2.1. Tamaños de matrices no constantes de entrada y salida en `iterMPC`, `PDIP` y `myChol`, para diferentes horizontes de predicción. 27
2.2. Tiempos de ejecución para 10.000 iteraciones de `iterMPC` con diferentes horizontes de predicción. 28

3.1. Tabla comparativa entre las implementaciones de multiplicación de matrices. . . 40

4.1. Análisis estadístico de los tiempos de ejecución, por iteración, de `iterMPC` en Bombe. 44
4.2. Análisis estadístico de los tiempos de ejecución, por iteración, de `iterMPC` en ZCU104. 45
4.3. Uso de Recursos de MPC_HLS en FPGA. 48
4.4. Análisis estadístico de los tiempos de ejecución de MPC_OpenBLAS y MPC_HLS ejecutados en ZCU104. 49

Índice de figuras

2.1. Lazo de control para motor. 22
2.2. Comparación de velocidad angular ω para diferentes métodos de control. 25
2.3. Comparación de posición angular θ para diferentes métodos de control y la referencia a seguir. 25
2.4. Comparación de el voltaje de entrada V_m para diferentes métodos de control. . 26

3.1. Flujo de trabajo utilizando High Level Synthesis. 33
3.2. Comparación como se guarda A con y sin *array partition*. 34
3.3. Visualización de un *pipeline* para un ciclo *for* 35
3.4. Multiplicación de matrices sin optimizaciones ni pragmas. 36
3.5. Orden temporal en que se ejecutan las etapas del *pipeline* para mmult. 37
3.6. Ejemplo ilustrativo de multiplicación de matrices. 37
3.7. Etapas del *pipeline* para la multiplicación de matrices. 38
3.8. Orden temporal en que se ejecutan las etapas del *pipeline* para mmultHardware. 38

4.1. Tiempo medio de `iterMPC` en Bombe para $N = 2, 3, 4, 6, 8, 16, 32, 64$. 45
4.2. Tiempo medio de `iterMPC` en ZCU104 para $N = 2, 3, 4, 6, 8, 16, 32, 64$. 46
4.3. Tiempos de ejecución para diez pruebas en Bombe, con *float* y $N = 16$. 46
4.4. Tiempos de ejecución para diez pruebas en ZCU104, con *float* y $N = 6$. 47
4.5. Tiempos medio de iterMPC en ZCU104 y FPGA. 49
4.6. Tiempos medio de ejecución para diez pruebas en ZCU104, con *float* y $N = 6$. . 50

1 Introducción

Este capítulo presenta el contexto y la motivación para el desarrollo de este trabajo, especifica el problema a resolver, los objetivos, alcances y contribuciones de esta memoria de título, y finalmente presenta la organización del resto del informe.

1.1. Motivación y Contexto

Control predictivo por modelo (*Model Predictive Control* o MPC) es una técnica de control avanzada basada en optimización que, utilizando un modelo matemático del sistema a controlar y el valor de los estados del sistema en un instante de tiempo dado, es capaz de predecir el comportamiento del sistema y determinar el valor óptimo de las entradas para que el sistema siga una referencia a lo largo de muestras futuras, sujeto a restricciones en los valores de las entradas y los estados [1].

En términos de procesamiento, implementaciones prácticas de MPC para sistemas lineales requieren resolver, para cada instante de muestreo, un problema de Programación Cuadrática (*Quadratic Programing* o QP) para encontrar la secuencia óptima de valores de entradas que minimizan una función de costo predefinida a lo largo de un horizonte de predicción de N muestras. Los algoritmos de optimización son, en general, demandantes en términos de recursos computacionales y costosos en tiempo de ejecución. Esta característica ha restringido la adopción de algoritmos MPC a sistemas con dinámicas lentas que pueden operar lazos de control con tiempo de muestreo en el orden de minutos u horas [2–4].

Teniendo como motivación la expansión del campo de aplicaciones de MPC a sistemas con dinámicas rápidas en el orden de fracciones de segundo, la literatura muestra varios trabajos que apuntan a reducir los tiempos de cómputos para resolver el problema de optimización. Por un lado, la aceleración de cómputo se logra al explotar particularidades de los sistemas físicos a controlar que permiten simplificar la formulación del problema QP e implementar algoritmos ad-hoc que reducen la complejidad matemática y el costo computacional. Por otro lado, muchos de estos algoritmos además se pueden acelerar mediante el uso de hardware paralelo, arquitecturas especializadas, o una mezcla de ambos [5–7]. Respecto a este último punto, la literatura reciente reporta múltiples implementaciones de QP solvers en *Field Programmable Gate Arrays* (FPGAs) basadas en el uso intensivo de operaciones de álgebra lineal que pueden ser eficientemente mapeadas a arquitecturas paralelas [4, 8]. En específico, una FPGA ofrece capacidades de especialización de cómputo en términos de representación de datos, resolución aritmética, frecuencia de reloj, y tipo de paralelismo para alcanzar el desempeño deseado considerando los diferentes *trade-offs*. El grado de control fino sobre la arquitectura de hardware permite alcanzar

niveles de desempeño y determinismo temporal que no son factibles de alcanzar con sistemas de arquitectura fija, como microcontroladores, DSPs o GPUs. Por ejemplo, el trabajo reportado en [9] provee una descripción y comparación de múltiples implementaciones de optimizadores en FPGAs. En general, este trabajo provee evidencia de que a pesar de los avances que han habido en este campo en los últimos años, aún quedan aspectos teóricos, técnicos, y tecnológicos por resolver antes de alcanzar implementaciones de MPC rápidas y económicamente viables para ser integradas en sistemas industriales complejos. Uno de los aspectos por resolver apunta a la alta complejidad y costo de diseño de sistemas de cómputo especializados utilizando lógica programable.

Tradicionalmente, las FPGAs se programan utilizando un Lenguaje de Descripción de Hardware (Hardware Description Language, HDL), como Verilog, SystemVerilog o VHDL, los que permiten definir la estructura y comportamiento de un circuito digital a bajo nivel, típicamente usando la abstracción de transferencia entre registros (*Register-Transfer Level*, RTL). Este enfoque le ofrece al ingeniero control fino sobre el diseño mientras abstrae la síntesis del circuito subyacente; no obstante, es ampliamente reconocido que describir hardware al nivel RTL tiende a ser costoso en términos de horas hombre necesarias para el diseño, y posterior descripción, implementación, y verificación del sistema implementado. En general, los procesos de descripción a bajo nivel, optimización y validación de un sistema digital en FPGA suelen ser engorrosos y requieren una expertiz en diseño de hardware, lo cual dificulta la replicación y reutilización de los diseños reportados en la literatura.

En los últimos años, el paradigma de Síntesis de Alto Nivel (*High-Level Synthesis* o HLS) ha emergido como una alternativa al diseño RTL tradicional. En términos generales, HLS permite tomar un algoritmo escrito en un lenguaje de alto nivel como C o C++ y automáticamente mapearlo a una descripción RTL que puede ser directamente implementada en un FPGA. Para ayudar a este proceso, se utilizan directivas en el código de alto nivel, en forma de *pragmas*, que permiten al usuario especificar las propiedades de la arquitectura de hardware para optimizar cierta parte del código, ya sea en términos de latencia, uso de recursos, y nivel y tipo de paralelismo (espacial o temporal). Por un lado, el paradigma de HLS promete facilitar y acelerar la exploración del espacio de diseño por medio de la generación automática de diferentes alternativas de implementación utilizando distintos pragmas sobre un mismo código base, lo cual permite reducir significativamente la complejidad y el tiempo de diseño en comparación a escribir un código HDL específico para cada arquitectura que se quiera probar. Por otro lado, este concepto, promisorio en teoría, sigue siendo limitado en la práctica, ya que no cualquier código C/C++ puede ser directamente y eficientemente mapeado a una implementación en FPGA, existiendo restricciones como que no se pueden utilizar arreglos de tamaño dinámico, llamadas a sistema operativo, ni funciones recursivas, entre otros. Además, para sacar el mejor provecho a HLS, se debe tener una idea de cuál es el hardware objetivo que se quiere describir para poder utilizar correctamente los pragmas, por lo que sigue siendo necesario tener experiencia con diseño digital. Es también bien sabido que elevar el nivel de abstracción viene con el costo de perder control fino sobre la arquitectura generada, lo que suele derivar en una pérdida de desempeño con respecto a lo que se podría obtener con un diseño optimizado a bajo nivel. En consecuencia, el uso de HLS sería válido solo si la pérdida de desempeño con respecto a una implementación realizada a bajo nivel no afecta los requerimientos de la aplicación objetivo.

Este trabajo representa un primer paso para adquirir experiencia práctica en el uso del paradigma HLS para la implementación de algoritmos de MPC en FPGAs. Como caso de

estudio para canalizar el trabajo, se utilizó un ejemplo específico de control de un lazo MPC para un motor DC descrito en MATLAB para analizar e identificar los potenciales para aceleración de cómputo y evaluar los *trade-off* asociados a la implementación en FPGA del algoritmo en términos de latencia, precisión de los resultados, y complejidad de diseño e implementación. Todos los códigos y documentación generada a partir de este trabajo están disponibles en un repositorio [10] para facilitar la reproducción de los resultados y la continuación del trabajo en futuros proyectos. Para acceder al repositorio se necesita pedir autorización de acceso.

1.2. Planteamiento del Problema

Este trabajo de título apunta a evaluar la efectividad de procedimientos, técnicas, y herramientas actualmente disponibles orientadas a HLS para facilitar el diseño e implementación de algoritmos MPC en FPGAs.

Para focalizar el trabajo, utilizaremos como caso de estudio específico un código de referencia que simula un lazo de control MPC aplicado a un motor DC. El código de referencia se encuentra escrito en MALTAB, y utiliza la formulación densa y el algoritmo *Primal Dual Interior Point* (PDIP) para resolver los problemas de optimización cuadrática en cada instante de muestreo. Durante las evaluaciones y discusiones con profesores y estudiantes previas a esta memoria de título, se planteó que la optimización computacional de este caso de estudio era de interés para otros trabajos y proyectos que se estaban desarrollando en el Departamento de Electrónica. Además, el contar con una referencia funcional ya definida y validada, permite abstraernos de la teoría de control y algoritmos subyacentes y enfocarnos exclusivamente en los aspectos computacionales para poder implementar, evaluar y comparar el desempeño de este algoritmo en distintas plataformas embebidas. Se espera que, a partir de los resultados obtenidos sobre el caso de estudio específico, se puedan derivar directrices generales con base práctica para facilitar la exploración de nuevas técnicas y aplicaciones en trabajos futuros.

El desarrollo de este trabajo considera las siguientes etapas:

- **Obtención de una versión en C++ funcionalmente equivalente al código de referencia en MATLAB.** El código escrito en C++ se utiliza como referencia para analizar aspectos de estructuras de datos y flujo de procesamiento, con el fin de identificar cuellos de botella en el procesamiento y potenciales secciones de código que puedan ser aceleradas al ejecutarse en CPUs y FPGAs.
- **Integración de librerías especializadas para operaciones de álgebra lineal.** Considerando la premisa de que los algoritmos clásicos de MPC hacen uso intensivo de operaciones de álgebra lineal para resolver el problema de optimización en cada tiempo de muestreo, se explora el uso de librerías especializadas para facilitar el diseño y acelerar la ejecución en procesadores de escritorio y embebidos.
- **Evaluación del uso de pragmas para la generación de código HDL mediante HLS.** Siguiendo documentación general sobre el paradigma HLS y específica a las herramientas de Xilinx para esta tarea, se realizan los cambios necesarios al código C++ para que se pueda generar código HDL, para luego estudiar los efectos del uso de pragmas para obtener implementaciones eficientes bajo distintos criterios.
- **Evaluación experimental.** Se comparan los tiempos de ejecución de las implementacio-

nes en C++ (ejecutadas en procesadores de escritorio y embebido) y la obtenida mediante HLS (implementado en una FPGA).

- **Generación de un repositorio con documentación y ejemplos.** Se crea un repositorio GIT [10] en el que se encuentran todos los códigos e instrucciones necesarios para reproducir los resultados aquí reportados.

1.3. Alcances y Contribuciones

En base a evaluaciones preliminares previas a la definición del proyecto de memoria, para este trabajo consideramos los siguientes alcances:

- El trabajo se acota al estudio, implementación, y evaluación de un algoritmo MPC que utiliza la formulación densa del problema de control y el método PDIP para resolver el problema de programación cuadrática en cada instante de muestreo. Dentro de la variedad de alternativas de implementación para un lazo MPC, se escoge esta configuración en particular ya que es de interés para otros estudios y proyectos que están en ejecución en el Departamento de Electrónica al momento de desarrollar esta memoria.
- Los códigos en C++ derivados del código de referencia en MATLAB son ejecutados en un computador de escritorio con procesador AMD y en una tarjeta de desarrollo Zynq UltraScale+ MPSoC ZCU104 de Xilinx, la cual incluye un procesador ARM Cortex-A53 de 4 núcleos y bloques de lógica programable integrados en el mismo chip.
- Como entorno de desarrollo se utilizan las herramientas Vitis HLS 2020.2 y Vitis 2020.2 de Xilinx, las cuales entregan soporte completo para el hardware especificado en el punto anterior.
- En base a evaluaciones preliminares, para paralelizar las operaciones de álgebra lineal se utiliza la biblioteca OpenBLAS [11] que provee rutinas de bajo nivel optimizadas para operaciones comunes de álgebra lineal en procesadores multi-núcleo.

La principal contribución de este trabajo es la generación y reporte de evidencia empírica sobre las capacidades y limitaciones del uso de herramientas de HLS para la implementación de algoritmos MPC en sistemas embebidos. Como soporte al reporte escrito, se provee un conjunto de códigos y ejemplos que permiten reproducir los resultados acá reportados. Se espera que estos resultados provean directrices para continuar esta línea de trabajo y sistematizar el uso de las herramientas de diseño, apuntando a facilitar y masificar el desarrollo de sistemas computacionales especializados para implementar y evaluar algoritmos de MPC en sistemas con dinámicas rápidas, con tiempos de muestreo en el orden de fracciones de segundo y con requerimientos de tiempo real.

1.4. Organización del Informe

Este trabajo tiene un enfoque exploratorio, y los entregables principales consisten en códigos y documentación asociada para su ejecución en plataformas embebidas. Estos códigos junto con detalles técnicos se encuentran disponibles en un repositorio [10] ya probado y validado por el profesor referente y terceras personas.

El resto de este informe sirve como complemento al entregable principal, y se limita a entregar información complementaria sobre aspectos de diseño y resultados que no están disponibles en el repositorio [10].

Los siguientes capítulos se estructuran de la siguiente manera:

- El **Capítulo 2** describe de forma matemática los algoritmos MPC y PDIP para poder entender cuáles son las operaciones que se van a implementar en código. El objetivo de este capítulo no es explicar teóricamente cómo es que funcionan MPC y PDIP sino mostrar cómo es que funciona para que luego sea más fácil leer y entender los códigos. También se describen los aspectos más importantes de las implementaciones de MPC y PDIP en código MATLAB
- El **Capítulo 3** describe los aspectos más importantes de las implementaciones de MPC y PDIP en código C++ , tanto para CPU y HLS.
- El **Capítulo 4** presenta la evaluación final de las implementaciones de MPC y se comparan los tiempos de ejecución de estas.
- El **Capítulo 5** resume las principales conclusiones de este trabajo y entrega las pautas para los trabajos futuros.

2 Antecedentes

Este capítulo presenta conceptos y terminología relevante utilizada en la formulación del problema de control usando MPC. La descripción matemática del problema se restringe a un nivel general, el cual resulta suficiente para dar un entendimiento general para caracterizar las estructuras de datos y flujo de procesamiento del algoritmo implementado en el código MATLAB que sirve de referencia para este trabajo. Los detalles teóricos sobre los métodos de control y optimización asociados pueden ser consultados en múltiples referencias, como por ejemplo: [12].

2.1. Fundamentos de MPC

MPC opera en base a cuatro componentes principales: (i) un modelo matemático del sistema a controlar; (ii) un horizonte de control de N muestras; (iii) un conjunto de restricciones en estados internos, salidas y entradas de control, aunque no necesariamente todas al mismo tiempo; y (iv) un funcional de costo que penaliza las desviaciones de las entradas y los estados de sus valores deseados. En cada instante de muestreo, MPC primero usa el modelo del sistema y el estado medido u observado para predecir el comportamiento futuro del sistema a lo largo del horizonte de predicción, y usa estas predicciones para encontrar la secuencia de acciones de control que minimizan el funcional de costo mientras se respetan las restricciones. Luego de calcular la secuencia óptima de entradas, se aplica el primer elemento de esta secuencia, y se espera al siguiente intervalo de control para medir o estimar los nuevos estados, repitiendo el proceso de optimización con un horizonte de predicción desplazado (esta estrategia se conoce como *receding horizon*). MPC permite implementar objetivos de control complejos, como por ejemplo, "*la energía generada por la turbina debe mantenerse lo más cercana posible a su valor de referencia, evitando que la temperatura de la caldera exceda 1200 grados y consumiendo la menor cantidad de combustible posible*" [13].

Dadas las condiciones de que el sistema a controlar es lineal (o aproximadamente lineal en un rango de operación acotado), se define una función de costo cuadrática, y como las restricciones sobre las entradas y estados del sistema son lineales (por ejemplo, restricciones de caja), entonces se dice que el problema de MPC es lineal. La ventaja de un problema MPC lineal es que el problema de optimización a resolver en cada instante de muestreo se puede representar como la formulación canónica de un problema QP con restricciones de igualdad y desigualdad, la cual tiene la forma:

$$
\begin{aligned}
\text{mínJ}(\xi) &= \frac{1}{2}\xi^T H \xi + h^T \xi \\
\text{sujeto a} & \\
A\xi &= b; \\
C\xi &\leq d
\end{aligned}
\tag{2.1}
$$

donde H tiene que ser semidefinida positiva para que el problema sea convexo y, por lo tanto, sea posible encontrar un mínimo global [14].

El resto de este capítulo describe las transformaciones aplicadas para representar el problema de control como una formulación del problema QP, la cual debe resolverse para cada instante de muestreo. La representación del problema de control mediante una forma canónica QP presenta la ventaja de que se puede resolver por medio de distintos solvers disponibles, ya sea comerciales como académicos de código abierto.

En este documento las dimensiones de las matrices están dadas de la forma $filas \times columnas$. Si solo se indica una dimensión significa que es vector columna.

2.1.1. Definición del problema de control

Considérese el siguiente sistema lineal discretizado e invariante en el tiempo:

$$
\begin{aligned}
x_{k+1} &= Ax_k + Bu_k \\
y_k &= Cx_k
\end{aligned}
\tag{2.2}
$$

donde $x_k \in \mathbb{R}^n$ representa el valor de los n estados del sistema en el instante de muestreo k, e $y_k \in \mathbb{R}^p$ y $u_k \in \mathbb{R}^m$ son los correspondientes vectores de p salidas y m entradas de control, respectivamente. Además, $A \in \mathbb{R}^{n\times n}$, $B \in \mathbb{R}^{n\times m}$ y $C \in \mathbb{R}^{p\times n}$ representan las dinámicas del sistema. La matriz A describe la influencia del estado actual en el siguiente estado, B describe la influencia de la entrada actual en el siguiente estado y C describe como es que el estado afecta la salida. Inicialmente asumimos que todos los estados del sistema son medibles para el instante k.

Se requiere diseñar un controlador que sea capaz de mover la salida del sistema a un valor de referencia. Por tanto, se define el vector de error $e_k \in \mathbb{R}^p$ como la diferencia entre la referencia $r_k \in \mathbb{R}^p$ y la salida y_k:

$$
e_k = r_k - y_k \tag{2.3}
$$

Si se considera que $r_k = 0$ (se busca llevar la salida al origen), entonces obtenemos $e_k = -y_k$. Luego, para penalizar las desviaciones del punto de equilibrio, convenientemente se define una función de costo asociada al cuadrado del error:

$$
J_k = y_k^T y_k \tag{2.4}
$$

Sustituyendo la expresión de y_k en (2.2) en (2.4), se obtiene:

$$J_k = x_k^T \underbrace{C^T C}_{\Omega} x_k = x_k^T \Omega x_k \tag{2.5}$$

donde $\Omega \in \mathbb{R}^{n\times n}$ es una matriz de pesos que refleja la importancia relativa de cada estado en la determinación de la salida del sistema. Adicionalmente, al considerar aspectos prácticos como el desgaste de los actuadores, restricciones físicas de las componentes del sistema, la seguridad de operación, el consumo energético, etc., es conveniente agregar un término cuadrático que penalice cambios drásticos en las entradas de control u_k, quedando la función de costo como:

$$J_k = x_k^T \Omega x_k + u_k^T \Gamma u_k \tag{2.6}$$

donde la matriz $\Gamma \in \mathbb{R}^{m\times m}$ representa factores de penalización definidas a tiempo de diseño. Para el caso de interés de múltiples entradas, podemos considerar por simplicidad una matriz diagonal $\Gamma = Iq$, donde $I \in \mathbb{R}^{m\times m}$ es la matriz identidad y q es un escalar que pondera la entrada.

Dado que el control predictivo requiere calcular el costo obtenido para las futuras entradas dentro del horizonte de predicción, se generaliza el funcional de costo como la suma de los costos individuales dentro del horizonte de predicción como:

$$J(x_k, u_k) = \sum_{k=0}^{N-1} \left(x_k^T \Omega x_k + u_k^T \Gamma u_k \right) + x_N^T \Omega_N x_N \tag{2.7}$$

donde el término $x_N^T \Omega_N x_N$ representa un estado terminal con su correspondiente matriz de penalización $\Omega_N \in \mathbb{R}^{n\times n}$, el cual se introduce para aproximar el comportamiento de un lazo MPC con horizonte infinito (lo cual no se puede implementar en la práctica) mediante un lazo con horizonte finito. En general, la expresión terminal influencia la calidad del control logrado, la estabilidad del sistema, y la complejidad numérica del problema. La correcta selección de esta expresión es un problema no trivial, el cual se resuelve por medio de heurísticas.

Para determinar la acción de control óptima a lo largo del horizonte, es necesario encontrar la secuencia de entradas $\vec{u}^* = [u_0^{*T} \quad u_1^{*T} \ldots \quad u_{N-1}^{*T}]^T \in \mathbb{R}^{(m\cdot N)}$ que minimiza el funcional de costo a lo largo del horizonte de predicción. Formalmente, y suponiendo que se dispone de una medición o estimación del vector de estados actual x_k, el problema QP se describe como:

$$\begin{aligned} \vec{u}^* &= \arg\min_{\vec{u}} J(x_k, u_k) \\ \text{sujeto a} & \\ x_0 &= x \\ x_{k+1} &= Ax_k + Bu_k \\ u^{\text{mín}} &\leq u_k \leq u^{\text{máx}} \\ x^{\text{mín}} &\leq x_k \leq x^{\text{máx}} \\ x_N^{\text{mín}} &\leq x_N \leq x_N^{\text{máx}} \end{aligned} \tag{2.8}$$

donde x_0 corresponde al estado inicial cuyo valor al instante k puede ser medido o estimado mediante un observador, $u^{\text{mín}}$ y $u^{\max} \in \mathbb{R}^m$ que especifican límites para los valores mínimos y máximos para cada entrada, $x^{\text{mín}}$ y $x^{\max} \in \mathbb{R}^n$ específican límites para los valores mínimos y máximos para cada estado, y $x_N^{\text{mín}}$ y $x_N^{\max} \in \mathbb{R}^n$ específican límites para los valores mínimos y máximos para cada estado terminal.

La formulación anterior plantea un problema de control en lazo abierto, ya que la secuencia de entradas óptimas calculadas al instante k no considera alteraciones o perturbaciones que puedan ocurrir al momento de ejecutar las acciones futuras. En el caso de ocurrir perturbaciones, la aplicación de las entradas precalculadas no generarán la trayectoria óptima en los estados y salidas, generando una degradación en el desempeño o incluso inestabilidad del sistema. Además, como se mencionó previamente, la solución encontrada es inherentemente subóptima debido a que el funcional de costo es solo una aproximación del problema para un horizonte infinito. Para tratar ambas limitaciones, se plantea el concepto de *receding horizon*, el cual sostiene que *" Un controlador subóptimo de horizonte infinito puede ser diseñado resolviendo repetidamente problemas de control óptimo de tiempo finito en una forma de horizonte en retroceso (...)"* [15].

Bajo el enfoque de *receding horizon*, se resuelve un problema de optimización de horizonte finito en cada tiempo de muestreo. Una vez determinada la secuencia de entradas $\vec{u}^*$, solo se aplica el primer elemento u_0^* al sistema, descartando el resto de valores de la secuencia óptima. Luego, en el siguiente intervalo de muestreo se determinan los nuevos estados obtenidos y se define un nuevo problema de optimización con un nuevo valor inicial, lo cual introduce un *feedback* indirecto, cerrando el lazo de control y evitando que las diferencias entre el estado medido y el predicho se propaguen a instantes futuros.

Los pasos para aplicar MPC en cada instante de muestreo se resumen a continuación:

1. Medir en el instante de muestreo k el estado de la planta, x_k.
2. Plantear el problema QP
3. Calcular el nuevo funcional de costo y resolver para determinar la secuencia óptima de entradas que lo minimizan, $\vec{u}^*$.
4. Aplicar en la entrada el primer elemento de la secuencia de control óptima, u_0^*.
5. Repetir para el siguiente instante de muestreo, $k+1$.

Matemáticamente, el problema QP a resolver en cada intervalo de control se puede plantear de distintas formas según la elección de las variables de decisión. Cuando se elige como única variable de decisión la entrada u_k, entonces se obtiene la forma denominada *Formulación Densa*, y si se eligen como variables de decisión tanto la entrada u_k como el estado x_k se obtiene la forma denominada *Formulación Sparse*. La selección de la forma de representar el problema QP normalmente influye significativamente en el costo computacional, en los requerimientos de memoria y tiempos de convergencia asociados al problema QP.

2.1.2. Formulación densa

Considerando como variable de decisión a la entrada del sistema, se definen los siguientes vectores:

$$\vec{u} = \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ \vdots \\ u_{N-1} \end{bmatrix} \qquad \vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_N \end{bmatrix} \tag{2.9}$$

donde $\vec{u} \in \mathbb{R}^{(m \cdot N)}$ y $\vec{x} \in \mathbb{R}^{(n \cdot N)}$. Con estas definiciones podemos reescribir el sistema (2.2) en términos de $\vec{u}$ y $\vec{x}$ como:

$$\vec{x} = \underbrace{\begin{bmatrix} A \\ A^2 \\ \vdots \\ \vdots \\ \vdots \\ A^N \end{bmatrix}}_{\mathscr{A}} x_0 + \underbrace{\begin{bmatrix} B & 0 & 0 & \dots & 0 & 0 \\ AB & B & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & 0 \\ A^{N-1}B & A^{N-2}B & \dots & \dots & AB & B \end{bmatrix}}_{\mathscr{O}} \vec{u} \qquad = \mathscr{A} x_0 + \mathscr{O} \vec{u} \tag{2.10}$$

donde $\mathscr{A} \in \mathbb{R}^{(n \cdot N) \times n}$ y $\mathscr{O} \in \mathbb{R}^{(n \cdot N) \times (m \cdot N)}$. También expandir el funcional de costo (2.7) para dejarlo en términos de x_0 y $\vec{u}$:

$$J(x_0, \vec{u}) = x_0^T \Omega x_0 + \vec{x}^T \underbrace{\begin{bmatrix} \Omega & 0 & \dots & 0 \\ 0 & \Omega & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & \dots & \Omega_N \end{bmatrix}}_{Q} \vec{x} + \vec{u}^T \underbrace{\begin{bmatrix} \Gamma & 0 & \dots & 0 \\ 0 & \Gamma & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & \dots & \Gamma \end{bmatrix}}_{R} \vec{u} = x_0^T \Omega x_0 + \vec{x}^T Q \vec{x} + \vec{u}^T R \vec{u} \tag{2.11}$$

donde $Q \in \mathbb{R}^{(n \cdot N) \times (n \cdot N)}$ y $R \in \mathbb{R}^{(m \cdot N) \times (m \cdot N)}$. Luego, concentrando (2.10) y (2.11) se obtiene:

$$\begin{aligned} J(x_0, \vec{u}) &= x_0^T \Omega x_0 + (\mathscr{A} x_0 + \mathscr{O} \vec{u})^T Q (\mathscr{A} x_0 + \mathscr{O} \vec{u}) + \vec{u}^T R \vec{u} \\ &= x_0^T \Omega x_0 + x_0^T \mathscr{A}^T Q \mathscr{A} x_0 + x_0^T \mathscr{A}^T Q \mathscr{O} \vec{u} + \vec{u}^T \mathscr{O}^T Q \mathscr{A} x_0 + \vec{u}^T \mathscr{O}^T Q \mathscr{O} \vec{u} + \vec{u}^T R \vec{u} \end{aligned} \tag{2.12}$$

donde los términos $x_0^T \mathscr{A}^T Q \mathscr{O} \vec{u}$ y $\vec{u}^T \mathscr{O}^T Q \mathscr{A} x_0$ son iguales ya que uno es el transpuesto del otro y cada uno es de dimensiones 1×1, por lo que se puede transponer el segundo término y sumarlo al primero, resultando:

$$J(x_0, \vec{u}) = x_0 (\Omega + \mathscr{A}^T Q \mathscr{A}) x_0 + \vec{u}^T (R + \mathscr{O}^T Q \mathscr{O}) \vec{u} + 2 x_0^T \mathscr{A}^T Q \mathscr{O} \vec{u} \tag{2.13}$$

Se define $H \in \mathbb{R}^{(m \cdot N) \times (m \cdot N)}$ y $h^T \in \mathbb{R}^{1 \times (m \cdot N)}$ como:

$$\frac{1}{2} H = R + \mathscr{O}^T Q \mathscr{O} \tag{2.14}$$

$$h^T = 2 x_0^T \mathscr{A}^T Q \mathscr{O} \tag{2.15}$$

Reemplazando (2.14) y (2.15) en (2.13) se obtiene el funcional de costo:

$$J(x_0, \vec{u}) = \frac{1}{2} \vec{u}^T H \vec{u} + h^T \vec{u} \tag{2.16}$$

Nótese que el termino $x_0 (\Omega_N + \mathscr{A}^T Q \mathscr{A}) x_0$ no aparece en el funcional de costo (2.16) ya que para cada instante de muestreo su valor es constante y por lo tanto no va a afectar la búsqueda de un $\vec{u}$ que minimice el costo.

Finalmente se reescriben el resto de las restricciones en términos de $\vec{u}$ y $\vec{x}$ como se describe a continuación:

a) Limitaciones en la señal de estado:

$$\begin{aligned} \vec{x}^{\text{mín}} &\leq \vec{x} \leq \vec{x}^{\text{máx}} \\ \vec{x}^{\text{mín}} &\leq (\mathscr{A} x_0 + \mathscr{O}\vec{u}) \leq \vec{x}^{\text{máx}} \\ \left(\vec{x}^{\text{mín}} - \mathscr{A} x_0\right) &\leq \mathscr{O}\vec{u} \leq \left(\vec{x}^{\text{máx}} - \mathscr{A} x_0\right) \end{aligned} \tag{2.17}$$

$$\begin{aligned} \mathscr{O}\vec{u} &\leq \left(\vec{x}^{\text{máx}} - \mathscr{A} x_0\right) \\ -\mathscr{O}\vec{u} &\leq -\left(\vec{x}^{\text{mín}} - \mathscr{A} x_0\right) \end{aligned} \tag{2.18}$$

donde

$$\vec{x}^{\text{mín}} = \begin{bmatrix} x^{\text{mín}} \\ \vdots \\ x^{\text{mín}} \end{bmatrix} \qquad \vec{x}^{\text{máx}} = \begin{bmatrix} x^{\text{máx}} \\ \vdots \\ x^{\text{máx}} \end{bmatrix} \tag{2.19}$$

tal que $\vec{x}^{\text{mín}}$ y $\vec{x}^{\text{max}} \in \mathbb{R}^{(n \cdot N)}$.

b) Limitaciones en la señal de entrada:

$$\begin{aligned} u^{\text{mín}} &\leq u_k \leq u^{\text{máx}} \\ \vec{u}^{\text{mín}} &\leq \vec{u} \leq \vec{u}^{\text{máx}} \end{aligned} \tag{2.20}$$

donde

$$\vec{u}^{\text{mín}} = \begin{bmatrix} u^{\text{mín}} \\ \vdots \\ u^{\text{mín}} \end{bmatrix} \qquad \vec{u}^{\text{máx}} = \begin{bmatrix} u^{\text{máx}} \\ \vdots \\ u^{\text{máx}} \end{bmatrix} \tag{2.21}$$

tal que $\vec{u}^{\text{mín}}$ y $\vec{u}^{\text{max}} \in \mathbb{R}^{m \cdot N}$.

y en forma matricial, todas las restricciones se pueden escribir como:

$$\underbrace{\begin{bmatrix} \mathscr{O} \\ -\mathscr{O} \end{bmatrix}}_{M} \vec{u} \leq \underbrace{\begin{bmatrix} \vec{x}^{\text{máx}} - \mathscr{A} x_0 \\ -\vec{x}^{\text{mín}} + \mathscr{A} x_0 \end{bmatrix}}_{c} \qquad \underbrace{\begin{bmatrix} u^{\text{mín}} \\ \vdots \\ u^{\text{mín}} \end{bmatrix}}_{a} \leq \vec{u} \leq \underbrace{\begin{bmatrix} u^{\text{máx}} \\ \vdots \\ u^{\text{máx}} \end{bmatrix}}_{b} \tag{2.22}$$

donde $M \in \mathbb{R}^{(2 \cdot n \cdot N) \times (m \cdot N)}$, $c \in \mathbb{R}^{(2 \cdot n \cdot N)}$, $a \in \mathbb{R}^{(m \cdot N)}$ y $b \in \mathbb{R}^{(m \cdot N)}$.

Nótese que las restricciones en (2.8) se transforman en restricciones lineales en $\vec{u}$, con lo cual se define el problema de optimización QP como :

$$\begin{aligned} \vec{u}^* &= \min_{\vec{u}} J(x_0, \vec{u}) \\ \text{sujeto a} \quad & \\ & M\vec{u} \leq c \\ & a \leq \vec{u} \leq b \end{aligned} \tag{2.23}$$

2.1.3. Seguimiento de referencia

Para el seguimiento de una referencia $r \neq 0$ en la salida se requiere que $y_\infty \to r$, donde y_∞ representa el valor de la salida en estado estacionario. Para determinar el valor de los estados y

entradas que permiten llevar la salida al valor de referencia, se reescribe la expresión en estado estacionario para el sistema en (2.2) como:

$$\begin{aligned} x_\infty &= Ax_\infty + Bu_\infty \\ y_\infty &= Cx_\infty \end{aligned} \tag{2.24}$$

y se despeja x_∞ y u_∞:

$$\underbrace{\begin{bmatrix} I-A & -B \\ C & 0 \end{bmatrix}}_{L} \begin{bmatrix} x_\infty \\ u_\infty \end{bmatrix} = \begin{bmatrix} 0 \\ r \end{bmatrix} \tag{2.25}$$

$$\begin{bmatrix} x_\infty \\ u_\infty \end{bmatrix} = L^{-1} \begin{bmatrix} 0 \\ r \end{bmatrix} \tag{2.26}$$

donde $L \in \mathbb{R}^{(n+p)\times(n+m)}$.

Y se realiza el siguiente cambio de variables:

$$\begin{aligned} \tilde{x} &= x - x_\infty \\ \tilde{u} &= u - u_\infty \end{aligned} \tag{2.27}$$

donde $\tilde{x}$ y $\tilde{u}$ representan la desviación de los estados y entradas de los valores que permiten llevar la salida a su valor de referencia en estado estacionario. Considerando que las variables de desviación satisfacen el mismo modelo dinámico que las variables originales, el funcional de costo en función de las variables de desviación queda dado por:

$$J_N(\tilde{x}_0, \vec{\tilde{u}}) = \frac{1}{2}\vec{\tilde{u}}^T H \vec{\tilde{u}} + h^T \vec{\tilde{u}} \tag{2.28}$$

donde:

$$\begin{aligned} \frac{1}{2}H &= R + \mathscr{O}^T Q \mathscr{O} \\ \tilde{h}^T &= 2\tilde{x}_0{}^T \mathscr{A}^T Q \mathscr{O} \end{aligned} \tag{2.29}$$

Para encontrar las restricciones de esta nueva función de costo hay que aplicar el cambio de variables (2.27) a las restricciones originales (2.22). Se llega a:

a) Limitaciones en las entradas:

$$\begin{aligned} \vec{u}^{\text{mín}} &\leq \vec{u} \leq \vec{u}^{\text{max}} \\ \vec{u}^{\text{mín}} &\leq \vec{\tilde{u}} + \vec{u}_\infty \leq \vec{u}^{\text{max}} \\ \vec{u}^{\text{mín}} - \vec{u}_\infty &\leq \vec{\tilde{u}} \leq \vec{u}^{\text{máx}} - \vec{u}_\infty \end{aligned} \tag{2.30}$$

b) Limitaciones en los estados:

$$\begin{aligned} \vec{x}^{\text{mín}} &\leq \vec{x} \leq \vec{x}^{\text{máx}} \\ \vec{x}^{\text{mín}} &\leq \vec{\tilde{x}} + \vec{x}_\infty \leq \vec{x}^{\text{max}} \\ \vec{x}^{\text{mín}} - \vec{x}_\infty &\leq \vec{\tilde{x}} \leq \vec{x}^{\text{máx}} - \vec{x}_\infty \\ \vec{x}^{\text{mín}} - \vec{x}_\infty &\leq \mathscr{A}\tilde{x}_0 + \mathscr{O}\vec{\tilde{u}} \leq \vec{x}^{\text{máx}} - \vec{x}_\infty \\ \vec{x}^{\text{mín}} - \vec{x}_\infty - \mathscr{A}\tilde{x}_0 &\leq \mathscr{O}\vec{u} \leq \vec{x}^{\text{máx}} - \vec{x}_\infty - \mathscr{A}\tilde{x}_0 \end{aligned} \tag{2.31}$$

$$\begin{aligned} \mathscr{O}\vec{u} &\leq \vec{x}^{\text{máx}} - \vec{x}_\infty - \mathscr{A}\tilde{x}_0 \\ -\mathscr{O}\vec{u} &\leq -\left(\vec{x}^{\text{mín}} - \vec{x}_\infty - \mathscr{A}\tilde{x}_0\right) \end{aligned} \tag{2.32}$$

Y finalmente se pueden representar las restricciones en estado estacionario:

$$\underbrace{\begin{bmatrix} \mathscr{O} \\ -\mathscr{O} \end{bmatrix}}_{\tilde{M}} \vec{\tilde{u}} \leq \underbrace{\begin{bmatrix} \vec{x}^{\text{máx}} - x_\infty - \mathscr{A}\tilde{x}_0 \\ -\vec{x}^{\text{mín}} + x_\infty + \mathscr{A}\tilde{x}_0 \end{bmatrix}}_{\tilde{c}} \qquad \underbrace{\begin{bmatrix} u^{\text{mín}} - u_\infty \\ \vdots \\ u^{\text{mín}} - u_\infty \end{bmatrix}}_{\tilde{a}} \leq \vec{\tilde{u}} \leq \underbrace{\begin{bmatrix} u^{\text{máx}} - u_\infty \\ \vdots \\ u^{\text{máx}} - u_\infty \end{bmatrix}}_{\tilde{b}} \tag{2.33}$$

donde $\tilde{M} \in \mathbb{R}^{(2\cdot n\cdot N)\times(m\cdot N)}$, $\tilde{c} \in \mathbb{R}^{(2\cdot n\cdot N)}$, $\tilde{a} \in \mathbb{R}^{(m\cdot N)}$ y $\tilde{b} \in \mathbb{R}^{(m\cdot N)}$.

2.1.4. Restricción de desigualdad

La restricción de caja en (2.33) se pasa a una de desigualdad para que todas queden en una ecuación.

Primero se separa en dos ecuaciones de desigualdad:

$$\begin{aligned} \tilde{a} \leq \vec{\tilde{u}} \leq \tilde{b} & \\ \tilde{a} \leq \vec{\tilde{u}} \qquad & \vec{\tilde{u}} \leq \tilde{b} \\ -\vec{\tilde{u}} \leq -\tilde{a} \qquad & \vec{\tilde{u}} \leq \tilde{b} \end{aligned} \tag{2.34}$$

Luego se juntan con las restricciones de desigualdad de (2.33) de la siguiente manera:

$$\begin{aligned} \begin{bmatrix} I \\ -I \end{bmatrix} \vec{\tilde{u}} &\leq \begin{bmatrix} \tilde{b} \\ -\tilde{a} \end{bmatrix} \\ \hat{M} = \begin{bmatrix} \tilde{M} \\ I \\ -I \end{bmatrix} \qquad \hat{c} &= \begin{bmatrix} \tilde{c} \\ \tilde{b} \\ -\tilde{a} \end{bmatrix} \end{aligned} \tag{2.35}$$

De esta manera $\vec{\tilde{u}}$ queda restringido de la siguiente manera:

$$\hat{M}\vec{\tilde{u}} \leq \hat{c} \tag{2.36}$$

donde $\hat{M} \in \mathbb{R}^{(2\cdot n\cdot N+2\cdot m\cdot N)\times(m\cdot N)}$ y $\hat{c} \in \mathbb{R}^{(2\cdot n\cdot N+2\cdot m\cdot N)}$

2.1.5. Planteamiento final del problema QP

Finalmente, el problema QP queda planteado, de forma densa, con seguimiento de referencia r distinta a 0 y con todas las restricciones en forma de desigualdad, como encontrar $\vec{\tilde{u}}^*$ tal que minimice la función de costo $J_N(\tilde{x}_0, \vec{\tilde{u}})$:

$$J_N(\tilde{x}_0, \vec{\tilde{u}}) = \frac{1}{2}\vec{\tilde{u}}^T H \vec{\tilde{u}} + \tilde{h}^T \vec{\tilde{u}} \tag{2.37}$$

donde:

$$\begin{aligned} \frac{1}{2}H &= R + \mathscr{O}^T Q \mathscr{O} \\ \tilde{h}^T &= 2\tilde{x}_0{}^T \mathscr{A}^T Q \mathscr{O} \end{aligned} \tag{2.38}$$

sujeto a:

$$\hat{M}\vec{\tilde{u}} \leq \hat{c} \tag{2.39}$$

2.1.6. Resolución del problema QP

Habiendo representado el problema de control como una forma canónica de un problema QP, es posible utilizar distintos solvers numéricos para obtener la señal de entrada $\vec{u^*}$ que minimiza el funcional de costo generado para cada instante de muestreo. En este trabajo nos enfocamos exclusivamente en el uso de PDIP para este propósito. La teoría y detalles de la formulación matemática del algoritmo PDIP para resolver problemas QP escapan del foco de este trabajo y se pueden encontrar en diversos libros de la materia (por ejemplo, [16]). Para este trabajo nos limitamos a estudiar la implementación de PDIP ya incluida en el código MPC usado como referencia.

Desde el punto de vista computacional, nos limitamos a describir que PDIP implementa un proceso iterativo que, a partir de una estimación inicial de la solución, aplica un proceso iterativo basado en el método de Newton para determinar una secuencia de pasos que permitan aproximarse a la solución real. En cada iteración de PDIP se aplica el método de Newton, el cual deriva en la necesidad de resolver un sistema de ecuaciones lineales cuyas dimensiones, en el caso de la formulación densa, dependen del horizonte de predicción N. A su vez, la resolución de sistemas lineales puede realizarse por métodos directos (e.g. Cholesky) o iterativos (e.g., MinRes, Conjugate Gradient). Independiente de la forma de resolución, es reconocido que la necesidad de resolver estos sistemas lineales representa el cuello de botella computacional de estos algoritmos de optimización, y por lo tanto, la implementación eficiente de solver lineales suele ser el objetivo de los trabajos que buscan acelerar la ejecución de problemas QP (y por ende MPC).

2.1.7. Resultado final

Una vez encontrado el vector $\vec{\tilde{u}}^*$ es necesario revertir el cambio de variables (2.27) que se hizo para seguir la referencia y así obtener las entradas que son apropiadas para aplicar en la planta. Esto se hace de la siguiente manera:

$$\vec{u}^* = \vec{\tilde{u}}^* + \vec{u_\infty} \tag{2.40}$$

El resultado final de MPC es el vector $\vec{u}^*$ que contiene las entradas para N instantes de tiempo que son necesarias para llevar la salida de la planta a la referencia deseada. Solo se aplica la primera entrada u_0^* a la planta y en el siguiente instante de muestreo se repite el proceso.

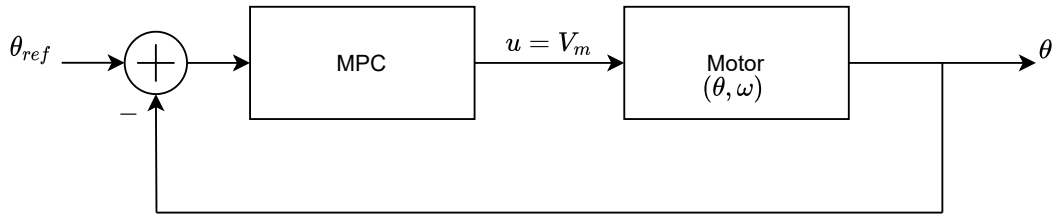

Figura 2.1: Lazo de control para motor.

2.2. Ejemplo MPC Aplicado a Motor DC

Como se mencionó en el Capítulo 1, para este trabajo consideramos como caso de estudio el lazo de control para un motor DC utilizando MPC en formulación densa, el cual ya ha sido modelado, estudiado y validado analítica y experimentalmente en trabajos previos. El contar con modelos y códigos de referencia permite abstraernos de los aspectos teóricos del modelamiento y teoría de control, y enfocarnos exclusivamente en los aspectos computacionales para caracterizar los tiempos de ejecución asociados al sistema. El resto de esta sección describe el modelo del motor y las características del lazo de control simulado en Matlab que proveen las referencias funcionales para el resto del este trabajo.

2.2.1. Modelo del motor y descripción del lazo de control.

El código de referencia considera un modelo discretizado del sistema QUBE Servo de Quanser que establece como variables de estado la velocidad angular ω y la posición angular θ, teniendo como salida la posición angular. Siguiendo la representación genérica entregada en la Ecuación (2.1), el sistema queda representado como:

$$x_{k+1} = Ax_k + Bu_k = \begin{bmatrix} e^{-Ts/\tau} & 0 \\ \tau(1 - e^{-Ts/\tau}) & 1 \end{bmatrix} \begin{bmatrix} \omega \\ \theta \end{bmatrix} + K \begin{bmatrix} 1 - e^{-Ts/\tau} \\ Ts + \tau(e^{-Ts/\tau} - 1) \end{bmatrix} u_k \tag{2.41}$$

$$y_k = Cx_k = \begin{bmatrix} 0 & 1 \end{bmatrix} x_k \tag{2.42}$$

en donde T_s corresponde al período de muestreo, y τ y K son constantes que dependen de las características del motor. En el código de referencia, las constantes se fijan en $T_s = 0{,}001$, $\tau = 0{,}0358$, y $K = 1{,}4$.

La Figura 2.1 muestra una representación del lazo de control diseñado para que la posición angular siga una referencia utilizando el voltaje de armadura V_m como entrada de control.

2.3. Código MATLAB

Al iniciar este trabajo se contaba con un código de referencia que implementaba un lazo de control MPC. Las funciones en el código original se reestructuraron para separar las operaciones que se realizan solo una vez de las que tienen que realizarse por cada instante de muestreo (las que dependen de x y r). El código también contiene instrucciones que permiten almacenar en

archivos los valores temporales de variables internas asociadas a cada paso dentro de un ciclo de control, con el fin de generar conjuntos de datos conteniendo entradas y resultados de las operaciones internas al lazo de control para facilitar el análisis detallado y validación numérica de los algoritmos implementados. Es importante mencionar que este código de referencia fue concebido con fines didácticos e ilustrativos, por lo que la estructura y organización del código favorece legibilidad y facilidad de uso por sobre desempeño computacional.

En el código existen matrices y vectores que tienen tamaño $2 \cdot n \cdot N + 2 \cdot m \cdot N$ en alguna de sus dimensiones. Por conveniencia se define la variable I_{aux} como:

$$I_{aux} = 2 \cdot n \cdot N + 2 \cdot m \cdot N \tag{2.43}$$

En el Algoritmo 1 se describe el pseudocódigo para MPC implementado en el código de referencia usado en este trabajo. El procesamiento parte con el cálculo de las matrices $\mathscr{A}$, $\mathscr{O}$ Ω, Q, H y $\hat{M}$, las cuales se consideran constantes en todo instante de muestreo (asumiendo sistema invariante en el tiempo y restricciones fijas) y por lo tanto se calculan solo una vez. Luego, en cada instante de muestreo, se toma una muestra del estado x_0 de la planta a controlar y la referencia a seguir r. Se hace un cambio de referencia y otras operaciones para llegar a las matrices $\tilde{h}$ y $\hat{c}$. Al tener las matrices H, $\tilde{h}$, $\hat{M}$ y $\hat{c}$ se procede a resolver, mediante PDIP, el problema QP que consiste en encontrar el valor de $\vec{\tilde{u}}^*$ que minimiza (2.37) cumpliendo las restricciones (2.39).

En términos de operaciones, PDIP implementa un proceso iterativo para converger a una solución aproximada, donde en cada iteración se arma un sistema lineal, se resuelve, y en base a la solución se calcula el siguiente paso para acercarse a la solución óptima. El orden del sistema lineal $A_k \times z_k = b_k$ viene dado por el horizonte de predicción N. La implementación de PDIP usada en esta memoria considera un número fijo de iteraciones que se decide a tiempo de diseño. Si bien también es posible definir una condición de término dinámica que itere hasta que la solución estimada alcance un valor estable, esta configuración no permite predecir en forma confiable el tiempo para alcanzar la solución, lo cual no es adecuado para sistemas de control en el que se cuenta con un tiempo de muestreo fijo.

Una vez finalizadas las iteraciones de PDIP y obtenido el valor estimado de $\vec{\tilde{u}}^*$, se hace el cambio de variables para volver del estado estacionario al actual y tener $\vec{u}^*$, que corresponde a las actuaciones para el horizonte de predicción N. El primer elemento del vector, u_0^*, se aplica a la planta y se repite el lazo MPC utilizando las nuevas mediciones/estimaciones de los estados.

2.3.1. Uso del código MATLAB.

A continuación se describen los principales pasos de configuración y resultados para este ejemplo. Los códigos completos, junto con información más detallada de su funcionamiento, se encuentran en la carpeta **GenerateSamplesMATLAB** del repositorio [10].

El código principal es `motorMpcReferenceTracking.m`, en el que se configuran los parámetros para este proyecto, se simula el motor y se guarda la información relevante que va a ser utilizada por los códigos C++ . Entre los parámetros que se pueden configurar se encuentra el vector *Ns* que contiene los horizontes con los que se va a realizar las simulaciones. Cambiando la variable `compare = true;` se puede comparar como queda el control MPC cuando se resuelve

Algorithm 1: Algoritmo de MPC utilizando PDIP

1 **function** Model Predictive Control
2 Calcular $\mathscr{A}, \mathscr{O}$ en base a A, B ▷ Ecuación: (2.10)
3 Calcular Q, R en base a Ω, Ω_N, Γ ▷ Ecuación: (2.11)
4 Calcular H en base a $R, \mathscr{O}, Q$ ▷ Ecuación: (2.14)
5 Calcular $\tilde{M}$ en base a $\mathscr{O}$ ▷ Ecuación: (2.33)
6 Calcular $\hat{M}$ en base a $\tilde{M}$ ▷ Ecuación: (2.35)
7 **for** *cada instante de muestreo* **do**
8 Obtener x_0 y r ▷ Muestra y referencia actual
9 Calcular x_∞ y u_∞ ▷ Llevar x y u a estado estacionario (2.25)
10 Calcular $\tilde{x}$ ▷ Calcular desviación del estado (2.27)
11 Calcular $\tilde{a}$ y $\tilde{b}$ ▷ Restricciones de caja en estado estacionario (2.33)
12 Calcular $\hat{c}$ ▷ Restricciones de desigualdad en estado estacionario (2.35)
13 Calcular $\tilde{h}$ ▷ Ecuación: (2.29)
14 $\vec{u}^* =$PDIP$(H, \tilde{h}, \hat{M}, \hat{c})$ ▷ Resuelve problema QP
15 Calcular $\vec{u}^*$ ▷ Volver del estado estacionario (2.40)
16 Aplicar u_0^* a la planta
17 **function** PDIP $(H, \tilde{h}, \hat{M}, \hat{c})$
18 $t_k = init$ ▷ Vector de tamaño N
19 $l_k = init2$ ▷ Vector de tamaño I_{aux}
20 $s_k = init3$ ▷ Vector de tamaño I_{aux}
21 **for** *k=0 to IterPDIP* **do**
22 $A_k = f_{A_k}(H, M, s_k, l_k)$ ▷ Sumas y multiplicaciones de matrices, resultado de tamaño $N \times N$
23 $b_k = f_{b_k}(H, h, M, c, t_k, s_k, l_k)$ ▷ Sumas y multiplicaciones de matrices, resultado de tamaño N
24 Resolver sistema $A_k \times z_k = b_k$ ▷ Mediante un linear solver
25 $\Delta lk = f_{\delta l_k}(l_k, s_k, M, z_k)$ ▷ Sumas y multiplicaciones de matrices, resultado de tamaño I_{aux}
26 $\Delta s_k = f_{\delta s_k}(l_k, s_k, M, \Delta l_k)$ ▷ Sumas y multiplicaciones de matrices, resultado de tamaño I_{aux}
27 $alp = f_{alp}(\Delta s_k, \Delta l_k)$ ▷ Busqueda del siguiente alp
28 $t_k = t_k + alp * z_k$ ▷ Actualizar t_k
29 $l_k = l_k + alp * \Delta l_k$ ▷ Actualizar l_k
30 $s_k = s_k + alp * \Delta s_k$ ▷ Actualizar s_k
31 **return** t_k

el problema QP con `PDIP` y con `quadprog()`(función implementada por MATLAB), también se compara con otro tipo de control llamado `Linear-Quadratic Regulator (LQR)` que no respeta las restricciones. Al final de la simulación se mostrarán gráficos de velocidad angular (Figura 2.2), posición angular (Figura 2.3) y voltaje de entrada (Figura 2.4).

El control por MPC está dividido en dos funciones, `setupMPC.m` y `iterMPC.m`, como se ve en el Algoritmo 2. Los cálculos que se hacen una sola vez se hacen en `setupMPC` y corresponde a las líneas 2 a la 6 del Algoritmo 1. La segunda función, `iterMPC`, es la que se utiliza por cada muestra y corresponde a las líneas 9 a la 15 del Algoritmo 1. En `iterMPC`, las únicas entradas que cambian iteración a iteración son x_0 y r.

Al ajustar las opciones `saveMat.MPC = true` , `saveMat.PDIP = true` o `saveMat.LS = true` se guarda la información necesaria para corroborar el correcto funcionamiento de `iterMPC`, `PDIP` y el *linear solver*, en un archivo .mat para tener como referencia y en un archivo binario para que sea leído por las implementaciones en C++ . La información guardada corresponde a parámetros, entradas y salidas de cada algoritmo.

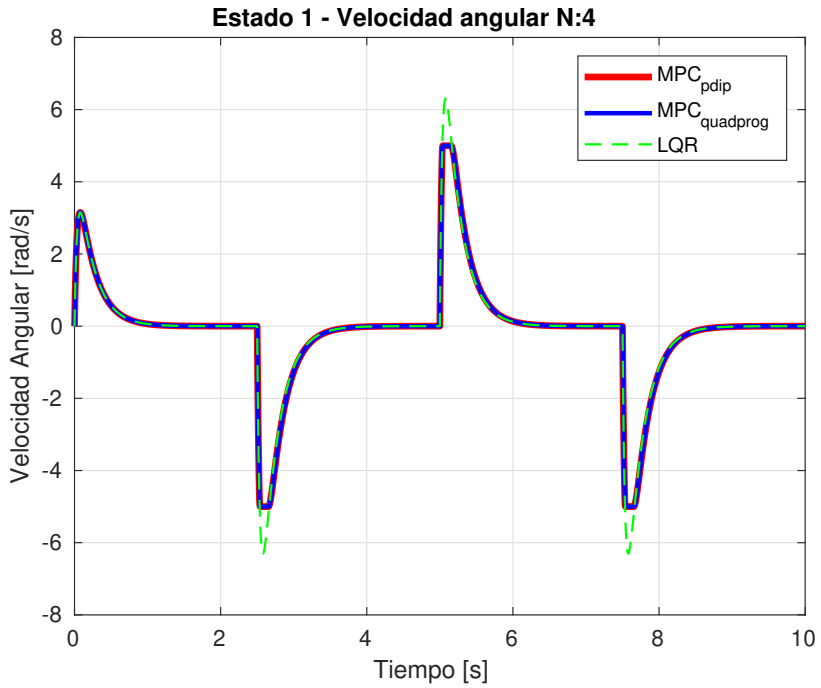

Figura 2.2: Comparación de velocidad angular ω para diferentes métodos de control.

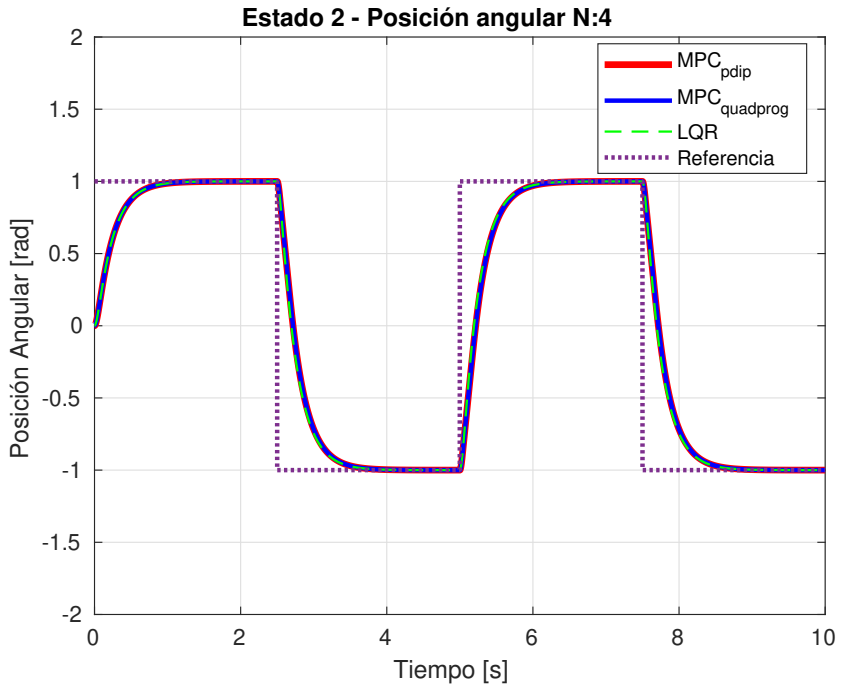

Figura 2.3: Comparación de posición angular θ para diferentes métodos de control y la referencia a seguir.

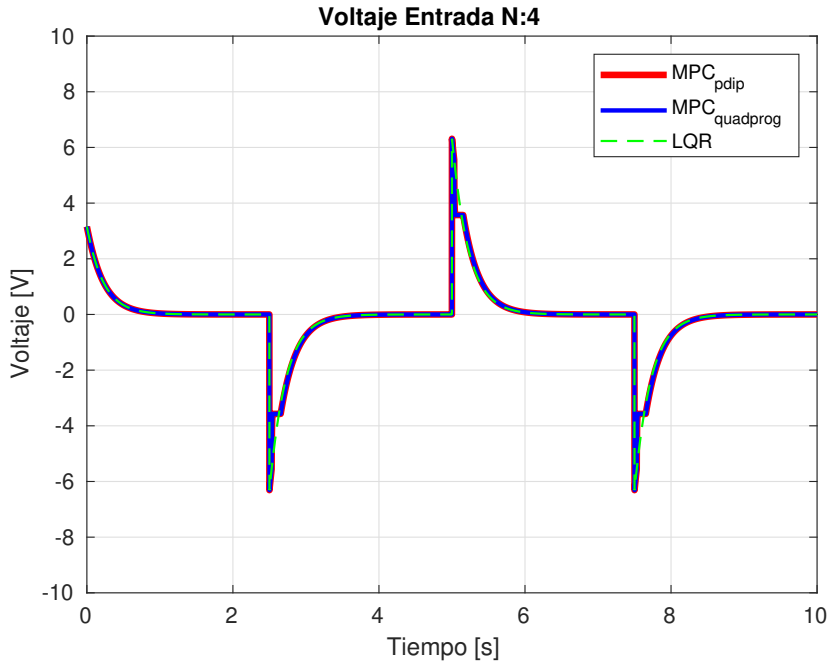

Figura 2.4: Comparación de el voltaje de entrada V_m para diferentes métodos de control.

Algorithm 2: MPC en MATLAB

```
function Model Predictive Control
    [𝒜, ℴ, Q, H, M̂] = setupMPC(A, B, Ω_N, Γ)        ▷ Cálculo de matrices constantes en MPC
    for cada instante de muestreo do
        Obtener x_0 y r                                ▷ Muestra y referencia actual
        u* = iterMPC(x_0, r, 𝒜, x^mín, x^máx, u^mín, u^máx, ℴ, Q, H, M̂)   ▷ Iteración de MPC
        Aplicar u*_0 a la planta
```

Tabla 2.1: Tamaños de matrices no constantes de entrada y salida en `iterMPC`, `PDIP` y `myChol`, para diferentes horizontes de predicción.

	`iterMPC`			`PDIP`			`myChol`		
N	x_0	r	u^*	h	c	t_k	A_k	B_k	z_k
2	2	1	2	2	12	2	2×2	2	2
3	2	1	3	3	18	3	3×3	3	3
4	2	1	4	4	24	4	4×4	4	4
6	2	1	6	6	36	6	6×6	6	6
8	2	1	8	8	48	8	8×8	8	8
16	2	1	16	16	96	16	16×16	16	16
32	2	1	32	32	192	32	32×32	32	32
64	2	1	64	64	384	64	64×64	64	64

2.3.2. Oportunidades de aceleración en hardware.

Al estar trabajando con una plataforma MPSoC que incluye un procesador multinúcleo y una FPGA, es posible implementar partes de MPC en FPGA y partes en CPU, por lo que es necesario examinar `iterMPC` y sus subalgoritmos para decidir dónde implementarlos. Para lo anterior, se presentan dos tablas que comparan `iterMPC`, `PDIP` y el *linear solver* `myChol` (una implementación propia de la descomposición de Cholesky). La primera tabla contiene información del tamaño de las matrices y la segunda los tiempos de ejecución.

En la Tabla 2.1 se muestran los tamaños para las matrices de entrada y salida, que cambian en cada iteración, para `iterMPC`, `PDIP` y `myChol`. Solo se muestran las matrices que cambian, ya que estas son las que se tendrían que transmitir entre la CPU y la FPGA. Donde x_0 y r tienen tamaño constante, los vectores u^*, h, t_k, B_k y z_k crecen linealmente de la forma N, el vector c crece linealmente de la forma $6N$, y finalmente la matriz A_k crece de forma cuadrática, $N \times N$.

Se realizan mediciones de tiempo con la herramienta de *profiling* de MATLAB para diferentes horizontes para identificar *bottlenecks*. Esta herramienta reporta el *total time* (tiempo gastado en la ejecución de la función, incluyendo el tiempo de todas las funciones hijas) y el *self time* (tiempo total ocupado en la ejecución del cuerpo de una función, excluyendo el tiempo gastado en llamadas a funciones hijas dentro del cuerpo principal) [1]. La Tabla 2.2 muestra ambos tiempos de ejecución para las funciones `iterMPC`, `PDIP` y `myChol` para 10000 iteraciones de iterMPC. Por cada llamada a `iterMPC` se hace una llamada a `PDIP`, y por cada llamada a `PDIP` se realizan 20 llamadas a `myChol`. Al *self time* de `iterMPC` se le agregan los tiempos de funciones que se escribieron por separado para hacer más legible el código de `iterMPC`.

Es importante considerar que los tiempos reportados en la Tabla 2.2 tienen como objetivo evidenciar como aumenta el tiempo de ejecución del lazo control y los subprocesos asociados a medida que aumenta el horizonte de predicción. Asimismo, como se menciona en la Sección 2.3, este código de referencia está orientado a fines didácticos y no fue diseñado con orientación a desempeño computacional, por lo cual estos tiempos de ejecución no deberían ser directamente comparados con los obtenidos en las implementaciones optimizadas que se describen en los siguientes capítulos.

[1] `https://www.mathworks.com/help/matlab/matlab_prog/profiling-for-improving-performance.html`

Tabla 2.2: Tiempos de ejecución para 10.000 iteraciones de `iterMPC` con diferentes horizontes de predicción.

	`iterMPC`		`PDIP`		`myChol`	
N	*Total* [s]	*Self* [s]	*Total* [s]	*Self* [s]	*Total* [s]	*Self* [s]
2	10,212	0,732	9,48	5,715	3,764	3,764
3	12,993	0,788	12,205	6,429	5,776	5,776
4	15,034	0,742	14,292	6,750	7,542	7,542
6	18,306	0,76	17,546	7,181	10,364	10,364
8	23,335	0,811	22,524	8,431	14,092	14,092
16	62,116	1,089	61,027	25,094	35,934	35,934
32	256,437	1,373	155,064	69,104	85,960	85,960
64	589,775	2,031	587,744	326,279	261,465	261,465

Los datos presentados en las Tablas 2.1 y 2.2 entregan información útil para decidir cuáles son las etapas del algoritmo que presentan el mayor tiempo de ejecución relativo y que son buenos candidatos para aceleración. En la Tabla 2.2 podemos observar que los *self times* de `PDIP` y de `myChol` son comparables y considerablemente mayores al *self time* de `iterMPC`, de uno a dos órdenes de magnitud más, por lo que son buenos candidatos para ser acelerados ya sea en hardware o software. Por otro lado, en la Tabla 2.1 se puede ver que `iterMPC` tiene entradas y salidas que son más pequeñas que las de `PDIP` o de `myChol`, por lo que al implementarse en FPGA se reduciría la cantidad de datos que se tienen que transmitir a la CPU y, por ende, el tiempo de comunicación. En base a las observaciones derivadas de estas evaluaciones, se decide implementar todo `iterMPC` en FPGA con el fin de reducir el *overhead* asociado a la comunicación entre CPU-FPGA y aprovechar de mejor manera las oportunidades de optimización que se ven en `PDIP` y `myChol`.

3 Implementaciones de IterMPC

Este capítulo sirve como complemento a los códigos desarrollados en C++ , los cuales están orientados a su ejecución en plataformas embebidas y son funcionalmente equivalentes al código de referencia en MATLAB descrito en el Capítulo 2.

Se presentan tres implementaciones de `iterMPC`, descrito en el Algoritmo 2:

- **MPC_CPP:** Se basa en el código MATLAB con algunas modificaciones para que sea más eficiente en términos de tiempo de procesamiento y sin uso de paralelismo explícito (sin hacer uso de *threads*).
- **MPC_OpenBLAS:** Se basa en el código MPC_CPP pero utilizando OpenBLAS para paralelizar operaciones básicas de álgebra lineal.
- **MPC_HLS:** Se basa en el código MPC_CPP pero utilizando pragmas para que, mediante Vitis HLS, se genere un código HDL que pueda implementarse en una FPGA.

El objetivo de este capítulo es presentar aspectos generales sobre las decisiones de diseño y otros aspectos relevantes que se consideraron al realizar la adaptación de los códigos. Los detalles de implementación y ejemplos de uso para cada caso se encuentran disponibles en el repositorio [10].

3.1. Implementación MPC_CPP

Esta implementación se basa en el código MATLAB referido en el Capítulo 2 manteniendo equivalencia en los resultados finales, sin necesariamente tener una relación uno a uno con las ecuaciones descritas en el Capítulo 2. Las optimizaciones con respecto al código MATLAB buscan eliminar las operaciones matemáticas redundantes y reducir el número de accesos a memoria. Este código no hace uso de paralelismo explícito, es decir, no se ocupan *threads* para paralelizar tareas, aunque sí es posible que el compilador aplique instrucciones del tipo "una instrucción, múltiples datos" (*single instruction, multiple data o SIMD*) [17] MPC_CPP apunta a proveer una base y referencia funcional para la posterior adaptación a MPC_HLS y MPC_OpenBLAS. Esta implementación se puede encontrar en la carpeta **MPC_CPP** del repositorio [10].

3.1.1. Implementación del algoritmo

El cálculo de A_k dentro del algoritmo `PDIP` (la línea 22 del Algoritmo 1) involucra una multiplicación de matrices donde una de ellas es una matriz diagonal, lo que se puede apreciar en el uso de la función `diag` en el código MATLAB de referencia. En esta implementación de `iterMPC` la operación es reemplazada por una multiplicación elemento a elemento entre cada fila de la primera matriz y el vector diagonal de la segunda matriz como se muestra en la Ecuación (3.1).

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \times \begin{bmatrix} j & 0 & 0 \\ 0 & k & 0 \\ 0 & 0 & l \end{bmatrix} = \begin{bmatrix} a \cdot j & b \cdot k & c \cdot l \\ d \cdot j & e \cdot k & f \cdot l \\ g \cdot j & h \cdot k & i \cdot l \end{bmatrix} \tag{3.1}$$

El cálculo del vector $\tilde{h}^T$, que viene de la Ecuación (2.29) o la línea 13 del Algoritmo 1, involucra un vector y tres matrices, y un único elemento que no es constante, $\tilde{x_0}^T$. Por lo que se precalcula $2\mathscr{A}^T Q \mathscr{O}$ y se entrega a la función `iterMPC` como un argumento de entrada con nombre `AcalQOcal` (C++ no permite nombres de variables que comiencen con números). Luego, al momento de calcular $\tilde{h}^T$ solo se debe realizar la multiplicación de la Ecuación (3.2).

$$\tilde{h}^T = \tilde{x_0}^T \cdot \texttt{AcalQOcal} \tag{3.2}$$

donde $\texttt{AcalQOcal} \in \mathbb{R}^{n \times (m \cdot N)}$.

En la Ecuación (2.26) la matriz L es constante y, además, luego de ser invertida se multiplica por un vector con ceros, salvo el último elemento que es r. Por esta razón es que se precalcula L^{-1} y, como se multiplica con un vector con solo un valor no nulo, se entrega como argumento a la función `iterMPC` solo la última columna de L^{-1} con el nombre `L_invLast`. De esta forma, al momento de calcular x_∞ y u_∞ se debe realizar la siguiente multiplicación:

$$\begin{bmatrix} x_\infty \\ u_\infty \end{bmatrix} = \texttt{L_invLast} \cdot r \tag{3.3}$$

donde $\texttt{L_invLast} \in \mathbb{R}^{(n+m)}$.

Tomando en cuenta que a la planta (en este caso el motor) únicamente se le aplica el primer elemento del vector $\vec{u}^*$, se decide que en esta implementación de `iterMPC` solo se aplique el cambio de variables de la Ecuación (2.40) (linea 15 del Algoritmo 1) al primer elemento del vector $\vec{\tilde{u}}^*$.

Finalmente, se decide implementar todos los arreglos con tamaño fijo por dos razones. En primer lugar, para que el compilador tenga más oportunidades de optimizar el código. En segundo lugar, ya que este código va a ser la base para la implementación MPC_HLS, que va a necesitar que los arreglos sean de tamaños fijos por limitaciones de HLS.

3.1.2. Parámetros

Los parámetros que todos los códigos de esta implementación tienen en común son definidos en `specs.h`. Se define el horizonte de predicción (N), las variables que especifican el tamaño del

sistema a controlar (m, n y p), cómo se representan los números flotantes (*float* o *double*) y el *linear solver* a utilizar. Estos parámetros son definidos en un único archivo para asegurar que todos los códigos concuerden y, si se necesita hacer un cambio, este se haga en un solo lugar. El *linear solver* a utilizar queda fijo ya que si se pudiese cambiar al momento de ejecución, sería necesario una sentencia de control *if-else*, pero estas podrían beneficiar (en términos de tiempo de ejecución) al primer *linear solver* y perjudicar al último.

3.2. Implementación MPC_OpenBLAS

MPC utiliza operaciones comunes con vectores y matrices (multiplicación de matrices, suma de vectores, producto punto, etc.), por lo que resulta provechoso hacer uso de una biblioteca especializada de álgebra lineal que las implemente. Así podemos enfocarnos en el algoritmo y no en cómo implementar eficientemente las operaciones matriciales y vectoriales.

La especificación *Basic Linear Algebra Subprograms* (BLAS) [18] define un conjunto de rutinas de bajo nivel que implementan operaciones de álgebra lineal. Existen tres niveles, donde cada nivel define un conjunto de operaciones diferentes: las del nivel uno son vector-vector, las del nivel dos son vector-matriz y finalmente las del nivel tres son matriz-matriz. Si bien BLAS es una especificación general, las implementaciones de BLAS son comúnmente optimizadas para procesadores o arquitecturas específicas como AOCL [19] para AMD, MKL [20] para Intel o ARM Performance Libraries [21] para ARM. Existen implementaciones de BLAS que se ejecutan en GPU, como cuBLAS [22], pero no se utilizarán en este trabajo.

Las diferentes implementaciones de BLAS explotan el paralelismo utilizando instrucciones SIMD o los *cores* y *threads* disponibles en el procesador objetivo. Las implementaciones BLAS también buscan minimizar los fallos de caché (*cache misses*), sabiendo cuales son los niveles y tamaños de memoria caché disponibles.

En esta memoria se escoge utilizar la biblioteca OpenBLAS [11] ya que se encuentra optimizada para distintos procesadores, solo basta indicar el procesador específico al compilar la biblioteca. Lo anterior permite que se pueda utilizar un mismo código fuente en un computador de escritorio con CPU Intel y en una tarjeta de desarrollo con procesador ARM, únicamente recompilando el código y la biblioteca OpenBLAS, siempre y cuando se utilice el compilador correspondiente a la arquitectura.

El código de la implementación MPC_OpenBLAS se puede encontrar en la carpeta del mismo nombre en el repositorio [10].

3.2.1. Implementación del algoritmo

Esta implementación se basa en MPC_CPP pero utilizando funciones de OpenBLAS para realizar operaciones como: multiplicaciones de matrices, producto punto, multiplicación de un vector por constante o suma de matrices. Por ejemplo, en una parte de la función f_{A_k} del Algoritmo 1 (línea 22) se suman las matrices A_k y H (ambas de tamaño $N \times N$) de la siguiente manera:

```
1     // Ak = Ak+H
2     for (int r=0; r<N; r++){
3         for(int c=0; c<N; c++){
```

```
4            Ak[r][c] = Ak[r][c]+H[r][c];
5        }
6    }
```

En MPC_OpenBLAS, la misma operación se realiza como una suma de dos vectores de largo $N \cdot N$, con la siguiente función:

```
1    // Ak = Ak+H
2    cblas_daxpy(N*N, 1, Ak, 1, H, 1);
```

Los detalles de cómo utilizar esta y otras funciones de BLAS pueden encontrarse en la documentación de Intel para BLAS [23] (se puede utilizar cualquier documentación BLAS ya que las llamadas a funciones son iguales entre las diferentes implementaciones de BLAS).

También se resuelve el sistema lineal (línea 24 del Algoritmo 1), mediante factorización LU con pivoteo utilizando la biblioteca LAPACK [24]. LAPACK viene incluida en OpenBLAS y es otra especificación de operaciones de álgebra lineal pero de más alto nivel que BLAS, para resolver sistemas de ecuaciones lineales, problemas de mínimos cuadrados lineales, problemas de valor propio y valor singular, y realizar una serie de tareas computacionales relacionadas.

3.3. Implementación MPC_HLS

En la Figura 3.1 se muestra cual es el rol de HLS en un flujo de trabajo en diseño de hardware. Para realizar HLS se necesita un código fuente, restricciones de diseño y directivas. El código fuente corresponde a la descripción del diseño en un lenguaje de alto nivel, C++ en este trabajo. Las restricciones de diseño son la frecuencia del reloj, incertidumbre del reloj y características técnicas de la FPGA a utilizar, como el número de registros, bloques de memoria, entre otros. Finalmente, las directivas, que toman forma de pragmas, ayudan a conducir el motor de optimización hacia los objetivos de rendimiento y arquitectura RTL deseadas. Con estos tres elementos Vitis HLS puede crear un código HDL (Verilog o VHDL). Si por alguna razón Vitis HLS no puede llevar a cabo una directiva se reporta un *warning* y se intenta sintetizar el código relajando o ignorando la directiva que causa problemas. El código HDL generado puede ser utilizado por otra herramienta, por ejemplo Vivado, para realizar los procesos de *synthesis* y *place* & *route* para obtener un *bitstream* o archivo de configuración para un chip FPGA.

En esta sección se describe en forma breve como se ocupan los pragmas y cuáles son los cambios que se hacen con respecto a MPC_CPP. El código de la implementación MPC_HLS se puede encontrar en la carpeta del mismo nombre en el repositorio [10].

3.3.1. Pragmas

Los pragmas son directivas que le entregan información adicional al compilador, en este caso Vitis HLS, para indicar cómo es que se deben implementar ciertas partes del código para optimizar, reducir latencia o mejorar el *throughput*.

Vitis HLS entrega soporte para distintos pragmas, los que se explican en detalle en la documentación [25]. Para efectos de este trabajo, se consideran cuatro de los más comunes, `#pragma HLS array_partition` , `#pragma HLS unroll` , `#pragma HLS pipeline` , y

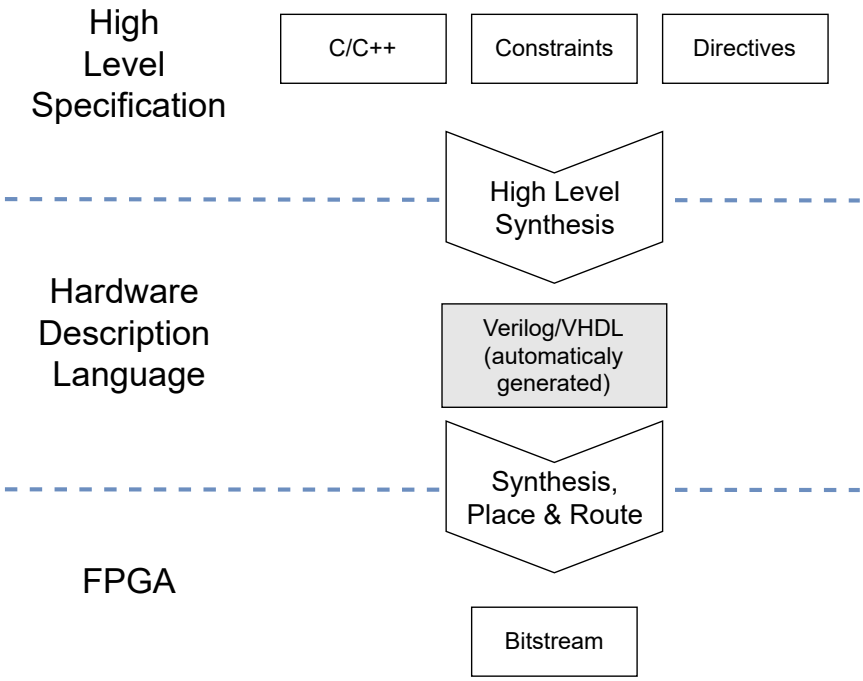

Figura 3.1: Flujo de trabajo utilizando High Level Synthesis.

`#pragma HLS inline` . La funcionalidad de los cuatro pragmas utilizados se describe brevemente en los siguientes párrafos.

El pragma `#pragma HLS array_partition` se utiliza para particionar un vector o matriz en varias memorias más pequeñas con el fin de aumentar la capacidad de lectura y escritura. Por ejemplo, si se tiene una matriz $A \in \mathbb{R}^{4\times4}$, se puede particionar de la siguiente manera:

```
int A[4][4];
#pragma HLS ARRAY_PARTITION variable=A dim=2
```

La variable *dim* indica que dimensión se particiona, en este caso los elementos de cada columna son los que se reparten entre las memorias. En la Figura 3.2 se muestra cómo es que queda guardada la matriz A. Al aplicar *array partition*, se utilizan cuatro memorias más pequeñas para almacenar A, con esto se logra que, en vez de leer A elemento a elemento, se lea fila por fila.

El pragma `#pragma HLS unroll` se utiliza para desenrollar un ciclo *for*. Desenrollar significa repetir el cuerpo del ciclo *for* en el código RTL, lo que permite que algunas o todas las iteraciones se realicen en paralelo (dependiendo de si se hace un *unroll* completo o no).

El siguiente ejemplo corresponde a un *unroll* completo:

```
for (int i=0; i<4; i++){
  #pragma HLS unroll
  c[i] = a[i] + b[i];
}
```

Sería equivalente a escribir un código Verilog:

```
assign c[0] = a[0] + b[0];
```

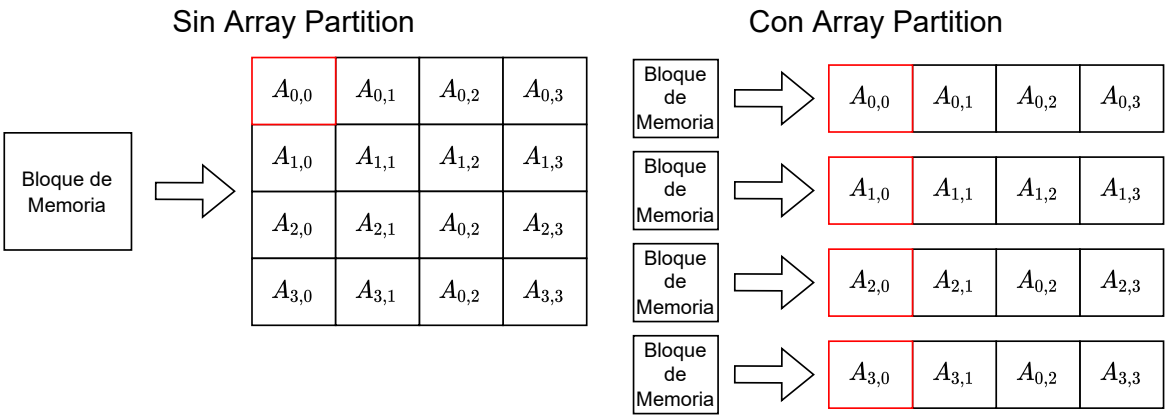

Figura 3.2: Comparación como se guarda A con y sin *array partition*.

```
assign c[1] = a[1] + b[1];
assign c[2] = a[2] + b[2];
assign c[3] = a[3] + b[3];
```

Con lo anterior se logra paralelizar las operaciones, es decir, en vez de realizar las sumas de manera secuencial una después de la otra, estas se realizan al mismo tiempo y en paralelo. Un *unroll* parcial de, por ejemplo, factor 2, sería realizar dos sumas en paralelo y llamarlas dos veces.

Hay que tener en cuenta que un *unroll* significa un mayor uso de recursos. En el caso anterior se requieren más sumadores y que las variables estén guardadas en memorias que permitan acceso paralelo a los datos (con *array partition*). No es posible aplicar *unroll* si es que el resultado de una iteración depende del resultado anterior y tampoco es posible aplicar un *unroll* completo si es que el número de iteraciones no es fijo. Lo ideal sería aplicar este pragma para todos los ciclos *for* que cumplan con los requerimientos, pero si existen numerosas operaciones dentro del ciclo es posible que los recursos necesarios para paralelizar no estén disponibles en la FPGA.

El pragma `#pragma HLS pipeline` permite crear *pipelines*. Un *pipeline* divide el procesamiento de una función o ciclo *for* en varias etapas que se ejecutan de forma concurrente, reduciendo el Intervalo de Iniciación (Initiation Interval, o II). El II corresponde el número de ciclos transcurridos entre la emisión de entradas a la función o iteraciones del ciclo *for*. Un *pipeline* no puede disminuir la latencia de una función o de una iteración de un ciclo *for*. Lo que logra el *pipeline* es un aumento del *throughput* de la función y una baja de la latencia total del ciclo *for*.

Un *pipeline* requiere que no exista dependencia entre cada llamada a la función o entre cada iteración del ciclo *for*. También aumenta la complejidad del diseño lo que se traduce en un mayor uso de recursos, particularmente *flip-flops* para sincronizar las etapas.

En la Figura 3.3 se compara la implementación con y sin *pipeline* de un *for* de tres ciclos con las operaciones: Lectura, Cálculo y Escritura.

```
for (int i=0; i<3; i++){
  Lectura;
  Calculo;
  Escritura;
}
```

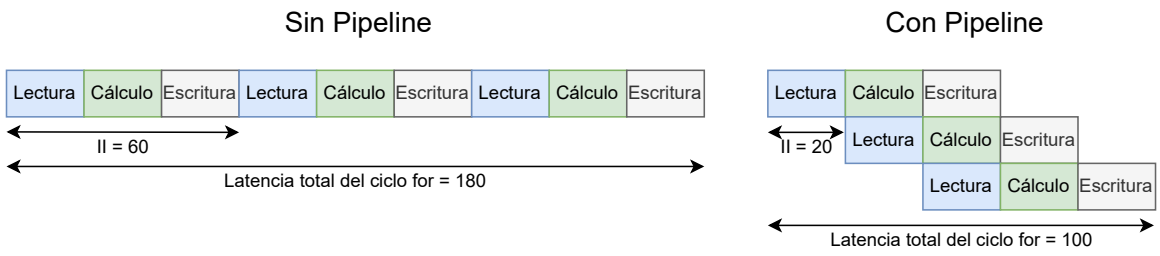

Figura 3.3: Visualización de un *pipeline* para un ciclo *for*

Sin *pipeline* se realizarían las operaciones en forma secuencial y, si cada operación dura 20 ciclos de reloj, la latencia total del ciclo *for* sería 180 y el II sería 60. Al aplicar *pipeline* la latencia total del ciclo *for* baja a 100 debido a que se pueden ejecutar concurrentemente las operaciones. También baja a 20 el II, por lo que cada 20 ciclos de reloj puede comenzar a trabajar una nueva iteración del ciclo *for*.

El pragma `#pragma HLS inline` permite que una función se disuelva en la función que la llama y que ya no aparezca como un nivel separado en jerarquía del RTL. Al aplicar *inline* a una función permite que las operaciones dentro ella se compartan y optimicen de manera más efectiva con las operaciones circundantes.

3.3.2. Implementación del algoritmo

El código de la implementación MPC_HLS se basa en MPC_CPP, con algunas modificaciones al código y utilizando pragmas para paralelizar ciclos *for*, indicar como se guardan matrices en memoria, entre otras cosas.

Por ejemplo, para el cálculo del vector $\tilde{x}$ (Ecuación 2.27 o línea 10 del Algoritmo 1) se particionan los arreglos correspondientes y se agrega un *unroll* en el ciclo *for*.

```
1 #pragma HLS ARRAY_PARTITION variable=x_tilde complete
2 #pragma HLS ARRAY_PARTITION variable=x complete
3 #pragma HLS ARRAY_PARTITION variable=x_inf complete
4   // MATLAB: x_tilde=x-x_inf;
5   set_x_tilde:
6   for (int i=0; i<N_SYS; i++){
7 #pragma HLS UNROLL
8     x_tilde[i] = x[i] - x_inf[i];
9   }
```

Para reducir la comunicación entre FPGA y CPU se decide que `iterMPC` retorne solo el primer elemento del vector $\vec{u}^*$, ya que el resto del vector no se ocupa para el control. De esta forma, desde la CPU se envían el estado x y la referencia r, y desde la FPGA se devuelve $\vec{u}_0^*$. La comunicación entre la CPU y la FPGA se realiza mediante la interfaz AXI [26].

Se decide optimizar la multiplicación de matrices con más atención debido a que en `PDIP` existen seis multiplicaciones de matrices y, si se utiliza un código C++ pensado para CPU, las operaciones no hacen uso del paralelismo que es posible lograr en una FPGA.

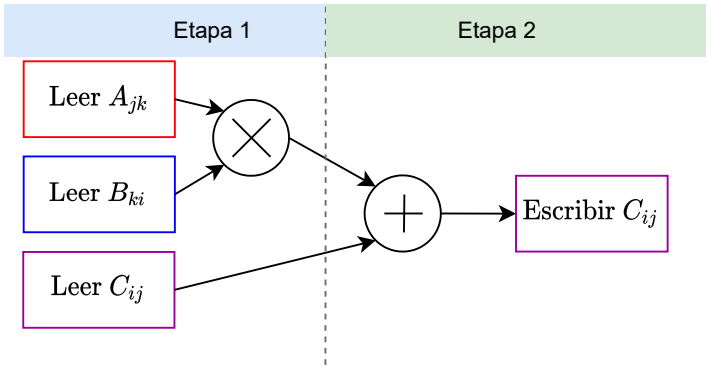

Figura 3.4: Multiplicación de matrices sin optimizaciones ni pragmas.

Multiplicación de matrices

El siguiente extracto de código muestra cómo se implementa la función `mmult`, que realiza una multiplicación de matrices pensada para CPU y sin pragmas, donde $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{n \times p}$ y $C \in \mathbb{R}^{m \times p}$:

```
void mmult (float *A, float *B, float *C, int m, int n, int p){

    for (int i = 0 ; i < m ; i++){
        for(int j = 0 ; j < p ; j++){
            C[i*p +j] = 0;
            for(int k = 0; k < n; k++){
                C[i*p +j] += A[i*n +k]*B[k*p +j];
            }
        }
    }
}
```

La Figura 3.4 ilustra en forma simplificada (existe lógica que controla este bloque que no se está mostrando) el hardware al que se llega mediante HLS: un multiplicador y un sumador en pipeline. A este bloque se le dan las entradas para $i = 0, j = 0, k = 0$, luego para $i = 0, j = 0, k = 1$ y así sucesivamente hasta llegar a $i = m, j = p, k = n$, por lo que la latencia crece con $m \cdot n \cdot p$. El orden temporal en que se ejecutan las etapas se encuentra en la Figura 3.5, donde $m = 2$, $n = 4$ y $p = 2$. Para reducir la latencia es necesario escribir un código C++ que utilice pragmas para describir el hardware al que se quiere llegar.

A continuación se describe la arquitectura a la que se busca llegar para multiplicar matrices. En la Figura 3.6 se muestra un ejemplo genérico de multiplicación de matrices $A \times B = C$, donde $A \in \mathbb{R}^{2\times4}$, $B \in \mathbb{R}^{4\times2}$ y $C \in \mathbb{R}^{2\times2}$. La matriz A está guardada en cuatro memorias, una para cada columna, lo cual permite acceder a todos los elementos de una fila en forma simultánea. Lo mismo ocurre para B, pero en este caso las memorias guardan las filas y el acceso en forma simultánea es a las columnas. No es necesario particionar la matriz C ya que no se necesita acceder a más de un elemento al mismo tiempo. En rojo, azul y morado se muestran los primeros accesos a memoria para el primer producto punto en la multiplicación de matrices.

En la Figura 3.7 se muestran las etapas del *pipeline* implementado, donde los accesos a memoria mostrados corresponden al primer producto punto. Se accede, a los elementos fila a fila en A y

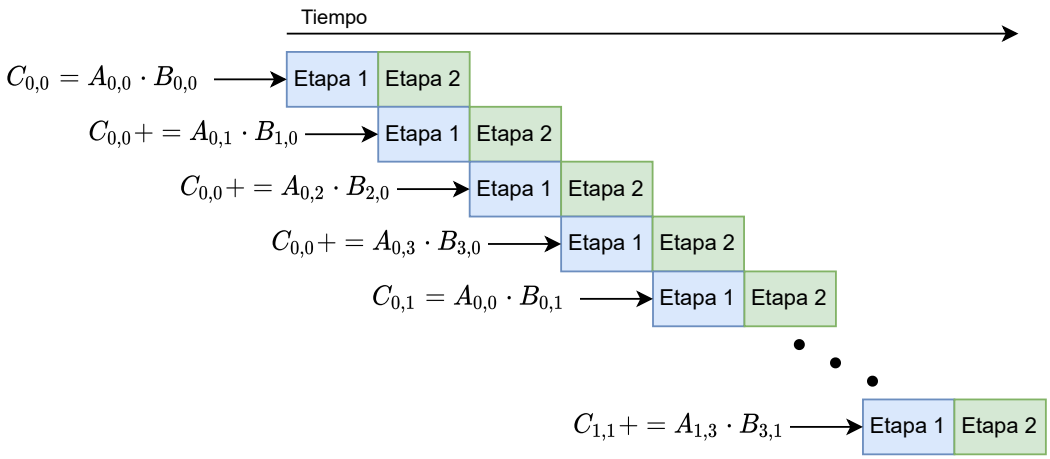

Figura 3.5: Orden temporal en que se ejecutan las etapas del *pipeline* para mmult.

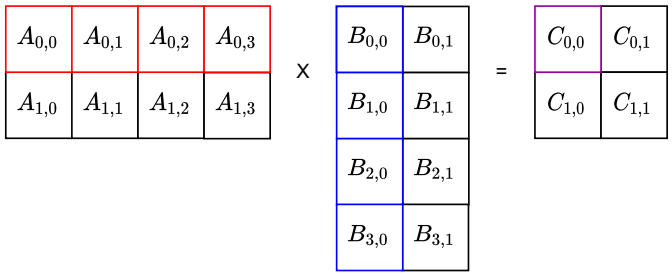

Figura 3.6: Ejemplo ilustrativo de multiplicación de matrices.

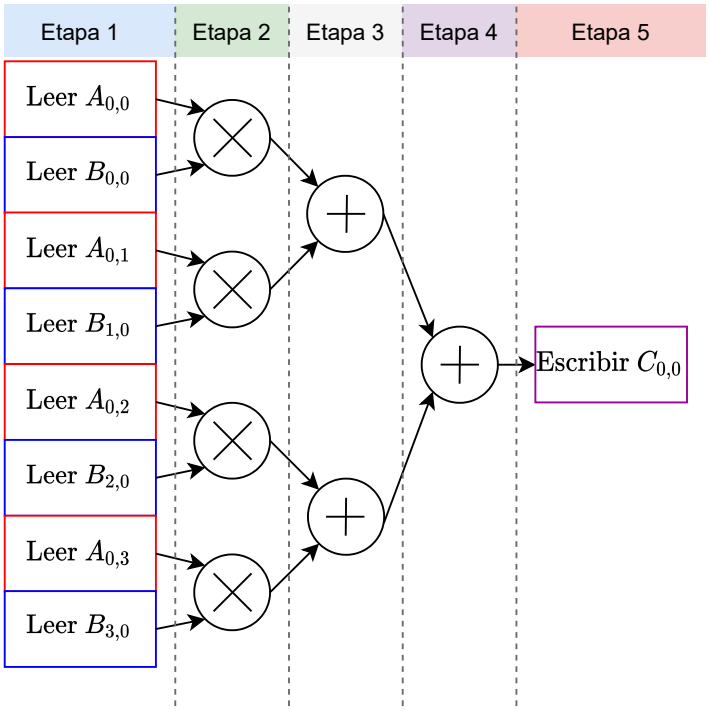

Figura 3.7: Etapas del *pipeline* para la multiplicación de matrices.

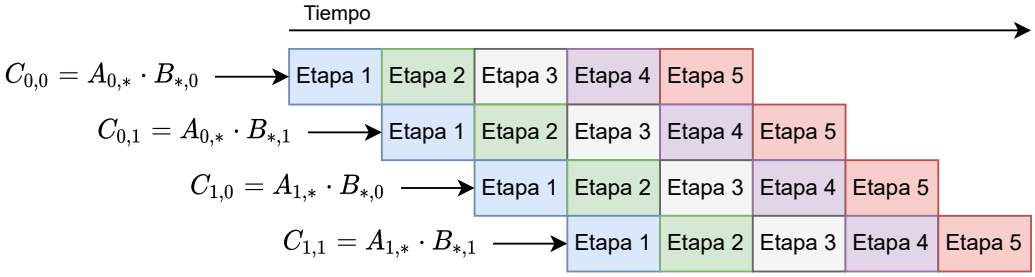

Figura 3.8: Orden temporal en que se ejecutan las etapas del *pipeline* para mmultHardware.

columna a columna en B (etapa 1). Al tener todos los elementos se realiza la multiplicación elemento a elemento en paralelo (etapa 2). Luego se suman los resultados mediante un *addertree* (etapas 3 y 4) y finalmente se guarda el resultado en C (etapa 5). En la Figura 3.8 se puede ver el orden temporal en que se ejecutan las etapas, además se indica a qué parte de la multiplicación corresponden (el $*$ indica que es toda la columna o fila correspondiente).

A continuación se describe cuáles son los pragmas utilizados para llegar a la arquitectura de la Figura 3.7.

Para lograr acceder a todos los elementos en paralelo en la etapa 1 se utiliza el pragma `#pragma HLS array_partition` . Luego con el pragma `#pragma HLS pipeline` se crea el *pipeline*. No es necesario agregar el pragma `#pragma HLS unroll` para realizar las multiplicaciones en paralelo ya que el pragma `#pragma HLS pipeline` desenrolla automáticamente lo que se encuentra más abajo en la jerarquía [27].

Si es que se están utilizando números enteros basta el pragma `#pragma HLS unroll` para

generar el *addertree* de las etapas 3 y 4. Sin embargo, esto no ocurre con punto flotante. Al trabajar con punto flotante el orden de las operaciones (sumas, multiplicaciones, etc.) afecta el resultado final, por lo que al reordenar las operaciones para implementar un *addertree* se está cambiando el resultado. En este caso el compilador opta por preservar el orden de las operaciones para mantener el resultado, sumando los elementos secuencialmente. Para poder implementar un *addertree* con punto flotante existen dos opciones: indicarle al compilador que no respete el orden de las operaciones (asumiendo que va a haber una diferencia de resultados entre código C++ y la implementación en hardware) o implementar un *addertree* en C++ . En este trabajo se ocupa la segunda opción para no tener una diferencia entre los resultados.

Debido a limitaciones de Vitis HLS, una función debe tener argumentos de tamaño fijo, por lo que para cada par de tamaños de multiplicación se define una función. La consecuencia de lo anterior es que se tengan varios módulos multiplicadores provocando un aumento en el uso de recursos. En razón de evitar esto se utiliza el pragma `#pragma HLS inline` para que no sea un módulo separado, sino parte del modulo que los está instanciando. Así se logra que, de ser posible, se compartan los recursos y se tengan más oportunidades para optimizar.

Hay que notar que con este código y pragmas no necesariamente se llega a la arquitectura exacta que se está intentando describir. Vitis HLS intenta implementar los pragmas teniendo en consideración las restricciones físicas de la FPGA y la frecuencia de reloj que se tiene disponible. De no ser posible implementar un pragma este se relaja o se elimina (se indica con un *warning*). Por otro lado, las etapas del *pipeline* pueden no ser las mismas o no estar balanceadas de la misma manera que en la Figura 3.7, por ejemplo: las primeras dos multiplicaciones de la etapa 2 podrían ser parte de la etapa 1 y las últimas dos multiplicaciones ser parte de la etapa 3.

El siguiente extracto de código define la función `mmultHardware` que fue escrita apuntando al hardware de la Figura 3.7, donde $A \in \mathbb{R}^{2\times4}$, $B \in \mathbb{R}^{4\times2}$ y $C \in \mathbb{R}^{2\times2}$:

```
1  float adderTree (float input[4]){
2  #pragma HLS INLINE
3    for(int i=1; i<4; i = i*2){
4    #pragma HLS unroll
5      for(int j=0; j+i<(4); j=(j+i*2)){
6        input[j] = input[j] + input[j+i];
7      }
8    }
9    return input[0];
10 }
11
12 void mmultHardware (float A[2][4], float B[4][2], float C[2][2]){
13     #pragma HLS INLINE
14   #pragma HLS ARRAY_PARTITION variable=A complete dim=2
15   #pragma HLS ARRAY_PARTITION variable=B complete dim=1
16     for (int r = 0 ; r < 2 ; r++){      // rows of A
17         for(int c = 0 ; c < 2 ; c++){   // cols of B
18         #pragma HLS PIPELINE II=1
19             float multArray[4];
20             for(int k = 0; k < 4; k++){
21                 multArray[k] = A[r][k]*B[k][c];
22             }
23             C[r][c] = adderTree(multArray);
24         }
25   }
26 }
```

Tabla 3.1: Tabla comparativa entre las implementaciones de multiplicación de matrices.

Implementación	Latencia (ciclos de reloj)	BRAM	DSP	FF	LUT
mmult	63	0	5	685	666
mmultHardware	17	0	18	1799	1384

En la Tabla 3.1 se muestran las diferencias en la estimación de latencia y uso de recursos que provee Vitis HLS. Se comparan `mmult` y `mmultHardware` con matrices de tamaños: $A \in \mathbb{R}^{2\times 4}$, $B \in \mathbb{R}^{4\times 2}$ y $C \in \mathbb{R}^{2\times 2}$. La latencia baja considerablemente con `mmultHardware` debido a todas las consideraciones de diseño que se tomaron, sin embargo, es un diseño más grande que el de `mmult` debido al uso de recursos.

3.3.3. Flujo de trabajo con Vitis

Con el diseño listo en Vitis HLS se procede a exportar el código generado en un archivo *Xilinx Object* (XO) para ser utilizado en Vitis (Vitis y Vitis HLS son programas distintos). Vitis toma el archivo XO, una distribución de Linux y el código que se va a ejecutar en el procesador y automáticamente genera una imagen con el sistema operativo. La imagen se carga en la tarjeta de desarrollo y se puede ejecutar la implementación MPC_HLS haciendo uso de la FPGA.

El flujo de trabajo descrito genera el *bitstream* sin que el usuario tenga que, en un proyecto de Vivado, hacer manualmente las conexiones necesarias entre el procesador y el módulo implementado. Esto permite una velocidad de desarrollo más rápida pero con la desventaja de que ya no se tiene un control fino de los módulos que van en la FPGA. Existe otro flujo de trabajo que permite un control fino, el cual, por simplicidad, se decide omitir en este trabajo y no se espera que cambie significativamente los resultados obtenidos. Más detalles sobre este flujo se puede encontrar en la documentación provista por Xilinx [28].

4 Evaluación Experimental

En este capítulo se explican y presentan los experimentos realizados para medir el rendimiento de las diferentes implementaciones de `iterMPC`, que se muestra en el Algoritmo 2. En primer lugar, se comparan las implementaciones MPC_CPP y MPC_OpenBLAS, ambas ejecutadas en un procesador de escritorio y en un procesador embebido para comprobar si iterMPC se beneficia del paralelismo que explota OpenBLAS. En segundo lugar, se comparan las implementaciones MPC_OpenBLAS y MPC_HLS ejecutadas en el procesador embebido y en una FPGA, respectivamente, para comparar la implementación en hardware con una en CPU. En los experimentos se comparan los tiempos de ejecución de las implementaciones bajo los siguientes parámetros estadísticos: media aritmética, desviación estándar (SD), máximo y mínimo.

4.1. Metodología

Se realizan experimentos donde se simula el control de un motor eléctrico, descrito en la Sección 2.2.1, con las implementaciones MPC_CPP, MPC_OpenBLAS y MPC_HLS, bajo diferentes horizontes de predicción y con diferentes precisiones de punto flotante (*float* de 32 bits y *double* de 64 bits). Cada experimento consta de diez pruebas independientes configuradas con los mismos parámetros iniciales, donde cada prueba corresponde a una simulación de 10.000 iteraciones de control. Considerando una tasa de muestreo configurada en 1000 muestras por segundo, la simulación equivale a 10 segundos de la dinámica del sistema. Usualmente, las primeras muestras se consideran como *warmup*, asumiendo que al inicio hay mayor probabilidad de *cache misses* o fallas en el predictor de saltos (*branch predictor*) que pueden generar tiempos por iteración relativamente altos en comparación a las muestras siguientes. En este trabajo las primeras 200 muestras por prueba se consideran como *warmup* y no forman parte del análisis estadístico.

Utilizando el código `motorMpcRecerenceTracking.m`, se realizan simulaciones para distintos largos del horizonte de predicción, generándose para cada caso un archivo binario que contiene los parámetros, modelo matemático del motor y las matrices pre calculadas necesarias para la simulación y el control del motor. Para corroborar el correcto funcionamiento de las implementaciones de iterMPC, el archivo binario también incluye el estado inicial, la referencia a seguir y los estados a los que se llega con la implementación en MATLAB y que sirve de referencia para comparar el resto de las implementaciones. Después de cada iteración, se compara el estado al que se llegó con lo calculado en MATLAB. En todas las implementaciones evaluadas la diferencia entre estado al que se llegó y lo calculado en MATLAB es menor a 1e-4, lo que equivale a 0,0001 radianes para la posición y 0,0001 radianes/s para la velocidad angular.

Considerando que la diferencia entre los valores máximos y mínimos observados en la simulación son 5/-5 para la velocidad angular y 1/-1 para la posición angular, las diferencias observadas entre las distintas implementaciones resultan despreciables y se pueden asociar a la conversión entre tipos de dato *double* a *float*.

En el Algoritmo 3 se muestra cómo se realizan las mediciones de tiempo. Se parte leyendo del archivo binario las matrices y vectores constantes, las matrices que definen el sistema a controlar y el estado inicial. Por cada instante de muestreo se lee la referencia a seguir y el estado calculado en MATLAB, x_{MATLAB}. Se mide el tiempo de ejecución de `iterMPC` y, en el caso de MPC_HLS, se incluye el tiempo de envío de datos y recepción entre CPU-FPGA. Las mediciones de tiempo se realizan utilizando la clase *high_resolution_clock* de la biblioteca estándar Chrono [29]. Luego se aplica u_0^* al motor y se compara el estado al que se llega x con x_{MATLAB}. Todo el código se ejecuta en CPU, salvo en MPC_HLS, donde `iterMPC` se ejecuta en FPGA. Al final de la simulación se exportan los tiempos a un archivo csv para su posterior análisis.

Los experimentos se realizan utilizando dos plataformas de referencia, cuyas características principales se describen a continuación:

- **Bombe:** Computador de escritorio con procesador AMD Ryzen 7 1700X, a 3,4 GHz, con 8 núcleos (16 *threads*) y 64 Gb de RAM. Cuenta con el sistema operativo Ubunto Gnome.
- **ZCU104:** Tarjeta de desarrollo Zynq UltraScale+ MPSoC ZCU104. El procesador es un ARM Cortex-A53, a 1,2GHz, con 4 núcleos y 2 Gb de RAM. La FPGA contiene 228.780 *look up tables* (LUT), 101.632 *look up tables as memory* (LUTAsMem), 458.585 registros (REG), 312 block RAM (BRAM), 96 ultra RAM (URAM) y 1.728 *digital signal processors* (DSP). Cuenta con una distribución básica de Linux, sin interfaz gráfica, creada por Xilinx que tiene acceso a la FPGA.

En el resto del capítulo se hará referencia al nombre de cada plataforma cuando se reporte un resultado.

Algorithm 3: Estructura del código que mide los tiempos de ejecución.

```
1  function benchmarkMPC
2      [𝒜, x^mín, x^máx, u^mín, u^máx, ℴ, Q, H, M̂] = leer("data.bin")   ▷ Leer matrices constantes para iterMPC
3      [A, B, C]= leer("data.bin")                                      ▷ Leer matrices que definen el motor
4      x_0= leer("data.bin")                                            ▷ Leer estado inicial del motor
5      x = x_0
6      for cada instante de muestreo do
7          [r, x_MATLAB] = leer("data.bin")      ▷ Leer referencia actual y estado calculado en MATLAB
8          inicio =tic()
9          u_0^* = iterMPC(x, r, 𝒜, x^mín, x^máx, u^mín, u^máx, ℴ, Q, H, M̂)      ▷ Iteración de MPC
10         fin = toc()
11         tiempos.push(fin-inicio)                          ▷ Guardar tiempo de ejecución
12         x = A · x + B · u_0^*                             ▷ Aplicar u_0^* al motor
13         comparar(x, x_MATLAB)
14     tiempos.export(times.csv)                             ▷ Exportar los tiempos medidos a un .csv
```

4.2. Implementaciones MPC_CPP y MPC_OpenBLAS

En esta sección se compara el tiempo de procesamiento de MPC_CPP y MPC_OpenBLAS, que son ejecutadas en CPU, en las plataformas Bombe y ZCU104. El objetivo de este experimento es ver si la paralelización que aplica OpenBLAS resulta en una aceleración del algoritmo con respecto a MPC_CPP. El código C++ es el mismo para ambas plataformas, solo que compilados con su compilador correspondiente ya que son arquitecturas diferentes.

4.2.1. Caracterización de tiempos de ejecución de iterMPC

Las diez pruebas se realizan para cada combinación de los siguientes elementos:

- **Plataforma:** Bombe y ZCU104.
- **Implementación de iterMPC:** MPC_CPP y MPC_OpenBLAS.
- **Horizonte de predicción:** $N = [2, 3, 4, 6, 8, 16, 32, 64]$.
- **Tipo de punto flotante:** *float* y *double*.

El tiempo de ejecución de MPC_CPP en ZCU104 es de aproximadamente cuatro horas por prueba para $N = 32, 64$, por lo que en estos casos solo se realizan dos pruebas, y por lo tanto en la Tabla 4.2, los datos para estos horizontes no son estadísticamente significativos pero proveen una estimación gruesa para comparación.

La Tabla 4.1 resume las características de los tiempos de ejecución de `iterMPC` para los diferentes horizontes y tipos de precisión en Bombe. La aceleración al utilizar MPC_OpenBLAS se empieza a ver desde $N = 3$ y va desde 1,08 a 10,39 veces. Para $N = 2$ no hay aceleración, lo que se puede deber a que el horizonte de predicción es tan pequeño que el *overhead* de preparar y coordinar los *threads* es comparable al tiempo de procesamiento.

La Tabla 4.2 reporta los tiempos medios de `iterMPC` para los diferentes horizontes y tipos de precisión en ZCU104. Al igual que en Bombe, la aceleración al utilizar MPC_OpenBLAS se empieza a ver desde $N = 3$ y va desde 1,21 a 26,5 veces. Nuevamente, para $N = 2$ no hay aceleración, aunque la diferencia en tiempo de procesamiento es menor que en Bombe. A pesar de que las medias de la ZCU104 son más altas que en Bombe, las desviaciones estándar tienden a ser menores, lo que puede ser por la menor cantidad de procesos que corren en segundo plano o por la arquitectura del procesador.

En la Figura 4.1 se grafican los tiempos medios con respecto al horizonte de predicción. Se puede observar que en MPC_CPP hay poca diferencia en el tiempo de ejecución al utilizar *float* o *double*. En MPC_OpenBLAS si existe una diferencia, al utilizar *float* los tiempos bajan con respecto a *double* lo que demuestra que OpenBLAS toma en cuenta el tipo de dato al realizar las optimizaciones. Sucede lo mismo en ZCU104, como se ve en la Figura 4.2.

Si bien en Bombe es más rápido que en ZCU104, se puede observar en la Figura 4.3 que hay pruebas en las que el tiempo de procesamiento es considerablemente mayor a otras. En naranjo y verde se ven dos pruebas que parten alto y se mantienen así más allá del *warmup*. Esto se debe a que, como se está utilizando un sistema operativo de computador de escritorio, existen otros procesos que están corriendo en segundo plano.

Tabla 4.1: Análisis estadístico de los tiempos de ejecución, por iteración, de `iterMPC` en Bombe.

		MPC_CPP				MPC_OpenBLAS				
N	Tipo de dato	Media [μs]	SD [μs]	Máx [μs]	Mín [μs]	Media [μs]	SD [μs]	Máx [μs]	Mín [μs]	Aceleración [*veces*]
2	float	27,84	2,48	143,43	24,51	39,89	13,1	3373,74	34,89	0,70
2	double	32,09	2,07	68,04	28,55	42,53	7,68	200,65	38,13	0,75
3	float	58,41	2,46	87,25	54,67	54,07	8,82	105,5	47,99	1,08
3	double	61,57	2,01	90,73	56,52	55,72	27,39	8119,9	49,99	1,10
4	float	92,78	3,17	157,77	86,4	63,02	9,81	115,61	55,95	1,47
4	double	92,15	3,22	295,92	86,03	69,51	9,91	228,93	64,95	1,33
6	float	205,51	6,54	372,51	194,53	104,86	11,02	270,18	97,37	1,96
6	double	206,68	4,58	370,86	201	121,85	11,31	223,31	114,03	1,70
8	float	381,73	8,94	481,33	365,57	148,59	11,95	350,25	138,91	2,57
8	double	386,45	7,58	543,42	374,5	185,84	8,58	352,8	176,13	2,08
16	float	2140,54	28,48	2923,7	2071,01	716,24	100,44	1227,94	661,59	2,99
16	double	2155,09	29,38	2670,84	2087,57	735,05	12,07	1392,74	719,78	2,93
32	float	14372,91	125,7	15857,4	13982,6	3311,45	279,15	12972	2874,44	4,34
32	double	14526,93	144,13	19950,3	14056,5	4850,99	177,71	14089,9	4398,31	2,99
64	float	106222,44	624,28	117974	104543	10222,7	482,38	77386,9	9786,32	10,39
64	double	105866,2	984,88	116655	103562	16854,28	315,51	28476,5	16440,8	6,28

En algunos casos se puede observar que cuando la referencia cambia de signo, como en las muestras 2500, 5000 y 7500 de a la Figura 4.4, los tiempos de procesamiento suben ligeramente, lo cual indica que el tiempo de procesamiento presenta una dependencia de la referencia. Si se quisiese evaluar estas implementaciones para ser aplicadas con requerimientos de tiempo real, se debería buscar una referencia que permita realizar un *worst-case analysis* para determinar la máxima latencia para un ciclo de control. Esta latencia del peor caso determinará el período de control que permita que el sistema converja bajo todas las condiciones de operación evaluadas.

Tabla 4.2: Análisis estadístico de los tiempos de ejecución, por iteración, de `iterMPC` en ZCU104.

		MPC_CPP				MPC_OpenBLAS				
N	Tipo de dato	Media [μs]	SD [μs]	Máx [μs]	Mín [μs]	Media [μs]	SD [μs]	Máx [μs]	Mín [μs]	Aceleración [*veces*]
2	float	292,29	2,12	455,65	291,25	327,19	3,72	757,43	324,26	0,89
2	double	302,22	2,44	502,74	301,19	341,19	3,53	531,67	337,55	0,89
3	float	607,5	3,55	1139,02	605,59	463,26	4,32	939,44	458,85	1,31
3	double	623,59	3,04	821,78	621,47	514,72	5,12	1065,17	508,43	1,21
4	float	1073,02	3,67	1245,49	1070,02	544,79	4,82	985,55	539,74	1,97
4	double	1090,92	3,67	1262,24	1088,19	634,67	8,83	829,61	624,91	1,72
6	float	2605,66	4,43	2779,82	2599,68	921,27	9,97	1147,8	907,06	2,83
6	double	2638,24	4,42	2805,44	2632,86	1187,58	8,08	1357,39	1170,47	2,22
8	float	5152,29	5,19	5322,84	5146,77	1262,73	13,78	1452,79	1244,98	4,08
8	double	5197,32	5,19	5450,46	5191,69	1856,47	11,17	2038,89	1830,07	2,80
16	float	30411,33	12,64	30882,7	30384,8	4326,88	11,71	5611,96	4308,03	7,03
16	double	30594,9	12,74	30795,3	30571,5	7925,53	33,01	10044,5	7874,27	3,86
32	float	208037,34*	1319,97*	209531*	206657*	13965,87	56,36	16156,1	13862,4	14,90*
32	double	213711,63*	813,84*	216696*	212841*	23958,64	85,98	33444,4	23864,9	8,92*
64	float	1555821,08*	2769,65*	1560820*	1552760*	58711,19	229,21	79520,2	58251,8	26,50*
64	double	1842550,14*	505,84*	1846130*	1841710*	138467,36	866,46	168027	137116	13,31*

* Solo se realizan dos pruebas en vez de diez.

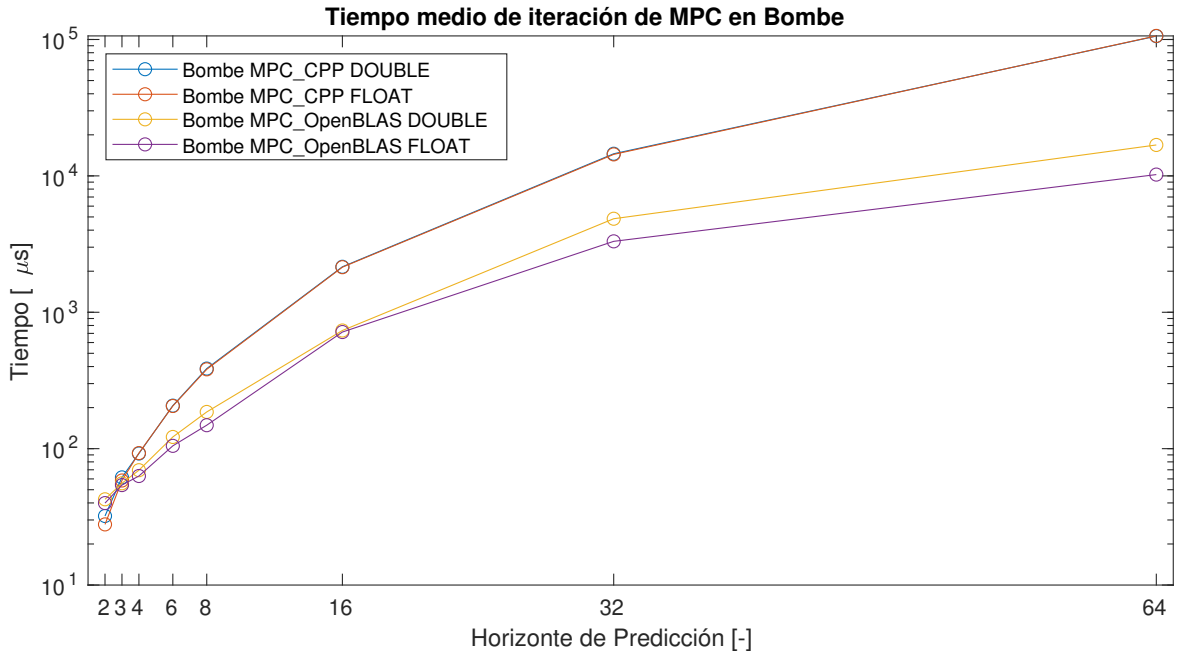

Figura 4.1: Tiempo medio de `iterMPC` en Bombe para $N = 2, 3, 4, 6, 8, 16, 32, 64$.

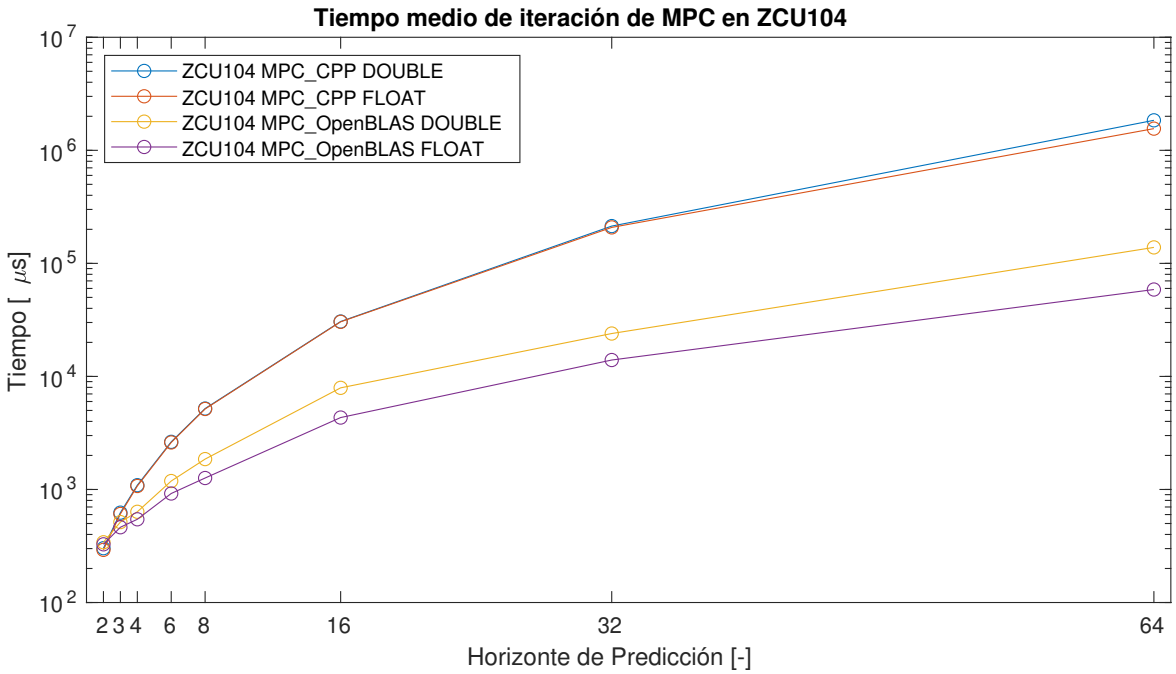

Figura 4.2: Tiempo medio de `iterMPC` en ZCU104 para $N = 2, 3, 4, 6, 8, 16, 32, 64$.

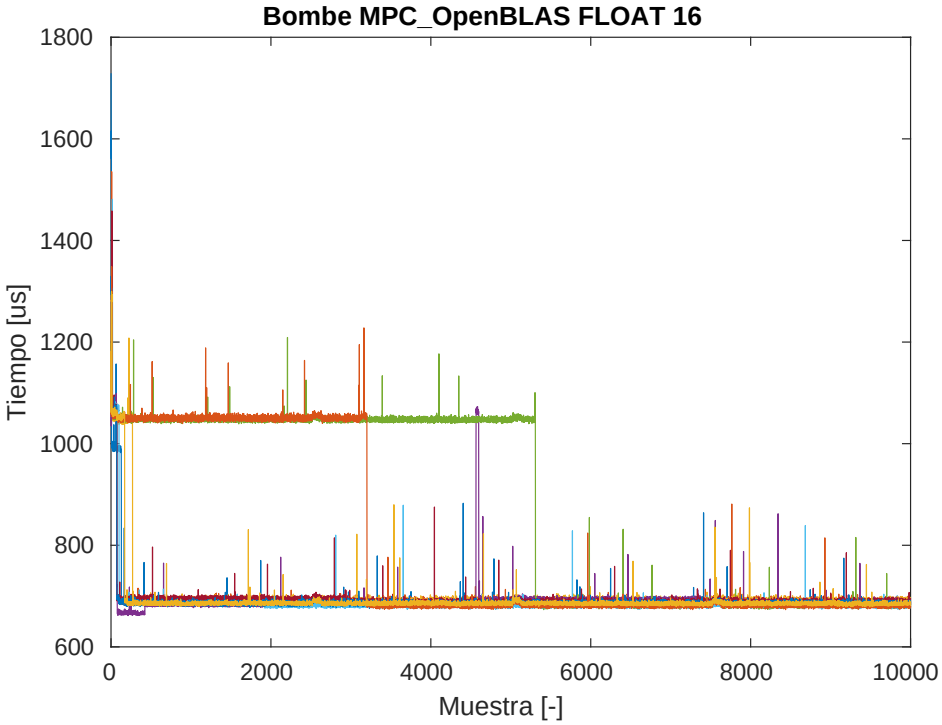

Figura 4.3: Tiempos de ejecución para diez pruebas en Bombe, con *float* y $N = 16$.

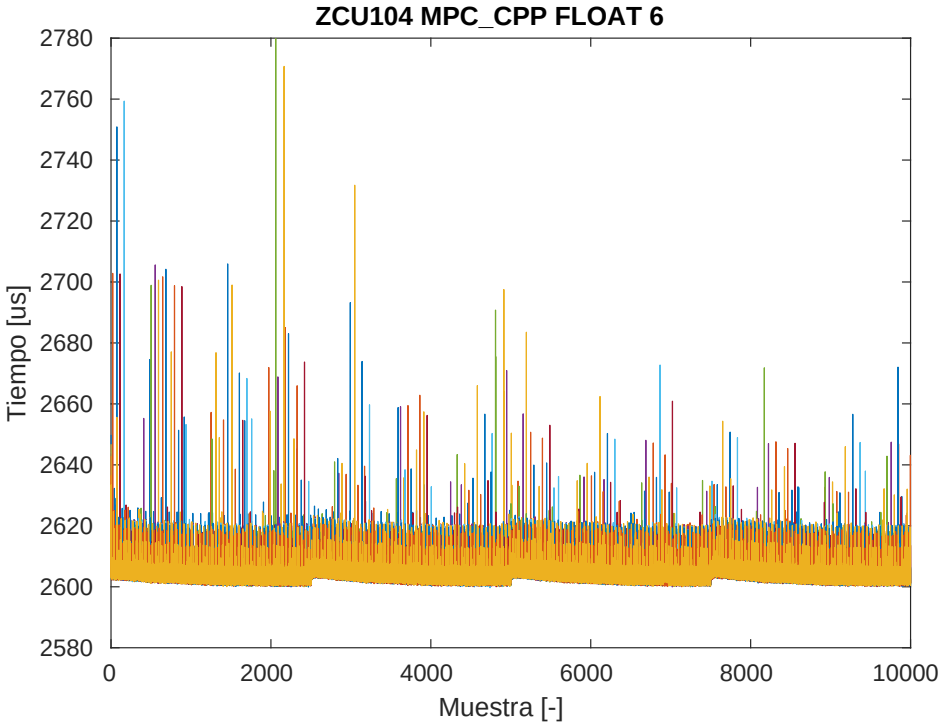

Figura 4.4: Tiempos de ejecución para diez pruebas en ZCU104, con *float* y $N = 6$.

4.3. Implementaciones MPC_OpenBLAS y MPC_HLS

En esta sección se presentan los resultados de la implementación MPC_HLS para compararlos con los resultados obtenidos con MPC_OpenBLAS, ya que fue la implementación más rápida en CPU embebida. Ambas implementaciones se ejecutan en ZCU104, que es una plataforma MPSoC que contiene un procesador ARM y una FPGA en el mismo chip, donde `iterMPC` se ejecuta en CPU para MPC_OpenBLAS y en FPGA para MPC_HLS. El objetivo de esta comparación es ver como se comparan los resultados obtenidos mediante HLS y los en CPU y obtener conclusiones respecto al rendimiento obtenido.

Los reportes de utilización de recursos para cada horizonte de predicción y tipo de dato se presentan en la Tabla 4.3. Para *double* el uso de recursos aumenta considerablemente con respecto a *float*, aproximadamente el triple de LUTs, el doble de REGs y el doble de DSPs. Lo anterior resulta en que no sea posible generar el *bitstream* para $N = 4, 6, 8$ debido a una alta congestión en una sección de la FPGA o necesidad de utilizar más recursos de los disponibles. El uso de recursos es lo que impide que se puedan implementar los horizontes $N = 16, 32, 64$.

Tabla 4.3: Uso de Recursos de MPC_HLS en FPGA.

		Recursos					
N	Tipo de Dato	LUT	LUTAsMem	REG	BRAM	URAM	DSP
2	float	11,07 %	1,13 %	6,53 %	2,24 %	0,00 %	3,47 %
2	double	31,77 %	2,17 %	19,66 %	3,21 %	0,00 %	7,64 %
3	float	17,59 %	1,57 %	9,29 %	1,28 %	0,00 %	5,21 %
3	double	49,18 %	3,18 %	28,96 %	2,24 %	0,00 %	11,46 %
4	float	23,39 %	1,69 %	13,58 %	3,53 %	0,00 %	6,94 %
4	double	65,63 %	2,88 %	32,50 %	3,85 %	0,00 %	15,28 %
6	float	28,82 %	4,89 %	15,33 %	4,17 %	0,00 %	10,42 %
6	double	86,93 %	8,86 %	39,72 %	7,05 %	0,00 %	22,92 %
8	float	37,38 %	4,36 %	20,36 %	21,47 %	0,00 %	13,89 %
8	double	116,76 %	12,36 %	51,52 %	39,74 %	0,00 %	30,56 %

4.3.1. Caracterización de tiempos de ejecución de MPC

Se realizan 10 pruebas para cada combinación de los siguientes elementos:

- **Plataforma:** ZCU104.
- **Implementación de MPC:** MPC_OpenBLAS y MPC_HLS.
- **Horizonte de predicción:** $N = [2, 3, 4, 6, 8]$.
- **Tipo de dato:** *float* y *double*.

Se puede observar en la Tabla 4.4 que la desviación estándar es mayor en MPC_HLS. Se sabe que en FPGA el tiempo de ejecución es constante (por características propias de una FPGA y cómo fue diseñada la implementación), por lo que la variación que aparece proviene de la comunicación entre CPU-FPGA. Bajo la misma consideración de que el tiempo de ejecución en FPGA es constante podemos concluir que el tiempo de procesamiento en la FPGA no puede ser mayor que el mínimo reportado en la Tabla 4.4.

Tomando en cuenta los tiempos medios de la Tabla 4.4, la aceleración obtenida es de 1,41 hasta 2,31 veces. Los tiempos medios se muestran en un gráfico en la Figura 4.5, pudiéndose observar que mientras más grande sea N, más es la diferencia de tiempo de procesamiento.

En la implementación MPC_HLS no existe el efecto que tiene el cambio de signo de la referencia como se aprecia en la Figura 4.6, en contraste con lo visto con MPC_CPP en la Figura 4.4, lo que muestra que el tiempo de ejecución en la FPGA es constante.

Tabla 4.4: Análisis estadístico de los tiempos de ejecución de MPC_OpenBLAS y MPC_HLS ejecutados en ZCU104.

		MPC_OpenBLAS				MPC_FPGA				
N	Tipo de dato	Media [μs]	SD [μs]	Máx [μs]	Mín [μs]	Media [μs]	SD [μs]	Máx [μs]	Mín [μs]	Aceleración [*veces*]
2	float	327,19	3,72	757,43	324,26	231,24	8,41	1763,03	214,54	1,41
	double	341,19	3,53	531,67	337,55	285,81	7,53	1376,15	266,6	1,19
3	float	463,26	4,32	939,44	458,85	270,31	9,48	2319,13	255,32	1,71
	double	514,72	5,12	1065,17	508,43	346,39	6,83	801,3	327,86	1,49
4	float	544,79	4,82	985,55	539,74	310	6,04	1474,48	295,32	1,76
	double	634,67	8,83	829,61	624,91	-	-	-	-	-
6	float	921,27	9,97	1147,8	907,06	407,54	7,21	1555,74	392,97	2,26
	double	1187,58	8,08	1357,39	1170,47	-	-	-	-	-
8	float	1262,73	13,78	1452,79	1244,98	546,21	11	1896,22	526,82	2,31
	double	1856,47	11,17	2038,89	1830,07	-	-	-	-	-

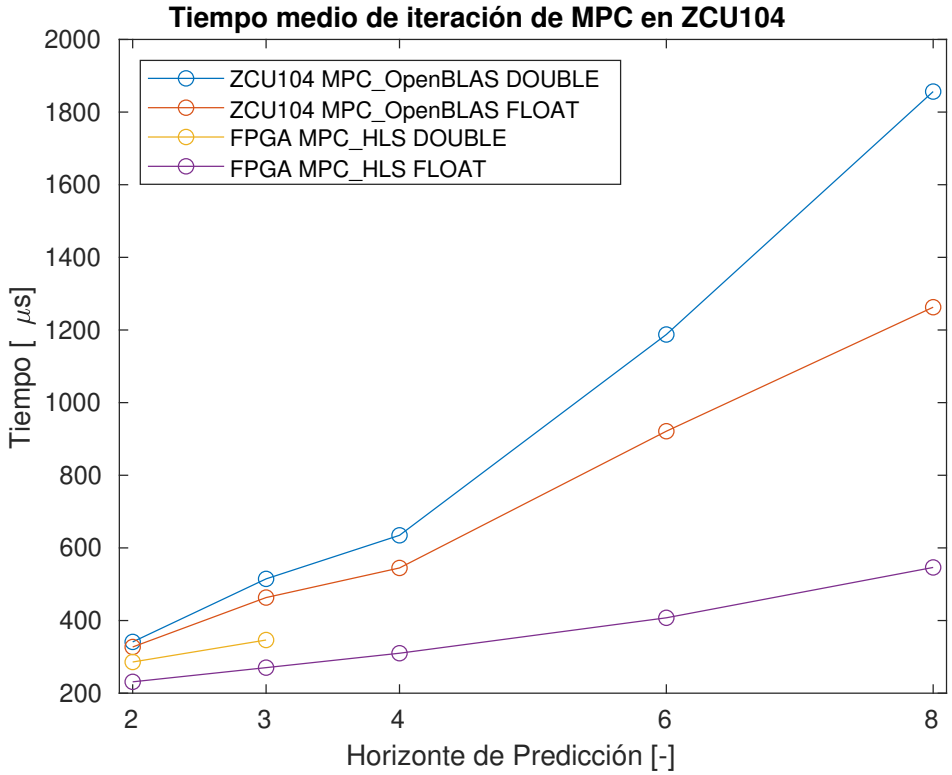

Figura 4.5: Tiempos medio de iterMPC en ZCU104 y FPGA.

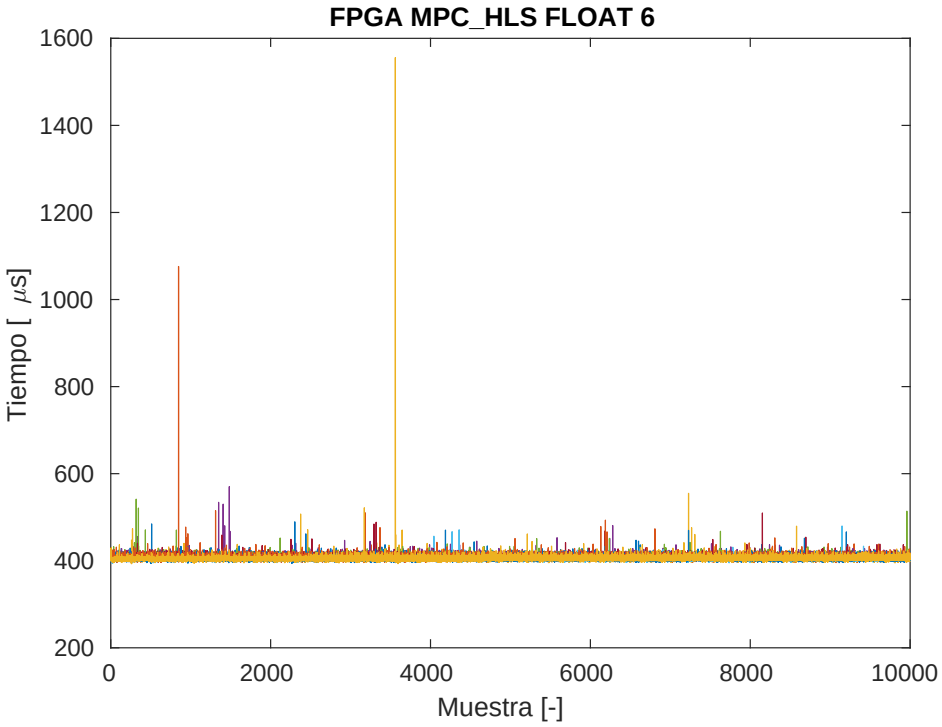

Figura 4.6: Tiempos medio de ejecución para diez pruebas en ZCU104, con *float* y $N = 6$.

5 Conclusiones y Trabajo Futuro

En el presente trabajo se evalúan las herramientas utilizadas para diseñar un sistema digital especializado para la implementación de lazos MPC utilizando un flujo de diseño basado en HLS. Para establecer una base de comparación, se han presentado y evaluado tres implementaciones de MPC en un computador de escritorio y en un procesador embebido con FPGA, donde la implementación en FPGA fue creada mediante HLS .

Este capítulo resume los principales resultados y hallazgos, discute los principales aportes de este trabajo y plantea las posibles direcciones a considerar en trabajos futuros.

5.1. Conclusiones

Al momento de iniciar este trabajo, se había identificado que el algoritmo MPC utilizado como base, basado en PDIP, presentaba potencial paralelismo que podría explotarse con una arquitectura especializada. Esto queda evidenciado al comparar las tres implementaciones presentadas. MPC_CPP no hace uso de ningún tipo de paralelismo y resultó ser la con mayor tiempo de ejecución, tanto en un computador de escritorio y en un procesador embebido. MPC_OpenBLAS hace uso de paralelismo a nivel de CPU utilizando todos los núcleos disponibles en la plataforma para realizar operaciones matriciales y resolver el sistema lineal. Aún en un procesador embebido con solo 4 núcleos se logra reducir el tiempo de ejecución hasta en 7,03 veces que MPC_CPP en la misma plataforma, solamente haciendo una utilización más eficiente de los recursos de hardware.

Utilizando HLS se logra llegar a la implementación MPC_HLS que se implementa en una FPGA. Los resultados experimentales muestran que esta implementación llega a los resultados numéricos esperados en un tiempo menor a MPC_OpenBLAS ejecutado en un procesador embebido. La aceleración va desde 1,19 veces para $N = 2$ punto flotante de precisión doble (*double*), hasta 2,31 veces para $N = 8$ punto flotante de precisión simple (*float*). Estos resultados nos indican que las herramientas para HLS utilizadas permiten crear un código HDL que produce un resultado comparable con las otras alternativas exploradas, sin necesidad de describir o manipular código HDL directamente.

Debido al carácter exploratorio en el que se enmarca este trabajo y a que el diseño está orientado a la generalidad para facilitar la evaluación de distintas dimensiones del problema de control, se espera que los resultados provean una base mínima de desempeño computacional, quedando bastante espacio de exploración para seguir aumentando el desempeño mediante arquitecturas más especializadas para cada dimensión del problema y configuraciones más avanzadas de HLS.

El proceso de desarrollo para las implementaciones que se ejecutan en CPU no tiene mayores complicaciones. La documentación es abundante y no es específica a una arquitectura o compañía. Por ejemplo, la implementación de BLAS que se ocupa es OpenBLAS, pero se utiliza la documentación provista por Intel [23] para saber cómo emplear las funciones BLAS. Tampoco se utiliza un software de desarrollo propietario, lo que hace que sea simple cambiar de plataforma.

En cuanto al uso de HLS para implementar algoritmos en FPGA, se concluye que reduce el tiempo de desarrollo del código HDL. Habría sido poco realista que alguien que no tenga años de expertiz en diseño de circuitos digitales pueda desarrollar este trabajo de memoria en el tiempo provisto, escribiendo directamente el código HDL. Utilizar HLS permite un proceso de diseño donde es sencillo de explorar alternativas de implementación ya que, por ejemplo, basta con cambiar un pragma para ver el efecto de aplicar un *unroll* o *pipeline*, lo cual permite reducir significativamente la complejidad y el tiempo de diseño en comparación a escribir un código HDL específico para cada arquitectura que se quiera probar. Sin embargo, se enfatiza que se deben tener conocimientos sobre diseño de circuitos digitales para poder sacar provecho a HLS.

Existe una gran curva de aprendizaje para utilizar las herramientas Vitis y Vitis HLS. Hay abundante documentación pero llega a ser abrumador distinguir entra la que es útil y la que no. También hay documentación que está obsoleta donde no está claro cual la reemplaza. Sin embargo, como son herramientas relativamente nuevas, se espera que la documentación y soporte mejoren a medida que aparezcan nuevas versiones.

Una desventaja de utilizar HLS para crear código HDL que se ve en este trabajo, es que las variables internas tienen nombres asignados automáticamente que no siempre son auto explicativos. Por lo tanto, al sintetizar se pueden encontrar problemas de *timing* o congestión donde los nombres de las variables que tienen problemas no están directamente relacionados con el código C++ original. Hay que apelar a la experiencia del diseñador para saber cuáles son las secciones de código que potencialmente están causando problemas o realizar ingeniería inversa para identificar de donde provienen las variables problemáticas.

5.2. Trabajo Futuro

Una limitación de este trabajo es que el código que se utiliza como base para crear MPC_HLS está pensado para ser ejecutado en CPU, lo que puede influir en el orden de las operaciones y el uso de recursos. Se recomienda que para siguientes proyectos que planeen utilizar HLS, se estudie el algoritmo, luego se cree un diagrama de bloques del circuito digital que implemente tal algoritmo y, en base al diagrama, se escriba el código C++ . De esta manera, en el momento de diseño del diagrama de bloques, se puede optimizar el orden de las operaciones y el uso de hardware de una forma que no es evidente al ver un código C++ tradicional.

Por los problemas que no permitieron sintetizar el código generado por HLS ($N = 4, 6, 8$ con punto flotante *double*), es que se recomienda no utilizar HLS para diseños en donde se estime que sea muy exigente en cuanto a los recursos de la FPGA y/o frecuencia de reloj.

Finalmente, tomando en cuenta la exploración que se hizo en este trabajo, se sugiere que el siguiente paso sea enfocarse en un único horizonte de predicción y tipo de punto flotante para que sea posible realizar optimizaciones específicas, de este modo se puede llegar a una

implementación de MPC más rápida en tiempo de ejecución. También se puede evaluar no utilizar el procesador para eliminar las variaciones de tiempo de procesamiento y poder cumplir con requerimientos duros de tiempo real.

Bibliografía

[1] J. Rawlings and D. Mayne, "Postface to model predictive control: Theory and design," *Nob Hill Pub*, vol. 5, pp. 155–158, 2012.

[2] M. L. Darby and M. Nikolaou, "Mpc: Current practice and challenges," *Control Engineering Practice*, vol. 20, no. 4, pp. 328–342, 2012.

[3] G. C. Goodwin, S. F. Graebe, M. E. Salgado, *et al.*, *Control system design*, vol. 240. Prentice hall New Jersey, 2001.

[4] J. Liu, H. Peyrl, A. Burg, and G. A. Constantinides, "Fpga implementation of an interior point method for high-speed model predictive control," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8, IEEE, 2014.

[5] H. I. Khajanchi, J. N. Bruno, and A. A. Adegbege, "An embedded fpga architecture for real-time model predictive control," *IFAC-PapersOnLine*, vol. 53, no. 2, pp. 7833–7838, 2020.

[6] E. N. Hartley and J. M. Maciejowski, "Predictive control for spacecraft rendezvous in an elliptical orbit using an fpga," in *2013 European control conference (ECC)*, pp. 1359–1364, IEEE, 2013.

[7] L. G. Bleris, J. Garcia, M. V. Kothare, and M. G. Arnold, "Towards embedded model predictive control for system-on-a-chip applications," *Journal of Process Control*, vol. 16, no. 3, pp. 255–264, 2006.

[8] K. Ling, S. Yue, and J. Maciejowski, "A fpga implementation of model predictive control," in *2006 American Control Conference*, pp. 6–pp, IEEE, 2006.

[9] I. McInerney, G. A. Constantinides, and E. C. Kerrigan, "A survey of the implementation of linear model predictive control on fpgas," *IFAC-PapersOnLine*, vol. 51, no. 20, pp. 381–387, 2018.

[10] "Repositorio Git." `https://github.com/morrisort/embeddedMPC`. Accedido: 2021-06-30.

[11] "OpenBLAS." `http://www.openblas.net`. Accedido: 2021-05-30.

[12] J. Rawlings, D. Mayne, and M. Diehl, *Model Predictive Control: Theory, Computation, and Design*. Nob Hill Publishing, 2017.

[13] A. Diaz Dorado, "Efficient convex quadratic optimization solver for embedded mpc applications," 2018.

[14] J. Nocedal and S. Wright, *Numerical optimization*. Springer Science & Business Media, 2006.

[15] F. Borrelli, A. Bemporad, and M. Morari, *Predictive Control for Linear and Hybrid Systems*. Cambridge University Press, 2017.

[16] S. J. Wright, *Primal-dual interior-point methods*. SIAM, 1997.

[17] "ARM SIMD." https://developer.arm.com/documentation/101458/2100/Optimize/Optimizing-C-C---code-with-Arm-SIMD--Neon-. Accedido: 2021-06-30.

[18] "BLAS." http://www.netlib.org/blas/. Accedido: 2021-06-30.

[19] "AOCL." https://developer.amd.com/amd-aocl/. Accedido: 2022-01-03.

[20] "Intel MKL." https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html. Accedido: 2022-01-03.

[21] "ARM Performance Libraries." https://www.arm.com/products/development-tools/server-and-hpc/allinea-studio/performance-libraries. Accedido: 2022-01-03.

[22] "cuBLAS." https://developer.nvidia.com/cublas. Accedido: 2021-06-30.

[23] "Documentación Intel para BLAS." https://www.intel.com/content/www/us/en/develop/documentation/onemkl-developer-reference-fortran/top/blas-and-sparse-blas-routines.html. Accedido: 2021-06-30.

[24] "Documentación IBM para LAPACK." https://www.ibm.com/docs/en/essl/6.2?topic=reference-lapack-lapacke. Accedido: 2021-06-30.

[25] "HLS pragmas." https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/hls_pragmas.html. Accedido: 2021-06-30.

[26] "Device Topology." https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/appdev.html#ariaid-title2. Accedido: 2021-06-30.

[27] "Optimization Techniques in Vitis HLS." https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/vitis_hls_optimization_techniques.html. Accedido: 2021-06-30.

[28] "Documentación flujos de trabajo en Vitis." https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/devrtlkernel.html#ariaid-title6. Accedido: 2021-06-30.

[29] "Biblioteca Estandar Chrono." https://en.cppreference.com/w/cpp/chrono. Accedido: 2021-06-30.