Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías Industriales

Procesamiento masivamente paralelo en control predictivo basado en datos

Autor: Alfonso Daniel Carnerero Panduro

Tutor: Daniel Rodríguez Ramírez

Dep. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2017

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías Industriales

Procesamiento masivamente paralelo en control predictivo basado en datos

Autor:
Alfonso Daniel Carnerero Panduro

Tutor:
Daniel Rodríguez Ramírez
Profesor Titular

Dep. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2017

Trabajo Fin de Grado: Procesamiento masivamente paralelo en control predictivo basado en datos

Autor: Alfonso Daniel Carnerero Panduro
Tutor: Daniel Rodríguez Ramírez

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Agradecimientos

A mis familiares, amigos y profesores.

Resumen

Durante mucho tiempo, la computación paralela ha estado relegada a supercomputadores de grandes centros de investigación. Sin embargo, con la aparición de la arquitectura CUDA en las tarjetas gráficas de NVIDIA (las cuales no suponían una inversión desorbitada), empezó a estar disponible para un público mucho más amplio.

Por otro lado, comprobaremos que la programación con las APIs proporcionadas por el *CUDA Toolkit* se plantea realmente muy similar a la que estamos acostumbrados a usar habitualmente, permitiéndonos usar diversos lenguajes populares como C o Fortran.

La razón por la que planteamos el estudio de estas técnicas encuentra su sentido en el hecho de que existen gran cantidad de problemas que pueden beneficiarse del uso de este tipo de soluciones, más ahora que estamos moviendo continuamente cantidades enormes de información.

Dicho esto, el objetivo es explorar las distintas posibilidades actuales para programar en la GPU usando software comercial como Matlab y, una vez hecho esto, implementar algoritmos que usen esta tecnología. En concreto, se resuelven principalmente 2 problemas:

- Uso de una base de datos para predecir el estado de un sistema dinámico.
- Resolución de un problema de control predictivo multivariable usando una base de datos en lugar de un modelo matemático del sistema.

También comentaremos las ventajas e inconvenientes de este tipo de soluciones, haciendo especial hincapié en las mejoras de rendimiento que conseguiremos con respecto a las versiones de los algoritmos anteriores sin paralelizar.

Abstract

For a long time, Parallel Computing had been pushed into the background due to the high cost of a supercomputer, leaving it for some investigation centers. However, this changed with the aparition of the CUDA architecture on the NVIDIA graphic cards. Those aren´t that expensive, so more people could afford one. Furthermore, We´ll see that is not really hard to programme on one if you are an experienced programmer on languages such as C or Fortran.

The reason for we want to explore this techniques finds its sense in the fact that there are a lot of problems which can be improved with these kind of parallel implementations.

Said that, We´ll explore different possibilities in order to execute code in our GPU, including commercial software such as Matlab. Having done that, We´ll implement two algorithms using these technologies:

- A program that predicts the state of a dynamic system using a database.
- A solution of a multi-variable Predictive Control problem using a database, not a mathematical model of the system.

Also, We´ll discuss the advantages and disadvantages of these kind of solutions, focusing on the the performance improvements in order to compare with the non-parallel ones.

Índice Abreviado

Índice

1 Introducción

Este trabajo intenta ser una introducción al mundo de la programación paralela de manera que a partir de los ejemplos que aquí se plantean quede claro que la implementación de este tipo de técnicas no es una tarea tan complicada como ha venido siendo durante mucho tiempo y que existen cantidad de potenciales aplicaciones. El esquema aproximado que seguiremos a lo largo de este escrito será el siguiente:

1. En primer lugar, hablaremos de la historia de la arquitectura CUDA y de sus principales características. Esto incluye el flujo de información en la tarjeta gráfica, los tipos de memoria que posee, la estructura de sus hilos, etc.
2. Posteriormente, propondremos un ejemplo de programación básico para poder comparar las diferencias entre un programa en la CPU y otro en la GPU. Además, veremos la problemática asociada a este tipo de soluciones, lo cual nos impulsa a centrarnos en unos problemas determinados.
3. Más adelante, estudiaremos algunas posibilidades incluidas en el *Matlab Parallel Computing Toolbox*.
4. Una vez que conozcamos cómo funcionan las herramientas que vamos a utilizar, plantearemos el problema de modelado de un sistema dinámico usando únicamente una base de datos, sin ningún tipo de funciones analíticas para definir su comportamiento. Comprobaremos a su vez la bondad de este último.
5. Una vez resuelto el problema de modelado, solucionaremos el problema de control de la planta de 4 tanques con un algoritmo basado igualmente en bases de datos, haciendo un estudio detallado del comportamiento y de su rendimiento.
6. Finalmente, sacaremos unas conclusiones finales en función de los resultados obtenidos.

2 Unidades de procesamiento gráfico y CUDA

En la página web de NVIDIA se define CUDA como: "*...una arquitectura de cálculo paralelo de NVIDIA que aprovecha la gran potencia de la GPU (unidad de procesamiento gráfico) para proporcionar un incremento extraordinario del rendimiento del sistema.*" [1] Pero, ¿qué es una GPU?

Una GPU (*Graphics Processor Unit*) no es más que un coprocesador encargado de aligerar la carga de la CPU (*Central Processing Unit*) en algunas tareas determinadas. En el caso de nuestro día a día, se encarga de presentar los gráficos por pantalla. Antes de pasar a estudiar su estructura y diferencias con la CPU podría ser interesante esbozar su evolución histórica.

2.1 Historia de la CPU y GPU

Durante mucho tiempo, la manera de mejorar el rendimiento de un ordenador personal estaba completamente enfocado a la mejora de su CPU. Así, durante 30 años los fabricantes fueron siendo capaces de aumentar la velocidad de sus procesadores a través de mejoras en los relojes internos de estos. De esta manera, pasaron de funcionar a unas frecuencias en torno a 1MHz a 1 GHz. Esto suponía mejorar la velocidad 1000 veces. Pero, debido a las limitaciones que poseen los procesos de fabricación, el tamaño mínimo que transistores deben tener en un chip y problemas relacionados con la temperatura, los fabricantes e investigadores tuvieron que empezar a buscar otras posibilidades.

Paralelamente, se encontraba el terreno de los supercomputadores. Estos se estaban beneficiando de las mejoras de los ordenadores personales pero, empezaron a desarrollar otra tecnología que terminó resultando de gran interés para los investigadores y fabricantes anteriormente mencionados. No se trataba de otra cosa que la de poner a trabajar varios procesadores en paralelo. De esta manera, las CPUs empezaron a a incluir varios procesadores independientes, cosa a la que ya estamos bastante acostumbrados actualmente.

¿Y dónde entra en juego la GPU? Pues bien, estas empezaron a desarrollarse sobre los años 80 como consecuencia del éxito de sistemas operativos con interfaz gráfica. En aquel entonces, algunos usuarios empezaron a demandar dispositivos que mejoraran el rendimiento del sistema a la hora de presentar los gráficos en pantalla.

De igual manera que dos mercados muy distintos estuvieron trabajando con las CPUs de manera independiente, esto se repitió en el terreno de las GPUs. Por ello, en el terreno de la computación más profesional, la compañía *Silicon Graphics* empezó a popularizar el uso de gráficos en 3D con gran cantidad de aplicaciones. En el año 1992, liberaron su interfaz de programación y sus librerías (*OpenGL*) pretendiendo que se convirtiera en el estándar de creación de gráficos tridimensionales.

Como no podía ser de otra manera, los consumidores empezaron a demandar esta tecnología debido sobre todo al lanzamiento de videjuegos inmersivos en primera persona como *Doom* o *Quake*. A partir de aquí, varias empresas como NVIDIA, ATI Technologies y 3dfx empezaron a desarrollar procesadores gráficos

más asequibles para satisfacer la demanda de estos usuarios.

En paralelo al uso lúdico de las tarjetas gráficas, algunos investigadores empezaron a estudiar la posibilidad de usarlas para algo distinto a la renderización de unos determinados gráficos. Sin embargo, la única manera de lidiar con las GPUs por aquel entonces era a través de su API (*Application Programming Interface*). El funcionamiento era el siguiente: las aplicaciones tenían que pasar como datos a las tarjetas gráficas colores, texturas, luminosidad, etc. Lo interesante venía cuando los programadores empezaron a enviar información distinta, aunque disfrazada como la mencionada anteriormente. De esta manera, se podían realizar algunas operaciones. Sin embargo, esta programación era tan restrictiva y dependiente del modelo concreto de la GPU que no logró mucho éxito dentro del sector.

Esto cambió cuando NVIDIA presentó la arquitectura CUDA (*Compute Unified Device Architecture*) en su modelo *GeForce 8800 GTX*, la cual alivia gran parte de las limitaciones de sus predecesoras en el terreno de la computación de propósito general.

2.2 Arquitectura CUDA

Gracias a la cantidad de núcleos que poseen las tarjetas gráficas, existen un buen número de problemas que pueden ver mejorado su rendimiento haciendo uso de la computación paralela. A pesar de la mejora evidente en velocidad que se nos sugiere al aumentar en tal medida la cantidad de núcleos, existen alguna deficiencias como por ejemplo:

- Siempre hay un tiempo implícito en cada ejecución relacionado con el paso de memoria entre la CPU y GPU.
- Los núcleos CUDA deben ejecutar todos el mismo código. Es decir, no son independientes.
- La sincronización entre ellos es muy restrictiva, de igual modo ocurre con los posibles intercambios de información.

El flujo de procesos de una ejecución en CUDA viene representado en la figura 2.1.

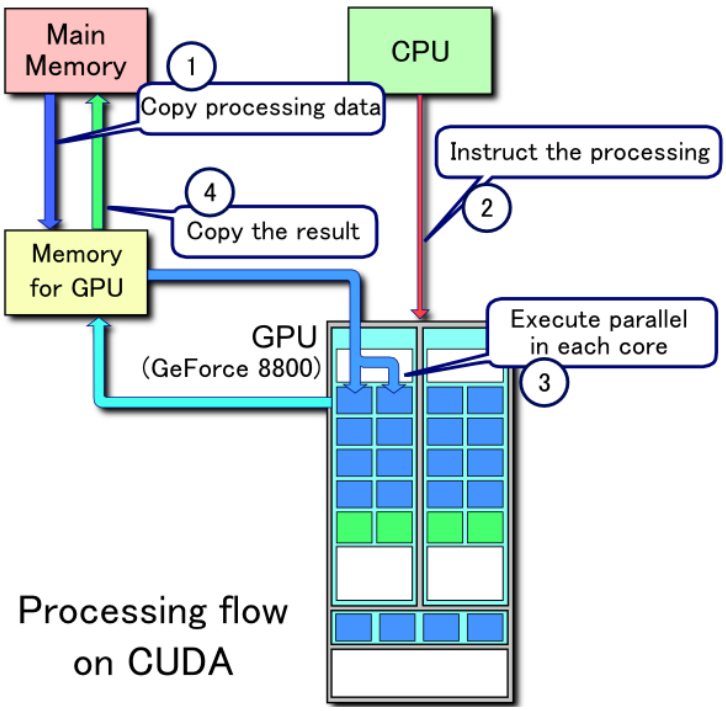


Figura 2.1 Representación del flujo de información en CUDA [1].

[1] Imagen procedente de: https://en.wikipedia.org/wiki/CUDA

En ella vemos como es necesario mover la información varias veces durante la ejecución de un programa. Los tiempos que son necesarios para este traspaso (y que son estrictamente necesarios) son denominados **overheads** y son completamente decisivos a la hora de elegir usar una GPU para una determinada aplicación. Por lo general, la mejora en velocidad surge para problemas a gran escala. Por el contrario, el tiempo de computación para problemas pequeños se mantiene igual al de un programa convencional en la CPU, llegando a empeorar en algunos casos.

Además, es necesario que el problema que queramos tratar sea paralelizable. Para explicar este concepto, haremos uso de varios ejemplos.

- Pensemos en primer lugar en las aplicaciones para las que fueran pensadas las GPUs, es decir, procesar imágenes. Una vez tenemos la información necesario del entorno que queremos representar (luces, colores, etc) cada pixel es completamente independiente del resto por lo que es posible calcular los valores RGB que son necesarios para mostrarlo por pantalla sin necesidad de conocer el resultado del resto. No importa de ninguna manera el orden de ejecución ni existe ninguna relación de dependencia.

- Planteemos ahora, en contraposición, la resolución de un sistema de ecuaciones diferenciales en tiempo discreto. En este caso no es posible resolver el sistema para un instante temporal en el que todavía no conozco el resultado del instante anterior. Es decir, debe iterarse en orden y éste no puede verse alterado.

  En cambio, ¿qué ocurriría si lo que quiero es resolver muchas veces ese sistema de ecuaciones en diferencias para distintos valores iniciales? En este caso vemos como cada núcleo de la GPU se encargará de realizar las operaciones convenientes de *su trayectoria*, completamente independiente de las del resto de núcleos.

Esto significa que la GPU nos plantea dos posibilidades de aplicación:

- Resolver problemas que son implícitamente paralelizables.
- Resolver múltiples problemas no paralelizables.

Algunos ejemplos de dispositivos que estén funcionando actualmente usando CUDA podríamos mencionar:

- Un sistema llamado *TechniScan Medical* dedicado a la recreación de imágenes tridimensionales a partir de ultrasonidos para la prevención del cáncer de mama [2].
- Prototipos de UAVs que funcionan con *NVIDIA Jetson* y tienen su funcionamiento basado en algoritmos de *deep-learning* [3].

Como podemos ver, el campo de aplicación de CUDA es realmente diverso.

La siguiente pregunta podría ser cómo está estructurada la memoria y la ejecución de programas en CUDA.

De momento, nos basta con imaginarnos que un programa en CUDA C es prácticamente código C con algunas funciones especiales referidas a la GPU. A la hora de ejecutar un *kernel* (lo cual vamos a entender como una función que se ejecuta en la GPU) se debe especificar una serie de parámetros: el número de bloques y el número de hilos. Pero, ¿qué son estos bloques e hilos?
Los hilos son la unidad indivisible de CUDA, significando que cada uno se ejecutará de manera independiente en un núcleo de la GPU en un determinado instante. Por otro lado, los bloques no son más que una manera de organizar los hilos tanto a nivel hardware como software. Esta estructura nos permite asignar índices en 3 dimensiones tanto a los hilos como a los bloques. De esta manera, podemos identificar completamente a cualquier hilo perteneciente al conjunto, lo cual facilita en gran medida muchas tareas de programación.

Esto se puede ver claramente en la figura 2.2 donde existe una malla bidimensional de bloques y cada uno de ellos tiene a su vez en su interior una matriz de hilos, siendo estos últimos la unidad más pequeña.

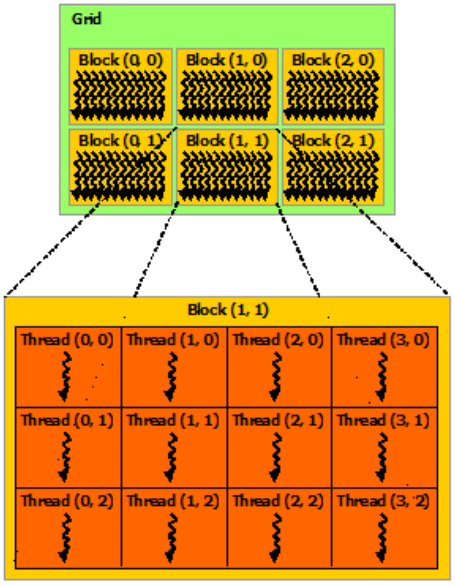


Figura 2.2 Representación esquemática de la jerarquía bloque-hilo [2].

Este concepto no se reduce únicamente a 2D sino que es extrapolable a 3D, como puede verse en la figura 2.3.

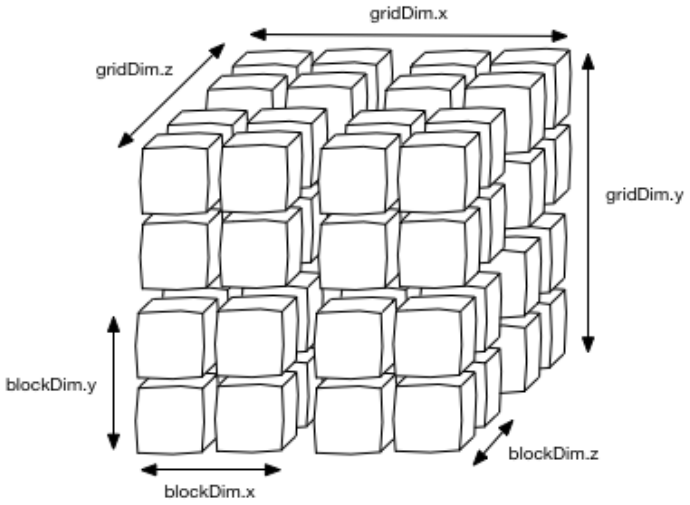


Figura 2.3 Representación tridimensional de una malla [3].

Es decir, nosotros somos los que indicamos la cantidad de hilos que se ejecutarán y también la manera en que se organizarán. Esto cobrará mayor importancia cuando expliquemos la diferencia entre programación paralela y estándar (a nivel de código).

Cabe aclarar que no se puede generar una cantidad arbitraria de bloques e hilos, sino que esto viene limitado por el modelo concreto de la GPU. Estas limitaciones incluyen un número total de hilos tanto en cantidad total como a lo largo de cada dimensión (que no es estrictamente lo mismo).

Otra parte importante de la arquitectura viene representada por los tipos de memoria que existen en su interior. Podemos diferenciar 4 tipos [2]:

- La *local memory*, propia de cada hilo e innaccesible para los demás.

[2] Imagen procedente de: http://docs.nvidia.com/cuda/cuda-c-programming-guide/#thread-hierarchy
[3] Imagen procedente de: http://www.karimson.com/posts/introduction-to-cuda/

- La *shared memory*, la cual es compartida por todos los hilos de un mismo bloque pero no es accesible a los hilos de otro bloque. La ventaja es que resulta tener un acceso más rápido que la memoria local. Este es un buen momento para explicar los mecanismos de sincronización entre hilos. Simplemente, únicamente es posible coordinar a los hilos (deben ser todos ellos) dentro de un mismo bloque. No se permite, por tanto, coordinar bloques ni todos los hilos de todos los bloques. Está limitado a la sincronización dentro de un bloque.
- La *texture memory*, la cual es de sólo lectura y se encuentra optimizada para accesos siguiendo patrones bidimensionales. Es decir, resulta mucho más favorable la consulta de vectores o submatrices que la de un elemento aislado.
- La *global memory*, visible para los hilos de todos los bloques y con un acceso más lento.

Una vez hecho este inciso, merece la pena explicar brevemente las diferencias de un programa CUDA C con uno corriente.

3 Ejemplos de programación CUDA C

Una vez esbozados cualitativamente los aspectos más generales de la GPU y sus propiedades, se presentan una serie de ejemplos con el objetivo de presentar algunos factores interesantes de la programación en CUDA.

3.1 Cálculo paralelo

En este primer ejemplo vamos a tratar de desarrollar las principales diferencias que surgen para un programa que sume 2 vectores basado en:

- La programación habitual en C, para ejecutar en la CPU.
- La programación en CUDA C, para ejecutar en la GPU.

La idea es tratar de esclarecer las diferencias que suponen a nivel de código una u otra opción. Para ello, se va a plantear un programa que suma dos vectores perfectamente conocidos. A modo de resumen, va a a haber principalmente 2 cambios:

- Va a ser necesario cambiar la sintaxis de las ”funciones” y la manera en la que estas se invocan.
- Es imprescindible plantear los problemas desde otra perspectiva. Muchos de los algoritmos que planteamos en nuestra imaginación los pensamos inherentemente secuenciales. Necesitamos, por tanto, buscar otra manera de resolver los problemas.

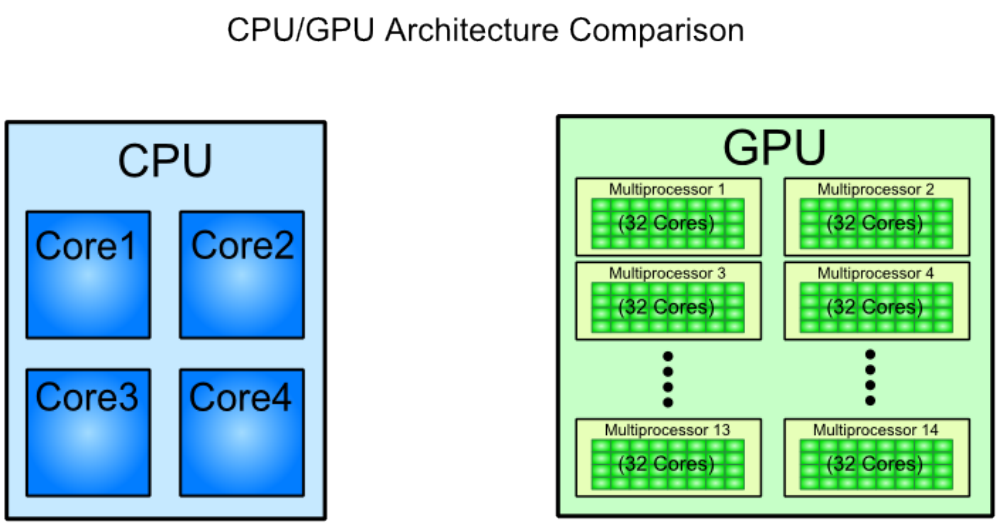


Figura 3.1 CPU vs GPU [1].

[1] Imagen procedente de: http://www.e2matrix.com/blog/wp-content/uploads/2014/01/cpu_vs_gpu-1.png

3.1.1 Versión CPU

El código es perfectamente conocido para cualquier persona con conocimientos de programación. Declaramos 3 vectores y a 2 de ellos les asignamos unos valores conocidos con el objetivo de almacenar el resultado en el tercero. Por último, mostramos el resultado por pantalla.

Para realizar este cálculo hemos optado por usar una función en lugar de asignar directamente el resultado de la operación. Esto lo hacemos con el objetivo de establecer posteriormente similitudes con el método en la GPU. Por esa misma razón, hemos decidido realizar el traspaso de la información por referencia. El ejemplo está basado en el propuesto en el capítulo 4 de [2].

```
#define N 10

void suma( int *a, int *b, int *c )

{

    for (int i = 0; i<N; i++)  c[i] = a[i] + b[i];

}

int main( void )

{

int *a, *b, *c;

a=malloc(N*sizeof(int));
b=malloc(N*sizeof(int));
c=malloc(N*sizeof(int));


for (int j=0; j<N; j++)

{
a[i] = j;
b[i] = -(j * j);
}

suma( a, b, c );

for (int j = 0; j<N; j++)  printf("%d + %d = %d\n", a[i], b[i], c[i]);

free(a);
free(b);
free(c);

return 0;
}
```

3.1.2 Versión GPU

Es fácil ver que se repiten varios patrones con respecto al ejemplo anterior:

- Necesitamos reservar memoria para las variables. En concreto, vemos como necesitamos tener la información tanto en la CPU como en la GPU. Es decir, una dirección de memoria válida en la CPU no lo es en la GPU y viceversa. Para la reserva de memoria en la GPU hacemos uso de la llamada *cudaMalloc*. Esta presenta un comportamiento muy similar al malloc habitual.

  Además, necesitamos de otra llamada para realizar los distintos intercambios de información: *cudaMemcpy*. En esta llamada aparecen algunos campos adicionales. En los primeros es necesario poner el puntero destinatario y el puntero del cual obtenemos los datos. Además, resulta interesante el último de ellos, en el cual debemos indicar en qué sentido se copia la información a través de diversas macros predefinidas por las librerías de CUDA.

  En estos casos, siempre que se hable de *device* nos estamos refiriendo a la GPU mientras que cuando mencionemos el *host* nos referimos a la CPU. Esta terminología tiene sentido debido a que nuestro programa *no se ejecuta por entero* en la GPU, sino que es lanzado por la CPU y en unos determinados instantes invocamos a la GPU para que realice una serie de operaciones (como la que tenemos más adelante).
- Invocamos a una función que realiza la operación. En este caso, se llama *suma*. Más adelante nos centraremos en ella.
- Necesitamos liberar la memoria después de usarla de la misma manera que lo haríamos en la CPU. Para ello, hacemos uso de la llamada *cudaFree*.

En cambio, también surgen algunas preguntas. La primera de ellas podría estar referida a la llamada *suma*, la cual está seguida de unos caracteres un poco extraños. Si recordamos un poco el capítulo anterior quizá lo relacionemos enseguida. Efectivamente, se trata de la cantidad de bloques e hilos que queremos ejecutar. El primero de ellos se refiere al número de bloques mientras que el segundo a los hilos.

Ahora pasaremos a comentar el contenido de la función *suma*. Se ve fácilmente que no se trata de ningún código enrevesado, a pesar de que no comprendamos algunas de las menciones que se hacen ahí.

En primer lugar, vemos que recibe parámetros como lo haría cualquier función en la CPU con la salvedad de que no es posible devolver ningún resultado por valor. Por lo tanto, el tipo de dato asociado a la función es siempre void.

Por otra parte, el comienzo *__global__* no es más que una cabecera que usamos para que el compilador sepa que se trata de un *kernel* CUDA. (A partir de ahora hablaremos de kernel refiriéndonos a este tipo de funciones).

Posteriormente, se nos presenta algo que parece una palabra reservada: *blockIdx.x*. Para facilitar su comprensión vamos a dividirla en 2 partes:

- blockIdx, viene del inglés *block index* y nunca varía. Podemos hacernos una idea de que nos va a facilitar alguna identificación del bloque que esté ejecutando actualmente el kernel. Es decir, cada bloque va a tener un identificador único que lo hace completamente diferente del resto.
- .x nos da un valor numérico en la dimensión de la malla en la que nos estamos moviendo. Rememorando un poco lo dicho anteriormente, éste puede tomar valores en 3 ejes distintos, dando como resultado 3 índices diferentes (x,y,z). Sin embargo, en este caso se ha especificado una malla monodimensional, por lo que sólo existe un índice en x.

Esto implica que somos capaces de identificar completamente a cualquier hilo en ejecución gracias a los índices anteriores, lo que nos facilita enormemente el trabajo. Por tanto, puedo asignar a cada uno de ellos la tarea de sumar una única componente del vector, de manera que si tengo un vector de 10 componentes habré sido capaz de hacer todo el trabajo ejecutando 10 hilos.

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>
#include <stdlib.h>

#define N 10

__global__ void suma(int *a, int *b, int *c)

{

    int indice = blockIdx.x;

    c[indice] = a[indice] + b[indice];

}
int main(void)
{

    int j;

    int *a, *b, *c;

    int *dev_a, *dev_b, *dev_c;

    a = (int*) malloc(N*sizeof(int));
    b = (int*) malloc(N*sizeof(int));
    c = (int*) malloc(N*sizeof(int));

    cudaMalloc((void**)&dev_a, N * sizeof(int));
    cudaMalloc((void**)&dev_b, N * sizeof(int));
    cudaMalloc((void**)&dev_c, N * sizeof(int));

    for(j = 0; j<N; j++)
    {
        a[j] = j;
        b[j] = -(j * j);
    }

    cudaMemcpy(dev_a, a, N*sizeof(int), cudaMemcpyHostToDevice);

    cudaMemcpy(dev_b, b, N*sizeof(int), cudaMemcpyHostToDevice);

    suma <<<N, 1 >>>(dev_a, dev_b, dev_c);

    cudaMemcpy(c, dev_c, N*sizeof(int), cudaMemcpyDeviceToHost);

    for (j = 0; j<N; j++)
    {
        printf("%d + %d = %d\n", a[j], b[j], c[j]);
    }

    free(a);
    free(b);
    free(c);

    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);

    getchar();

    return(0);

}
```

3.1.3 Variación de la versión GPU

En realidad, no somos capaces de sumar un vector de más de 65535 componentes (para nuestro modelo concreto) con el programa anterior. Esto es debido al límite hardware impuesto a la cantidad de bloques que podemos lanzar. Para solventarlo, usaremos una combinación de bloques e hilos.

Hemos añadido varias líneas de código, las cuáles pasamos a comentar:

- *int indice* $= threadIdx.x + blockIdx.x * blockDim.x$; Esta sentencia es muy similar a la que teníamos en el caso anterior. El funcionamiento de threadIdx es idéntico al de blockIdx, con la salvedad de que este último trabaja con bloques y el primero con hilos.

  Para comprender bien el funcionamiento de esa operación concreta puede ser ilustrativa la figura 3.2. En ella podemos ver como los distintos identificadores se complementan para poder dar a cada hilo un identificador completamente único. De igual manera es extrapolable para el caso bidimensional y tridimensional.

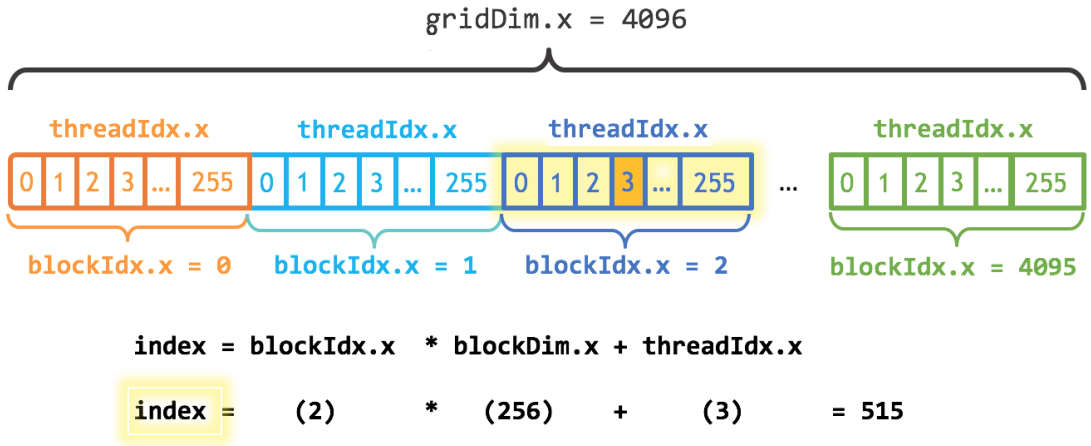


Figura 3.2 Representación de la obtención del índice de un hilo concreto [2].

- *int nbloques*, *nhpb*; , $nhpb = 16$; y $nbloques = (N+15)/16$;. Todas están orientadas a generar una malla de bloques e hilos determinada. En este caso, se encarga de generar una malla de 16 hilos por bloque (nhpb) con un número de bloques que vendrá dado por la última expresión. Esta no es más que una división entera redondeada hacia arriba. Esto lo hacemos así puesto que de lo contrario, puede darse el caso en el que no generaríamos la cantidad suficiente de hilos.

  Pero claro, vemos que por lo general se producirá lo contrario, es decir, se van a generar más hilos de los que realmente necesitamos. Para ello, hacemos uso de la sentencia siguiente.

- $if(indice < N)$ Esta sentencia es imprescindible cuando no podemos generar exactamente la cantidad de hilos necesarios, de manera que tenemos que establecer de alguna manera un límite para que únicamente actúen los que nos interesan. Por tanto, en nuestro caso ejecutarán la suma únicamente los N primeros hilos, terminando el resto inmediatamente al no cumplir la condición.

- *suma* $<<<$ *nbloques*,*nhpb* $>>>$ (dev_a, dev_b, dev_c);. Es necesario cambiar la llamada del kernel para que utilice la malla que deseamos.

[2] Imagen procedente de: https://devblogs.nvidia.com/parallelforall/wp-content/uploads/2017/01/cuda_indexing.png

```cpp
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>
#include <stdlib.h>

#define N 10

__global__ void suma(int *a, int *b, int *c)
{
    int indice = threadIdx.x + blockIdx.x*blockDim.x;

    if (indice<N)  c[indice] = a[indice] + b[indice];
}

int main(void)
{

    int j;

    int *a, *b, *c;

    int nbloques, nhpb;

    int *dev_a, *dev_b, *dev_c;

    a = (int*) malloc(N*sizeof(int));
    b = (int*) malloc(N*sizeof(int));
    c = (int*) malloc(N*sizeof(int));

    cudaMalloc((void**)&dev_a, N * sizeof(int));
    cudaMalloc((void**)&dev_b, N * sizeof(int));
    cudaMalloc((void**)&dev_c, N * sizeof(int));

    for(j = 0; j<N; j++)
    {
        a[j] = j;
        b[j] = -(j * j);
    }

    cudaMemcpy(dev_a, a, N*sizeof(int), cudaMemcpyHostToDevice);

    cudaMemcpy(dev_b, b, N*sizeof(int), cudaMemcpyHostToDevice);

    nhpb = 16;

    nbloques = (N + 15) / 16;

    suma << <nbloques, nhpb >> >(dev_a, dev_b, dev_c);

    cudaMemcpy(c, dev_c, N*sizeof(int), cudaMemcpyDeviceToHost);

    for (j = 0; j<N; j++)
    {
        printf("%d + %d = %d\n", a[j], b[j], c[j]);
    }

    free(a);
    free(b);
    free(c);
    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);
    getchar();

    return(0);
}
```

3.2 Overheads

Una vez explicadas brevemente las diferencias entre la programación en la CPU y en la GPU, vamos a pararnos en una de las deficiencias de la tarjeta gráfica que hacen que esta no sea útil para todo tipo de aplicaciones.

Como dijimos anteriormente, los overheads son los tiempos implícitos necesarios para el traspase de información de la CPU a la GPU y viceversa. Estos tiempos tienen unos valores mínimos que provocan que invocar un *kernel* para trabajar con una cantidad de datos demasiado pequeña no sea rentable y resulte más lento que usar el procesador central. Por el contrario, a partir de una determinada cantidad de datos, empezará a notarse una mejoría en la velocidad con respecto a la CPU.

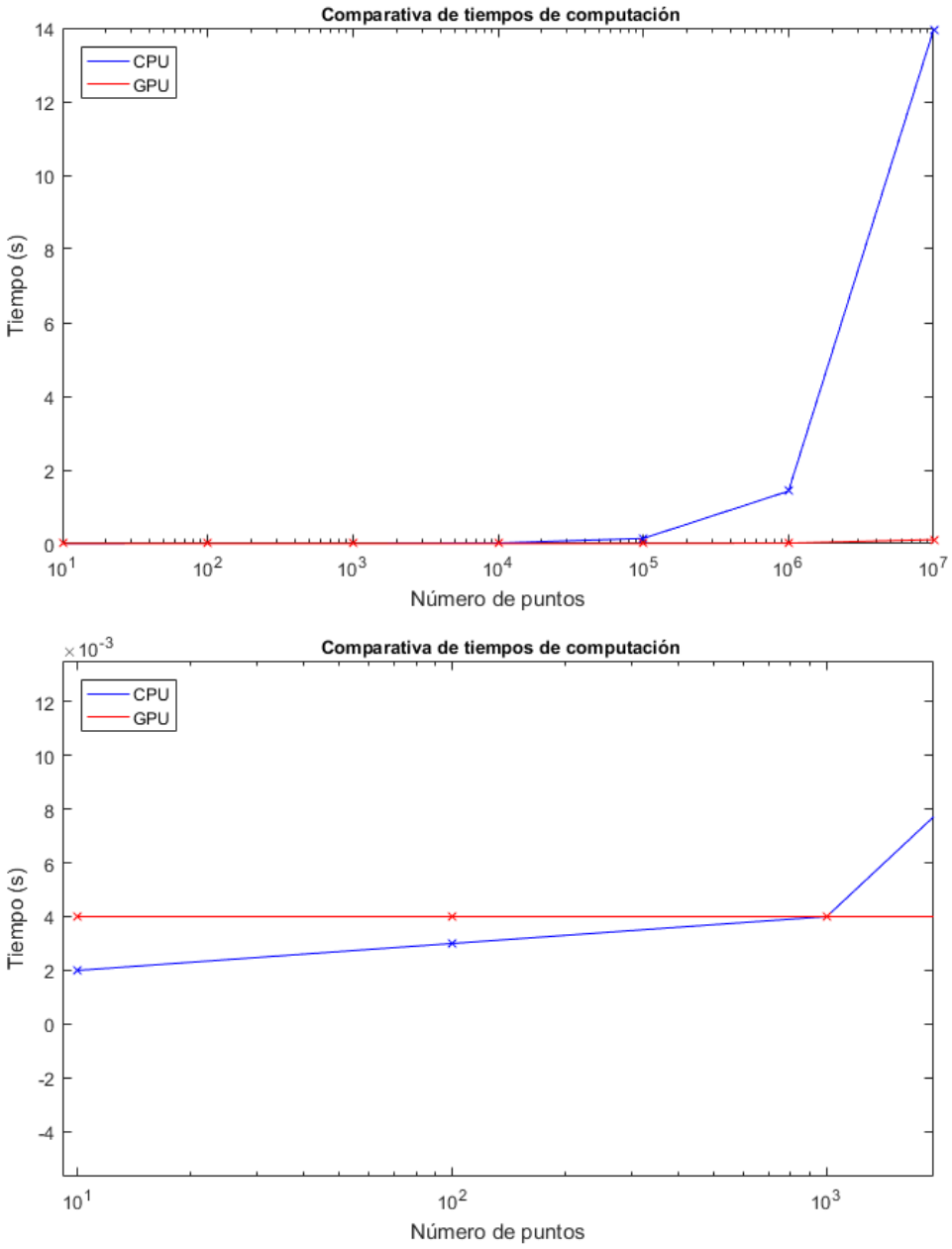


Figura 3.3 Comparativa de tiempos incluyendo un zoom del gráfico..

Lo dicho anteriormente viene contrastado por la figura 3.3. Esta corresponde a la ejecución de un *MEX*

function (ya explicaremos más adelante de qué se trata) para distintas cantidades de datos. La prueba ha sido realizada con un procesador Intel Core i7 5820K 3.3GHz y una Nvidia GeForce GTX TITAN Z. El código que se ha ejecutado se comentará posteriormente cuando expliquemos el funcionamiento de un MEX.

En el detalle podemos ver como al principio el tiempo de computación es ligeramente inferior en la CPU. Además, para el caso de la GPU, este se mantiene constante. Esto es completamente congruente con lo que hemos explicado con anterioridad. Es decir, ese tiempo se corresponde casi en su totalidad al traspaso de información entre CPU y GPU. Llegados a un determinado instante, este deja de ser representativo.

Avanzando en la gráfica, vemos que el tiempo en la CPU empieza a crecer de una manera desorbitada mientras que el de la GPU se mantiene en unos valores realmente bajos. Este tipo de problemas son los que se benefician de usar una GPU, aquellos en los trabajamos con cantidades enormes de información.

En caso contrario (cantidades pequeñas o medias), la mayor dificultad de implementación y otras limitaciones provocan que se trate de una opción enrevesada con la que no estamos seguros de obtener beneficios, llegando en algunos casos a empeorar el comportamiento.

4 Matlab Parallel Computing Toolbox

Para el desarrollo de este trabajo de fin de grado se ha hecho uso en gran medida del *Matlab Parallel Computing Toolbox*. Este incluye una serie funciones que permiten trabajar con la GPU directamente en Matlab además de permitir ejecutar programas en C, facilitando en gran medida el movimiento de la información.

El Toolbox presenta 2 posibles tipos de paralelización:

- Nivel 1: Paralelismo con una única CPU/GPU.
- Nivel 2: Paralelismo para varias CPUs/GPUs en una **misma máquina**.

Existe en realidad un tercer nivel basado en el *Matlab Distributed Computing Server*, pero está orientado a aplicaciones de supercomputación, con gran cantidad de ordenadores trabajando en un mismo problema.

Una vez dicho esto, podemos pasar a comentar brevemente algunas de las funcionalidades presentes.

4.1 GpuArray y Gather

Esta llamada permite la creación de una variable (incluyendo vectores y matrices) en la memoria de la GPU, para posteriormente trabajar con ellas. Es imprescindible para poder usar algunas de las funciones que veremos más adelante.

Gather supone el paso inverso al anterior. Es decir, se encarga de mover las variables de la GPU al workspace de la CPU.

4.2 Bsxfun

Esta función de Matlab permite realizar una serie de operaciones binarias elemento a elemento usando como datos dos vectores de entrada. Para ello, es necesario que sea una función del tipo:

$$c = f(a,b)$$

Siendo a y b vectores de igual longitud.

Los usos que puede recibir esta llamada están relacionados habitualmente con la evaluación de una función para gran cantidad de parámetros distintos. Además, se permite que el propio usuario programe su propia función.

Existen varias deficiencias que hacen que no sea de mucha utilidad:

- Sólo acepta como parámetros de entrada **2 escalares**. Ni siquiera puede prescindirse de uno de ellos.
- Únicamente devuelve un escalar. No es posible ampliar la cantidad de variables de salida.
- La función no puede acceder a un elemento arbitrario del vector, solo al que le ”corresponde”.

Para mostrar su uso, vamos a realizar la resta de dos vectores iguales de 10 componentes.

```
>> a=gpuArray(1:10)

a =

     1     2     3     4     5     6     7     8     9    10

>> b=gpuArray(1:10)

b =

     1     2     3     4     5     6     7     8     9    10

>> c=bsxfun(@minus,a,b)

c =

     0     0     0     0     0     0     0     0     0     0
```

Si necesitáramos tener el resultado de *c* en la CPU no hay más que invocar la llamada gather de la manera:

```
>> c=gather(bsxfun(@minus,a,b));
```

4.3 Pagefun

En el caso de pagefun, se permite aplicar funciones a cada página de una matriz. Matlab define como páginas los valores de la tercera coordenada de una matriz tridimensional. Es decir, estaríamos trabajando con cada una de las matrices bidimensionales que componen la matriz global (figura 4.1).

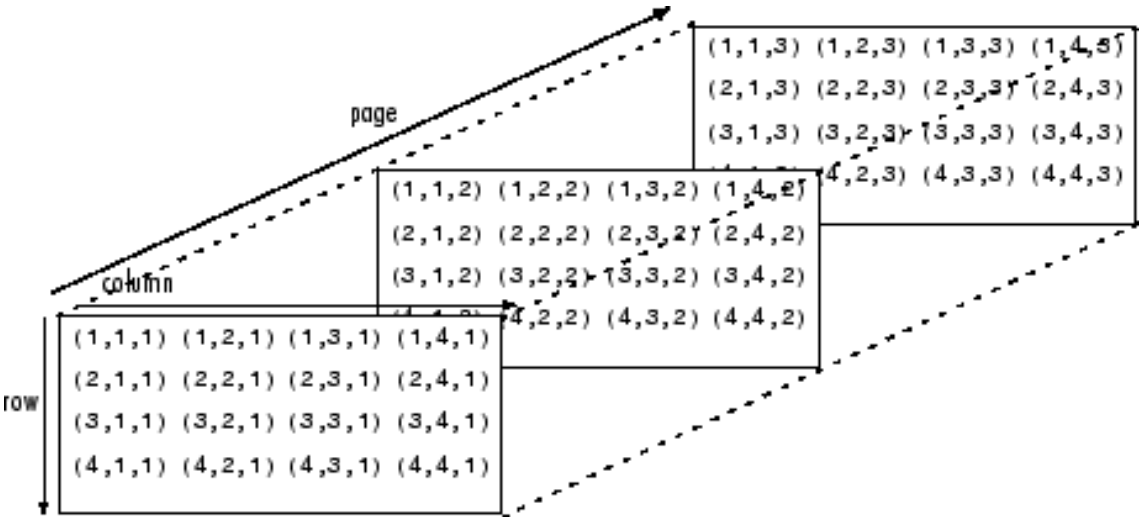


Figura 4.1 Representación gráfica de una matriz tridimensional [1].

En este caso, tenemos control total sobre cada ”sub-matriz” , permitiéndonos acceder a cualquier elemento de esta. Además, no existen restricciones en los parámetros de entrada ni de salida por lo que parece una solución más flexible que bsxfun.

Sin embargo, tiene una desventaja que provoca que tampoco sea de gran interés y es el hecho de que acepta únicamente unas pocas funciones de Matlab. Esto quiere decir que el usuario no tiene la oportunidad de

[1] Imagen procedente de: https://www.mathworks.com/help/matlab/math/ch_data_struct4.gif

programar funciones arbitrarias para usarlas junto a esta llamada.

Vamos a ampliar el ejemplo anterior para que muestre la resta de una nube de puntos con otro en un espacio tridimensional.

```
>> a=gpuArray(rand(3,1,2));
>> b=gpuArray(rand(3,1));
>> c=pagefun(@minus,a,b)

c(:,:,1) =

    0.4416
    0.0502
    0.7050


c(:,:,2) =

    0.2897
   -0.7021
   -0.1050
```

De esta manera, obtenemos en *c* la resta de los puntos de *a* (organizados por páginas como vectores columna) con *b*. Cabe aclarar que el resultado obtenido en *c* también está ordenado por páginas.

4.4 Arrayfun

Arrayfun supone una ampliación de lo que era bsxfun. Se adquiere flexibilidad a la hora de decidir los parámetros de entrada y salida. Sin embargo, sigue poseyendo algunas restricciones que tenía su hermana mayor: sólo puede devolver escalares, tomar operandos escalares y no tiene la posibilidad de acceder a otros elementos de los vectores de entrada.

Para mostrar el comportamiento de arrayfun se ha programado una función tal que:

```
function [res1,res2] = ejemplo_arrayfun(x,n1,n2)

res1=x*n1-n2;
res2=x*n2-n1;

end
```

De manera que la usaríamos así:

```
>> x=gpuArray(1:5);
>> n1=gpuArray(-5:1:-1);
>> n2=gpuArray(10:-1:6);
>> [res1,res2]=arrayfun(@ejemplo_arrayfun,x,n1,n2)

res1 =

   -15   -17   -17   -15   -11


res2 =

    15    22    27    30    31
```

Podemos ver como hemos ganado en versatilidad con respecto a bsxfun, puesto que nos ha permitido tener 3 variables de entrada y devolver 2 a la salida. Seguimos teniendo aún así la limitación que nos impide devolver otra cosa que no sea un escalar en cada iteración.

4.5 Ejecución de Kernels en CUDA C

La ejecución de Kernels desde Matlab es una de las opciones más interesantes que tenemos actualmente debido principalmente a las siguientes razones:

- Tenemos una gran flexibilidad a la hora de definir las variables de entrada, salida y las operaciones que se realizan en su interior. Es decir, no estamos restringidos a devolver escalares, aceptar una serie de componentes, etc. Además, tenemos control total sobre la malla y la cantidad de hilos que ejecutan el Kernel.
- Es posible decidir cómo gestionar algunos tipos de memoria de la GPU a los que Matlab no era transparente, como la *constant memory*.
- Libera de la complejidad que supone la programación de un fichero MEX (mucho más engorroso).
- Debido a que se comportan como funciones, son fácilmente implementables en un fichero MEX que se construya posteriormente.

Sin embargo, presentan una serie de inconvenientes:

- No permiten la reserva de memoria dinámica durante su ejecución. Por esta razón, es necesaria reservarla anteriormente en Matlab.
- A pesar de que los Kernels son realmente una herramienta muy flexible, no somos capaces de solucionar un problema entero con la ejecución de una única función. Esto conlleva a que necesitemos de varios de ellos, con el *overhead* correspondiente asociado en cada ejecución.

  Ya hemos visto anteriormente ejemplos de Kernels en C cuando explicamos las diferencias entre la programación en la CPU y la GPU. La diferencia es que ahora lo ejecutamos desde Matlab, por lo que hacemos explícito cómo sería necesario llamarlo desde este entorno:

  – Es necesario ”inicializarlo”. Es decir, tenemos que definir que archivo lo contiene, la malla y la cantidad de hilos por bloque con la que queremos ejecutarlo.

```
distancia = parallel.gpu.CUDAKernel('KernelResta.ptx','KernelResta.cu');
distancia.ThreadBlockSize = 1024;
distancia.GridSize=ceil(N/1024);
```

  – Una vez hecho esto, se ejecuta a través de la llamada *feval*. En el primer argumento indicamos el ”objeto” y, posteriormente, van todos las variables de entrada y salida.

```
d=feval(distancia,BD(:,5:8),h,d,N);
```

4.6 Ejecución de ficheros MEX en CUDA C

La resolución de problemas mediante la ejecución de ficheros MEX desde Matlab es la mejor solución actualmente. Tenemos una serie de ventajas tales como:

- Control total sobre el uso de los diferentes tipos de memoria, variables, etc.
- Nos permite acceder a librerías externas como por ejemplo CUBLAS.
- Podemos reservar memoria de manera dinámica.

El principal inconveniente del uso de este tipo de ficheros es el fuerte incremento de la complejidad con respecto a las anteriores posibilidades. Esto es así en primer lugar por el hecho de que ahora es necesario programar una aplicación entera en lenguaje C, por lo que perdemos las facilidades que Matlab nos proporciona en algunos aspectos. Además, hay que seguir estrictamente una serie de reglas para que todo funcione correctamente. Vamos por tanto a explicar la estructura y funcionamiento de un MEX a partir de uno ya programado.

Este ejemplo recibe dos parámetros: Una nube de puntos y la localización de otro punto. El objetivo es buscar en la nube de puntos aquel que se encuentra más cerca de este último. Por tanto, el programa va a tener que realizar una serie de operaciones básicas como:

- Definición de los Kernels y librerías.
- Inicialización.
- Cálculo de la distancia.
- Búsqueda del valor mínimo.
- Devolución de los resultados.

El aumento de complejidad a la hora de programar el MEX se encuentra en la segunda y última etapa, las cuáles están muy relacionadas entre sí.

4.6.1 Definición de los Kernels y librerías

Este apartado corresponde exactamente a lo mismo que describimos anteriormente. Los Kernels que invocamos desde Matlab tienen la misma estructura que los correspondientes a los ficheros MEX, únicamente cambia la manera de llamarlos. Aún así, comentamos brevemente la función de cada uno.

- La primera función se trata de una invocación en la CPU para calcular el logaritmo en base 2 a partir de las propiedades de los logaritmos. Lo usaremos para decidir la cantidad de bloques que lanzar.

- *Copia_vector* hace exactamente lo que su nombre indica. En realidad, también podrían usarse llamadas del tipo *CudaMemcpy*.

- El objetivo de *Compara* es encontrar el índice del punto mínimo una vez este ha sido calculado. Esta operación no se puede realizar directamente debido al algoritmo que se ha usado para el cálculo del mínimo.

- La función de la llamada *Norma* es calcular la raíz cuadrada de la resta.

- En *Min* es donde se realiza el cálculo del mínimo. Además, nos va a dar pie para poner énfasis otra vez en la necesidad de desarrollar nuevos algoritmos.

- La llamada *Resta* realiza la resta de la nube con respecto al punto concreto. Es un paso previo al cálculo de la norma.

```
#include "mex.h"
#include "gpu/mxGPUArray.h"
#include <math.h>

double Log2( double n )
{
    return log( n ) / log( 2 );
}

__global__ void Copia_Vector(double * vector_origen,double *vector_destino,const double N)
{
int Bloque = blockIdx.x*1024 +threadIdx.x;
if(Bloque<N)
{
vector_destino[Bloque]=vector_origen[Bloque];
}
return;
}

__global__ void Compara(const double *min,const double *norma,const double N,double *d_min,double *d_p_minimo,const double *datos)
{
int Bloque = blockIdx.x*1024 +threadIdx.x;
if(Bloque<N)
{
if(norma[Bloque]==(*min))
{
*d_min=*min;
d_p_minimo[0]=datos[3*Bloque];
d_p_minimo[1]=datos[3*Bloque+1];
d_p_minimo[2]=datos[3*Bloque+2];
}

}
return;
}

__global__ void Norma(const double * m,double *res,const double N)
{
int Bloque = blockIdx.x*1024 +threadIdx.x;
if(Bloque<N)
{
res[Bloque]=sqrt(m[Bloque*3]*m[Bloque*3]+m[3*Bloque+1]*m[3*Bloque+1]+m[3*Bloque+2]*m[3*Bloque+2]);
}
return;
}

__global__ void Min(double *m,int N,double *min,int impar)
{
int hilo = blockIdx.x*1024 +threadIdx.x;
                if(hilo<N)
                {
                        if(impar==1 && hilo==(N-1))
                                {
                                    m[hilo]=m[hilo*2];
                                }
                        else
                                {
                                    if(m[2*hilo]<m[2*hilo+1])
                                            {
                                                m[hilo]=m[hilo*2];
                                            }
                                    else
                                            {
                                                m[hilo]=m[2*hilo+1];
                                            }
                                }
                }
    if(hilo==0)
    {
        (*min)=m[0];
    }
return;
}

__global__ void Resta(const double * m,const double *punto,double *res,const double N)
{
int Bloque = blockIdx.x*1024 +threadIdx.x;
if(Bloque<N)
{
res[3*Bloque]=m[3*Bloque]-punto[0];
res[3*Bloque+1]=m[3*Bloque+1]-punto[1];
res[3*Bloque+2]=m[3*Bloque+2]-punto[2];
}
return;
}
```

```
void mexFunction(int nlhs, mxArray *plhs[],int nrhs, mxArray const *prhs[])
{

mxGPUArray const *datos;
mxGPUArray const *punto;

mxGPUArray *p_minimo;
mxGPUArray *minimo;

double const *d_datos;
double const *d_punto;

double *d_minimo;
double *d_p_minimo;

int i;
int N;
int val;
int f;
int n_iteraciones;
double *d_resta;
double *d_norma;
double *d_m;
double *d_temp;

mxInitGPU();

dim3 HebrasPorBloque(1024);
int BloquesPorRejillaX,BloquesPorRejillaY;
int GridX,GridY;

mwSize dim_p_minimo[3];
dim_p_minimo[0]=3;
dim_p_minimo[1]=1;
dim_p_minimo[2]=1;

mwSize dim_minimo[3];
dim_minimo[0]=1;
dim_minimo[1]=1;
dim_minimo[2]=1;


p_minimo=mxGPUCreateGPUArray((mwSize) 3,dim_p_minimo,mxDOUBLE_CLASS,mxREAL,MX_GPU_INITIALIZE_VALUES);
minimo=mxGPUCreateGPUArray((mwSize) 3,dim_minimo,mxDOUBLE_CLASS,mxREAL,MX_GPU_INITIALIZE_VALUES);

datos = mxGPUCreateFromMxArray(prhs[0]);
punto = mxGPUCreateFromMxArray(prhs[1]);

d_datos = (double const *)(mxGPUGetDataReadOnly(datos));
d_punto = (double const *)(mxGPUGetDataReadOnly(punto));

d_minimo=(double *)(mxGPUGetData(minimo));
d_p_minimo=(double *)(mxGPUGetData(p_minimo));

N=(int) mxGPUGetNumberOfElements(datos)/3;

BloquesPorRejillaX= (N+1023)/1024;
BloquesPorRejillaY= 1;
dim3 BloquesPorRejilla(BloquesPorRejillaX, BloquesPorRejillaY);

GridX= (N+1023)/1024;
GridY= 1;
dim3 Grid(GridX, GridY);

cudaMalloc( (void**)&d_resta, sizeof(double)*N*3);

cudaMalloc( (void**)&d_norma, sizeof(double)*N);

cudaMalloc( (void**)&d_temp, sizeof(double)*N);
```

4.6.2 Inicialización

En la página anterior puede verse el código correspondiente a la inicialización. Aprovechamos para detenernos a revisar la estructura que tiene el inicio de un fichero MEX:

- La definición de la *mexFunction* es siempre la misma. No puede cambiarse el tipo de dato que devuelve, su nombre o los parámetros que recibe.
  - nlhs. Es un número entero que indica la cantidad de variables de salida.
  - plhs. Es una tabla que almacena todas las salidas.
  - nrhs. Es un número entero que indica la cantidad de variables de entrada.
  - prhs. Es una tabla que almacena todas las entradas.
- Definimos una serie de punteros como *mxGPUArray*. Los que llevan añadido el "const" corresponderán a valores de entrada y, por tanto, serán constantes. Sin embargo, estos sirven únicamente para comunicarse con Matlab, por lo que necesitamos un puntero accesible desde nuestro programa. El criterio que se ha seguido para los nuevos punteros ha sido añadir *d_* a los nombres anteriores. Además, generamos algunas variables auxiliares que usaremos a lo largo del programa.
- Una vez hecho esto, es necesario usar la llamada *mxInitGPU* puesto que es imprescindible para cargar la librería GPU de Matlab.
- Una vez finalizado lo anterior, generamos algunas variables *dim3* que nos ayudarán a distintos propósitos. Algunas de ellas nos permitirán crear la malla deseada de bloques e hilos. Otras, en cambio, las usaremos para indicar la dimensión de los vectores de salida.
- Más adelante, creamos los objetos *mxGPUArray* y obtenemos los punteros a los datos, tanto de entrada como de salida. En algunas de estas sentencias se hace uso de algunas macros incluidas en las librerías, con el objetivo de inicializar de una manera determinada.
- Una vez hemos terminado de declarar variables, pasamos a poner valores en algunas de ellas.
- A partir de aquí, ya podemos usar llamadas del tipo *cudaMalloc* con normalidad.

4.6.3 Cálculo de la distancia

Para el cálculo de la distancia haremos uso de varios de los Kernels vistos anteriormente.

```
Resta<<<BloquesPorRejilla, HebrasPorBloque>>>(d_datos,d_punto,d_resta,N);

Norma<<<BloquesPorRejilla, HebrasPorBloque>>>(d_resta,d_norma,N);

Copia_Vector<<<BloquesPorRejilla, HebrasPorBloque>>>(d_norma,d_temp,N);
```

4.6.4 Búsqueda del valor mínimo

El fragmento de código posterior sintetiza el cálculo del mínimo en la GPU. En el subapartado siguiente nos detenemos brevemente en el método utilizado.

```
n_iteraciones=ceil(log2(N));

                if(N%2==0)
                {
                        val=N/2;
                        f=0;
                }
                else
                {
                        val=(N+1)/2;
                        f=1;
                }
for(i=0;i<n_iteraciones;i++)
{
Min<<<BloquesPorRejilla, HebrasPorBloque>>>(d_temp,val,d_m,f);
                if(N%2==0)
                {
                        val=N/2;
                        f=0;
                }
                else
                {
                        val=(N+1)/2;
                        f=1;
                }
GridX = ceil(val/1024);
dim3 Grid(GridX, GridY);
}

Compara<<<BloquesPorRejilla, HebrasPorBloque>>>(d_m,d_norma,N,d_minimo,d_p_minimo,d_datos);
```

Cálculo del Mínimo

Para calcular el valor mínimo de un vector en la CPU existen muchas formas de hacerlo. La mayoría de los métodos están basados en recorrer el vector entero y reordenarlo o bien comparar cada elemento con el valor mínimo actual, actualizándolo cuando encontramos uno menor.

Todas estas soluciones están basadas en una programación enteramente concurrente, por lo que no es posible trasladarlos a la GPU.

La solución propuesta, por tanto, trata de aprovechar las virtudes de la GPU: la gran cantidad de núcleos que posee. El objetivo es lanzar un hilo por cada 2 elementos del vector y comparar cuál de los 2 es inferior, de manera que nos quedaremos únicamente con ese. Este proceso se repite de manera iterativa hasta que sólo quede uno de ellos. Este esquema queda descrito en la figura 4.2.

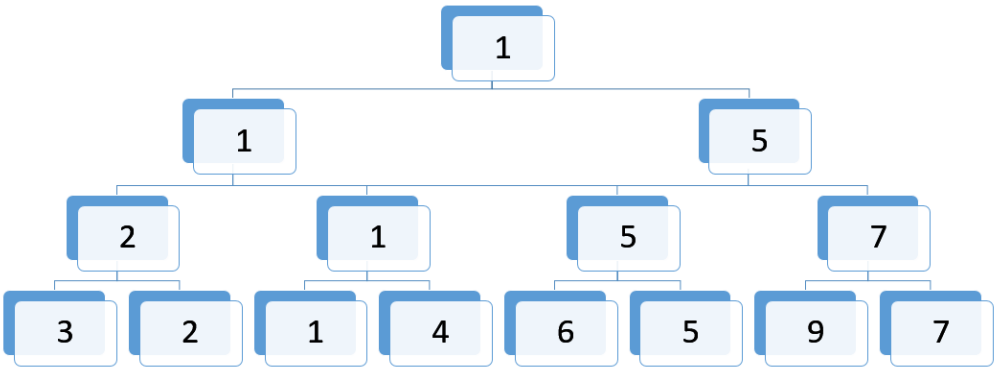


Figura 4.2 Representación gráfica del cálculo del mnínimo..

Aún así, esto necesita de alguna modificación. Por ejemplo, hemos supuesto que el número muestras es una potencia de 2, lo cual no tendría por qué ser necesariamente así. Por ello, se han incluido en el código algunas sentencias con las que solventar este problema. La solución implementada consiste en buscar si la cantidad de datos remanentes son pares o impares. En el caso de que sean pares no implica que sea necesariamente una potencia de 2, pero sí nos asegura que podemos iterar al menos una vez más. Para el caso contrario, trataremos el problema como si fuera par y, cuando lleguemos al último elemento (como no tenemos con qué compararlo) lo mantendremos sea cual sea su valor. De esta manera, ya seríamos capaces de trabajar con vectores de tamaño arbitrario.

Otro problema más fácil de solventar se correspondía con el caso de que hubiese varios elementos iguales. Esto no supone una problemática complicada. La solución propuesta es tan sencilla como quedarse con el más lejano en el vector. En realidad, el criterio podría haber sido cualquier otro.

4.6.5 Devolución de los resultados

Una vez hemos hallado el valor mínimo, buscamos el punto al que este corresponde usando el Kernel *Compara*. Una vez hecho esto, copiamos la información a los objetos *mxGPUArray*. Por último, liberamos toda la memoria reservada dinámicamente, tanto en la CPU como en la GPU.

```
Compara<<<BloquesPorRejilla, HebrasPorBloque>>>(d_m,d_norma,N,d_minimo,d_p_minimo,d_datos);

plhs[0] = mxGPUCreateMxArrayOnGPU(minimo);

plhs[1] = mxGPUCreateMxArrayOnGPU(p_minimo);

cudaFree(d_resta);
cudaFree(d_norma);
cudaFree(d_m);
cudaFree(d_temp);

mxGPUDestroyGPUArray(datos);
mxGPUDestroyGPUArray(punto);
mxGPUDestroyGPUArray(p_minimo);
mxGPUDestroyGPUArray(minimo);

}
```

5 Modelado del sistema

En el ámbito del control automático es imprescindible poseer suficiente información para que sea posible diseñar controladores capaces de llevar el sistema de un determinado estado a otro deseado por nosotros. Esto se puede realizar de diversas maneras, a veces de manera muy sencilla y sin necesidad de gran cantidad de datos, como es el caso del diseño de controladores por Ziegler-Nichols, donde únicamente es necesario consultar la señal de salida del sistema.

Otra opción sería la de plantear un modelo matemático de la dinámica del sistema en función de transferencia de la manera:

$$H(s) = \frac{Y(s)}{X(s)} \tag{5.1}$$

$$F(z) = \frac{Y(z)}{X(z)} \tag{5.2}$$

Correspondiendo la ecuación 5.1 a un modelado en tiempo continuo mientras que la ecuación 5.2 corresponde a un sistema en tiempo discreto.

Estos modelos pueden hallarse de manera precisa en algunos casos a partir de los primeros principios, es decir, planteando las ecuaciones diferenciales que modelan el comportamiento del sistema. Sin embargo, en gran cantidad de ocasiones, la dinámica es tan compleja que no es posible esta definición.

Además, gran cantidad de sistemas presentan comportamientos fuertemente no lineales, lo que hace necesaria una linealización de este último en torno a un punto de funcionamiento determinado.

5.1 Descripción del sistema

Antes de describir la solución propuesta, pasamos a describir el sistema, el cual es conocido como la planta de 4 tanques, presentada por *K.H. Johansson* en diversos artículos [11].

Como puede verse en la figura 5.1, la planta consta de 4 depósitos donde los tanques superiores descargan en los inferiores. Dispone además de dos bombas en la parte más baja que se encargan de proporcionar los caudales (q_a y q_b) para que estos recipientes puedan llenarse. Sin embargo, los caudales se ven divididos a medio camino tras pasar por una válvula de 3 vías, de manera que parte irá a parar al depósito superior y el resto al inferior (para cada uno de los 2 grupos de tanques). La apertura de estas válvulas se realiza de manera manual y definen en tanto por uno la manera en que se dividen los caudales (γ_a y γ_b).

Por ejemplo, el tanque 3 recibe un caudal $(1-\gamma_b)q_b$ y descarga en el 1 el cual a su vez recibe un caudal de valor $\gamma_a qa$. Lo mismo es extensible para los otros dos.

[1] Imagen procedente de: https://ai2-s2-public.s3.amazonaws.com/figures/2016-11-08/3f51ca30ec40fab27f9b71043eff97337df77ad7/1-Figure2-1.png

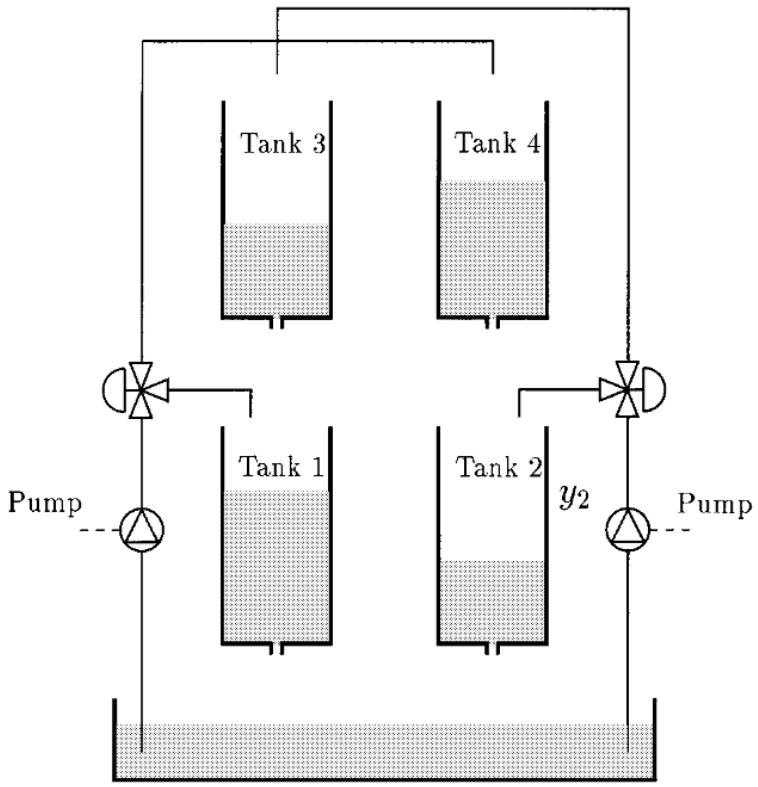


Figura 5.1 Representación gráfica de la planta de 4 tanques [1].

La planta posee por tanto dos variables manipulables (q_a,q_b) y cuatro alturas a controlar. Del balance anterior se sobreentiende que no será posible alcanzar alturas cualesquiera en cada uno de los tanques con ese número de actuaciones. Estamos por tanto ante un sistema multivariable que puede presentar ceros en su dinámica dependiendo del valor de γ_a y γ_b.

La dinámica del sistema es no lineal y puede describirse sin grandes problemas a partir de los primeros principios. El sistema de ecuaciones siguiente define la dinámica de la planta. Las letras h_k son las alturas de los tanques medidas en metros. Por otro lado, las a_k representan las secciones de los orificios de descarga mientras que la sección correspondiente a los tanques viene representada por la letra A, ambas en m^2.

$$A\frac{dh_1}{dt} = -a_1\sqrt{2gh_1} + a_3\sqrt{2gh_3} + \gamma_a\frac{q_a}{3600} \tag{5.3}$$

$$A\frac{dh_2}{dt} = -a_2\sqrt{2gh_2} + a_4\sqrt{2gh_4} + \gamma_a\frac{q_b}{3600} \tag{5.4}$$

$$A\frac{dh_3}{dt} = -a_3\sqrt{2gh_3} + (1-\gamma_b)\frac{q_b}{3600} \tag{5.5}$$

$$A\frac{dh_4}{dt} = -a_4\sqrt{2gh_4} + (1-\gamma_a)\frac{q_a}{3600} \tag{5.6}$$

5.2 Modelo basado en datos

Llegados a este punto, nos gustaría obtener un modelo discreto de la planta como el expresado en la ecuación 5.7. Esto corresponde obviamente a una definición genérica en tiempo discreto. Nuestro objetivo es predecir la evolución del vector de estados cada cierto tiempo de muestreo siguiendo la ecuación siguiente:

$$x_{k+1} = f(x_k, u_k) \tag{5.7}$$

Una posibilidad supondría la linealización del modelo continuo anterior y su posterior discretización, lo cual, además de ser engorroso, nos obligaría a realizar este proceso varias veces, para los distintos puntos de operación deseados.

La solución propuesta (ya planteada anteriormente en [12]) pretende aprovecharse de los datos históricos de la planta, la cual ha estado funcionando hipotéticamente un tiempo con unos controladores que sabemos por experiencia que funcionan. La ecuación anterior quedaría reescrita de la manera:

$$x_{k+1} = MBD(x_k, u_k, BD) \tag{5.8}$$

Siendo *BD* la base de datos antes mencionada.

En posteriores subapartados nos dedicaremos a explicar la estructura de la base de datos, el funcionamiento del modelo y su bondad en la predicción.

5.2.1 Estructura de la base de datos

La base de datos usada para el cálculo de los estados futuros está formada por 11 columnas (de las cuáles no usaremos todas, pues algunas únicamente proporcionan información para el problema de control) y un número de filas dependiente del número de trayectorias en su interior, el cual va a definir en gran parte el funcionamiento de la misma y su error de predicción.

Explicamos a continuación los contenidos de cada una de las columnas:

- El primer término identifica a la trayectoria con un número entero. Esto significa que todas las entradas a la base de datos con el mismo valor en la primera columna pertenecen a la misma trayectoria.
- El segundo nos indica el controlador que se usó durante la grabación de la trayectoria. En nuestro caso PI o LQR.
- Los siguientes 2 términos tienen relación con la referencia que queremos alcanzar en los tanques (en este caso, en el primero y en el segundo). La razón por la que únicamente almacenamos 2 valores se comentó brevemente en apartados anteriores. Esto era debido a que las referencias de los tanques 3 y 4 están acopladas a la resolución del problema de los otros 2, debido a que sólo poseemos dos actuaciones.
- Los 4 siguientes parámetros indican el estado actual del sistema en ese determinado instante de tiempo.
- En los 2 campos posteriores se guardan los valores de las actuaciones proporcionadas por el controlador.
- Por último, se guarda el instante temporal dentro de la trayectoria.

Por tanto, entendemos por trayectoria del sistema toda la información que almacenamos en la base de datos (estado, referencias, actuaciones, etc) desde su inicio hasta que alcanza el régimen permanente con un error del 1.5 %. Esto quiere decir que el tamaño de cada una de ellas es diferente en cada caso.

El criterio que se ha usado para generar la base de datos es el siguiente: Se ha realizado una malla para todos los pares de referencia factibles (ver figura 5.2) en los cuáles se ha ido avanzando con un paso de $0.02\,m$ en cada uno de ellos. Además, para cada uno de estos grupos de referencias, se ha generado una misma cantidad de trayectorias partiendo de estados iniciales aleatorios. Esto quiere decir que cuando posteriormente hablemos de diferentes tamaños de la base de datos, nos referiremos a que partimos de una base de datos la cual hemos diezmado para probar los distintos efectos que tiene la cantidad de trayectorias sobre el rendimiento del ordenador y el comportamiento del sistema.

La región factible proporcionada por la imagen 5.2 se obtiene al resolver el sistema de ecuaciones no lineales descrito anteriormente en el caso de régimen permanente para los valores máximos y mínimos de las actuaciones q_a y q_b, es decir:

Tabla 5.1 Ejemplo de celdas en la base de datos.

Nº trayectoria	Controlador	Referencia	Estado	Actuación	Tiempo
1	1	x_1^{ref}	x_0^1	u_0^1	0
1	1	x_1^{ref}	x_1^1	u_1^1	5
1	1	x_1^{ref}	x_2^1	u_2^1	10
⋮	⋮	⋮	⋮	⋮	⋮
1	1	x_1^{ref}	x_t^1	u_t^1	t
i	m	x_i^{ref}	x_0^i	u_0^i	0
i	m	x_i^{ref}	x_1^i	u_1^i	5
i	m	x_i^{ref}	x_2^i	u_2^i	10
⋮	⋮	⋮	⋮	⋮	⋮
i	m	x_i^{ref}	x_t^i	u_t^i	t

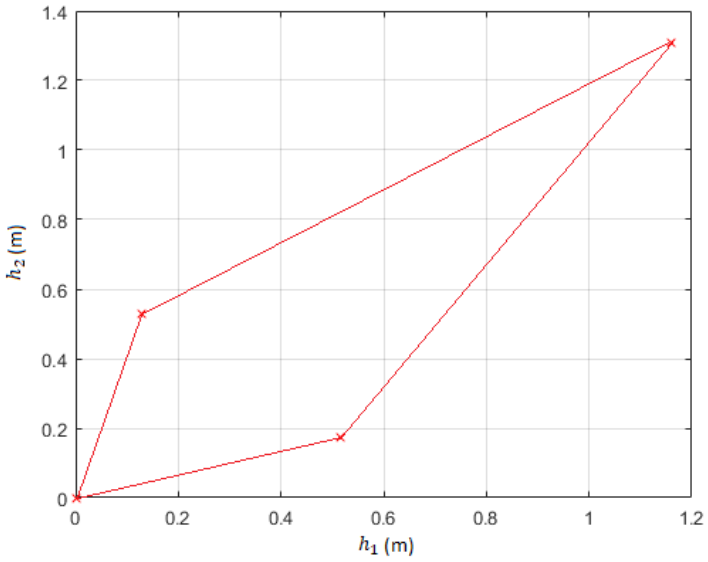


Figura 5.2 Referencias factibles.

$$\frac{dh_1}{dt} = 0$$
$$\frac{dh_2}{dt} = 0$$
$$\frac{dh_3}{dt} = 0$$
$$\frac{dh_4}{dt} = 0$$

Para todas las combinaciones $q_a = q_a^{max}$, $q_a = q_a^{min}$, $q_b = q_b^{max}$ y $q_b = q_b^{min}$. Esto proporciona 4 puntos en un plano de $\mathbb{R}^2$, lo que se traduce en un recinto como el expuesto anteriormente.

5.2.2 Algoritmo del modelo

Una vez tenemos la base de datos completa tenemos que preguntarnos cómo vamos a realizar el cálculo del estado posterior. En este momento sería interesante mencionar algunas posibles ventajas e inconvenientes respecto a otras posibilidades:

- Ventajas
  - Nos estamos quitando el trabajo de tener que realizar un modelo basado en ecuaciones diferenciales, lo cual, para sistemas con dinámica muy compleja, puede resultar engorroso o acabar en un modelo demasiado sencillo sujeto a muchos errores y restricciones.
  - A diferencias de otras técnicas que necesitan linealizar el modelo en diferentes puntos de operación, nosotros tenemos una única base de datos para cualquier estado dentro del conjunto factible.
- Inconvenientes
  - Necesitamos obtener una base de datos fiable, para lo que hemos necesitado previamente haber diseñado controladores estables de alguna manera.
  - Es indispensable poseer suficiente información puesto que, de lo contrario, la predicción estará sujeta a grandes errores. Por otro lado, aumentar en exceso el tamaño de la base de datos incrementa en gran medida la carga computacional del problema e implica adoptar soluciones específicas como la nuestra, una GPU.

Una vez dicho esto, explicamos los distintos pasos definidos por el algoritmo para cumplir nuestro objetivo.

Búsqueda de candidatos

En primer lugar, calculamos la distancia del estado actual con cada una de las filas de la base de datos. El concepto expresado anteriormente se refiere a la distancia euclídea en $\mathbb{R}^6$ de la manera $\tilde{x}_k = [x_k, u_k]$. Es decir, comparando el estado actual y los valores de las actuaciones en ese mismo instante. Sin embargo, esta información todavía necesita ser procesada para poder usarla posteriormente:

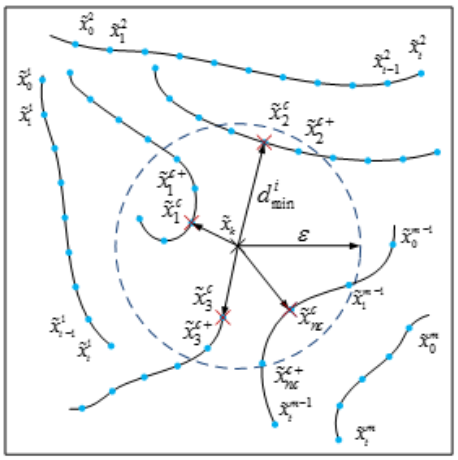


Figura 5.3 Simplificación en $\mathbb{R}^2$ de la búsqueda de estados candidatos [12].

- Antes de nada, buscamos el valor de la mínima distancia dentro de cada una de las trayectorias. Una vez hecho esto, tendremos un vector que contiene los valores mínimos para cada trayectoria concreta. Por otro lado, almacenaremos los índices que apuntan a esos elementos concretos en la base de datos.
- Para realizar la predicción del estado posterior es necesario que los puntos actuales no sean el último disponible en su trayectoria, puesto que no tendríamos información para predecir. Esto nos obliga a que descartemos estos puntos, quedándonos en todo caso con otro distinto, aunque no sea el de menor distancia.
- Dada una esfera $\mathbb{R}^6$ de radio ε, rechazaremos los puntos que no se encuentren dentro del conjunto.

Puede verse una representación gráfica del problema de búsqueda en la figura 5.3.

Búsqueda de envoltura convexa

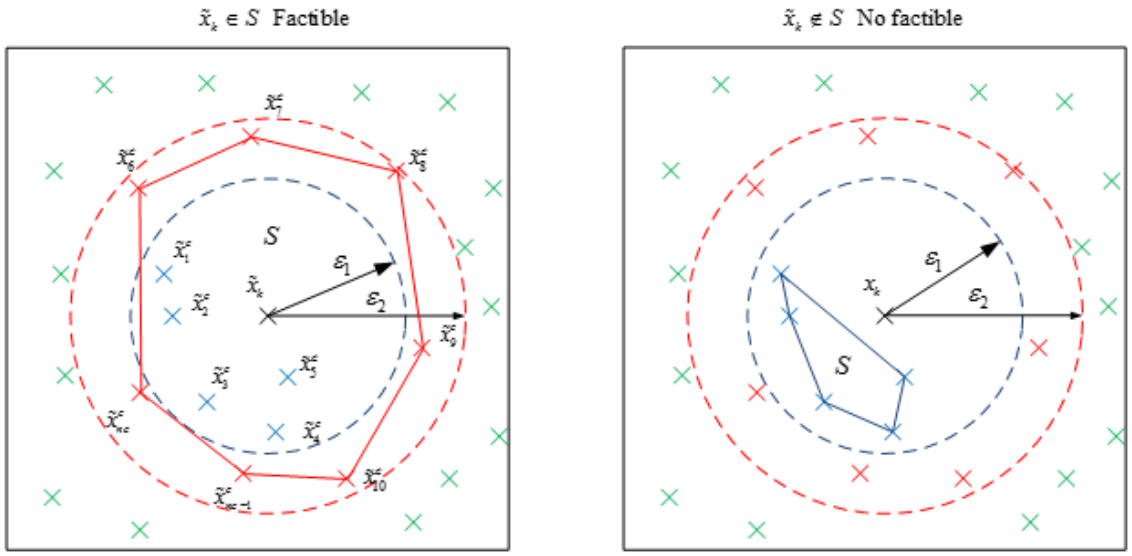
Una vez hecho lo anterior, necesitamos encontrar un conjunto de estados dentro de la base de datos que formen una envoltura convexa que contenga a nuestro estado actual. De esta manera, diremos que $\tilde{x}_k \in S$

cuando esto ocurra, siendo S la envoltura antes mencionada y $\tilde{x}_k = [x_k, u_k]$.

Para que nos vayamos haciendo una idea, la envoltura convexa implicará que se cumplen una serie de restricciones de igualdad y desigualdad las cuáles expondremos más adelante.

A esta serie de puntos que forman una envoltura convexa S (calculados en el paso anterior) los denominaremos estados candidatos $x_i^c \; \forall i = 1,...,nc$ siendo nc el número de estados candidatos, que estará limitado por el número de trayectorias de la base de datos.

Ahora entramos en un proceso iterativo en el cual buscamos obtener la envoltura S con el menor número posible de candidatos (ver figura 6.14), con el objetivo de no aumentar la carga computacional en el problema de optimización.



(a) $\tilde{x}_k \in S$. **(b)** $\tilde{x}_k \notin S$.

Figura 5.4 Simplificación en $\mathbb{R}^2$ de la búsqueda de envoltura convexa [12].

De esta manera, variando el valor de ε, aumentaremos o disminuiremos la cantidad de estados candidatos, puesto que cambiarán los puntos que quedan dentro o fuera de la esfera.

El método elegido para incrementar o decrementar el valor de ε es el método de la bisectriz, definiéndose:

- Partiendo de la resolución del problema para un ε_0 , se nos plantean dos posibilidades:
  - Obtenemos una solución factible. Esto significa que puede existir una solución mejor con menor cantidad de puntos, por lo que desplazamos ε de la manera que se muestra en la figura 5.5.

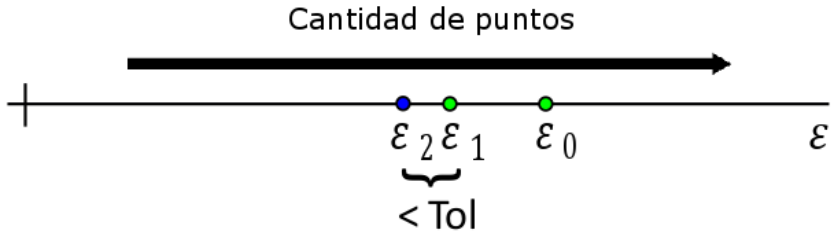


Figura 5.5 Ejemplo de evolución de epsilon. Los puntos verdes indican que se encontró envoltura convexa. Los puntos azules indican el estado actual, por tanto aún desconocido.

  - No se encontró envoltura convexa. Esto implica que tengo que aumentar el valor de ε para poder incluir más puntos en el rango. Para ello, duplicamos su valor.

- Si estoy en una iteración distinta de la primera, existen 4 posibilidades:
  - Si he encontrado una envoltura convexa y en la iteración anterior se dio el mismo caso, seguiré resolviendo el problema hasta que $\varepsilon_{k-1} - \varepsilon_k \leq Tol$, tal como se veía en la figura 5.5.
  - Si no he sido capaz de formar un conjunto S que contenga a $\tilde{x}_k$ y sí lo conseguí en el instante anterior, significa que he reducido demasiado la cantidad de puntos de la región. Por tanto, para volver a conseguir una envoltura convexa, necesito avanzar en la dirección contraria tal como se ve en la figura 5.6.

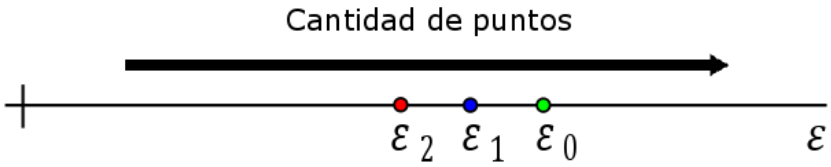


Figura 5.6 Ejemplo de evolución de epsilon. Los puntos rojos indican que no se encontró envoltura convexa, al contrario de los verdes. Los puntos azules indican el estado actual, por tanto aún desconocido.

  - Si anteriormente no fuimos capaces de formar una envoltura convexa y en este instante tampoco, seguiremos duplicando el valor de ε hasta que esto suceda. Esto se ve limitado por otra especificación del algoritmo, ε_{max} , la cual especifica el radio máximo de la esfera de $\mathbb{R}^6$ que contiene a los estados. Todo esto se ve reflejado en la figura 5.7.

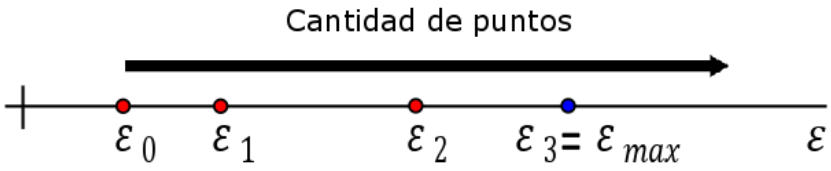


Figura 5.7 Ejemplo de evolución de epsilon. El valor de ε_{max} supone una barrera para el aumento de ε.

  - Si es la primera vez que encuentro una región convexa, volvemos al primer caso.

Sin embargo, seguimos teniendo algunos problemas:

- Existen puntos para los que no es posible encontrar una envoltura convexa. Esto puede darse cuando trabajamos con bases de datos demasiado pequeñas o estamos en puntos muy cercanos a los límites del espacio de trabajo del sistema. Para estos casos especiales, se propondrá una alternativa distinta a las restricciones del problema de optimización.
- Como ya hemos dicho anteriormente, no nos interesa tener una gran cantidad de candidatos puesto que complican innecesariamente el problema de optimización. Por tanto, decidimos establecer una cantidad de puntos máximo, lo cual rompe un poco la estructura del algoritmo anterior. Sin embargo, nos basta saber que nos proporcionará un mejor rendimiento y un mejor resultado. Esto tiene sentido debido a que una cantidad excesiva de estados candidatos es muy probable que incluya puntos que se encuentran relativamente lejos del estado actual, lo que significa que probablemente no aporten información satisfactoria. Además, aumentan el tamaño del problema a resolver, lo que implica una gran ralentización del proceso de predicción.

Resolución del problema de optimización

El objetivo de obtener un conjunto S que formara una envoltura convexa (que aún no hemos definido) es poder resolver un problema de minimización con el que obtener unos pesos (a partir de ahora α) que nos

permitirán calcular una predicción del estado siguiente.

Siendo $\tilde{x}_i^{c+}$ el estado siguiente al correspondiente $\tilde{x}_i^c$, buscamos estimar el estado tal como se ve reflejado en las ecuaciones 5.9.

$$\tilde{x}_{k+1} = \sum_{i=1}^{nc} \alpha_i \cdot \tilde{x}_i^{c+} \tag{5.9}$$

En función de los resultados obtenidos en apartados anteriores, seguimos un camino u otro:

- Hemos conseguido salir del bucle encontrando una envoltura convexa. En ese caso resolveremos el problema con las restricciones genuinas, tal como se ve en las ecuaciones 5.10.

$$\begin{aligned} &\text{Minimizar} \sum_{i=1}^{nc} \alpha_i^2 \\ &\text{Sujeto a} \\ &\sum_{i=1}^{nc} \alpha_i = 1 \\ &\sum_{i=1}^{nc} \alpha_i \tilde{x}_i^c = \tilde{x}_k \quad \forall i = 1,...,nc \\ &\alpha_i \geq 0 \quad \forall i = 1,...,nc \end{aligned} \tag{5.10}$$

- Si por el contrario, no fuimos capaces de encontrar una envoltura convexa para $\varepsilon = \varepsilon_{max}$ o bien para $N < N_{max}$, tendremos que liberar alguna restricción para que sea posible obtener alguna solución. En concreto, permitimos que los α_i puedan tener valores ≤ 0 , quedando el problema definido como se refleja en la ecuación 5.11.

$$\begin{aligned} &\text{Minimizar} \sum_{i=1}^{nc} \alpha_i^2 \\ &\text{Sujeto a} \\ &\sum_{i=1}^{nc} \alpha_i = 1 \\ &\sum_{i=1}^{nc} \alpha_i \tilde{x}_i^c = \tilde{x}_k \quad \forall i = 1,...,nc \\ &-1 \leq \alpha_i \leq 1 \quad \forall i = 1,...,nc \end{aligned} \tag{5.11}$$

Una vez calculados los pesos, es posible predecir el estado posterior usando la ecuación 5.9. Sin embargo, antes de continuar, vamos a detenernos en diversas posibilidades para la resolución de este problema que acabamos de plantear.

Programación cuadrática

Existen multitud de algoritmos dedicados a resolver problemas de optimización. Dependiendo del software que usemos, tendremos unas posibilidades u otras. En nuestro caso, al estar tratando de realizar la mayor cantidad posible de implementaciones en la GPU, nos planteamos la posibilidad de resolver también este problema en su interior, además de los anteriormente expuestos.

Por ello, nos basamos en la literatura que hay actualmente sobre algoritmos de programación cuadrática que fueran susceptibles de ser paralelizables. De esta manera, encontramos información en diversos artículos como ”A Parallel Quadratic Programming Algorithm for Model Predictive Control” [13], ”Parallel Quadratic Programming for Image Processing” [14] y ”Projection-free Parallel Quadratic Programming for Linear

Model Predictive Control" [15].

En ellos se desarrolla un algoritmo completamente paralelizable que se puede programar en la GPU. En algunos de los trabajos anteriores se usa para resolver los problemas que aparecen en las técnicas de control predictivo basado en modelo, mientras que en otros, tiene como finalidad la mejora computacional de programas de tratamiento de imágenes.

Partiendo de un problema de optimización como el 5.12, tenemos unas variables de decisión U_k con una matriz Q que multiplica a los términos cuadráticos y otra H para los lineales. Tenemos a su vez una serie de restricciones de desigualdad representadas por las matrices V y W.

$$\begin{aligned} &\text{Minimizar} \quad \frac{1}{2}U^T QU + H^T U \\ &\text{Sujeto a} \\ &\qquad VU < W \end{aligned} \tag{5.12}$$

Sin embargo, el algoritmo no es capaz de trabajar directamente con problemas expresados de esta manera. Por tanto, es necesario transformar las ecuaciones anteriores a su forma dual, obteniendo como resultado el problema 5.13.

$$\begin{aligned} &\text{Minimizar} \quad \frac{1}{2}y^T qy + h^T y \\ &\text{Sujeto a} \\ &\qquad y > 0 \end{aligned} \tag{5.13}$$

Las nuevas matrices q y h quedan definidas por las ecuaciones 5.14 y 5.15.

$$q = VQ^{-1}V^T \tag{5.14}$$

$$h = W + VQ^{-1}H \tag{5.15}$$

Una vez resuelto el problema, es necesario calcular la solución U^* a partir del óptimo y^* siguiendo la expresión 5.16.

$$U^* = -Q^{-1}(H + V^T y^*) \tag{5.16}$$

Sin embargo, todavía no podemos aplicar el algoritmo a nuestro problema concreto. Esto es debido a que no hemos tenido en cuenta las restricciones de igualdad. Se plantearon en un principio diversas posibilidades como el uso de "*penalty functions*". Finalmente, se decidió despejar un conjunto de variables igual al número de restricciones, de manera que se obtiene un sistema compatible indeterminado. Para resolver este sistema, se hace uso de la factorización QR [16]. Antes de presentar los pasos que se han de dar, conviene mostrar qué representa cada variable.

$$\begin{aligned} A &= \text{Matriz que aglutina las restricciones de desigualdad} \\ b &= \text{Vector que contiene el valor de las desigualdades} \\ Q, R &= \text{Factorización QR de la matriz } A^T \\ p &= \text{Número de restricciones a eliminar} \end{aligned}$$

Las otras matrices Q_1 y Q_2 provienen de la misma matriz Q. Esta subdivisión es dependiente del número de restricciones de igualdad que queramos transformar.

Dicho esto, se procede de la siguiente manera:

- Se calcula la factorización QR. Eso implica que hemos obtenido Q y R.
- Dividimos las matrices de la siguiente manera:

$$Q_1^T Q_1 = I \quad \text{Siendo } Q_1 \text{ una matriz que contiene las primeras } p \text{ columnas de la matriz } Q.$$
$$Q_2^T Q_2 = I \quad \text{Siendo } Q_2 \text{ una matriz que contiene el resto de columnas de la matriz } Q.$$

  Se cumple asimismo que $Q_1^T Q_2 = 0$. Esto puede servir para comprobar que las operaciones se han realizado correctamente.

- Una vez hecho esto, es posible calcular una de las infinitas soluciones del sistema con la expresión:

$$\hat{x} = Q_1 R^{-T} b$$

- LLegados a este punto, es conveniente recordar el problema 5.12. Vamos a nombrar a partir de ahora Q_{eq} y H_{eq} a las matrices que resultan de la eliminación de las restricciones de igualdad. Es posible calcularlas a partir de las expresiones siguientes:

$$Q_{eq} = Q_2^T Q Q_2$$
$$h_{eq} = (h^T + \hat{x}^T Q) Q_2$$

- Además, es necesario modificar las restricciones de desigualdad, puesto que también se ven afectadas por la transformación que acabamos de realizar. Por tanto, siguiendo el mismo criterio anterior, definiremos V_{eq} y W_{eq} de la siguiente manera.

$$V_{eq} = -Q_2$$
$$W_{eq} = \hat{x} + W$$

- Por último, una vez se hubiera resuelto el problema, es necesario revertir la transformación (de manera parecida a lo que ocurría con el problema dual). Para ello, simplemente hacemos uso de:

$$U^* = Q_2 x^* + \hat{x}$$

Sin embargo, nuestro problema original (5.10) posee tanto restricciones de igualdad como de desigualdad. Por tanto, necesitamos aplicar en primer lugar la eliminación de las primeras y, una vez hecho esto, convertirlo al problema dual. Esto va a requerir de igual manera cuando obtengamos la solución que haya que revertir estas transformaciones. Conviene tener en mente todas las operaciones que hemos tenido que realizar para llegar a este punto, puesto que suponen un punto clave a la hora de la resolución que hemos tomado.

El algoritmo que pasamos a desarrollar es llamado *Parallel Quadratic Programming* y es comúnmente conocido por sus siglas (PQP). Sus iteraciones están basadas en la realización de operaciones matriciales sencillas las cuáles pueden ser fácilmente paralelizables. Esto no quiere decir que únicamente se obtenga beneficio en el caso de la implementación en GPU, sino que es capaz de equipararse a muy diversos algoritmos y *solvers* comerciales [13]. Los pasos a seguir por el algoritmo son los siguientes:

1. A partir de la formulación genérica de programación cuadrática, eliminar las restricciones de igualdad usando, por ejemplo, la factorización QR.

2. Una vez definido este nuevo problema derivado del anterior, reescribirlo a su forma dual correspondiente.

3. Empezando con un valor inicial $y^0 > 0$, aplicar la siguiente actualización hasta alcanzar la tolerancia especificada.

$$y_i \leftarrow y_i \left[\frac{h_i^- + (Q^- y)_i}{h_i^+ + (Q^+ y)_i} \right], \tag{5.17}$$

Siendo:

$$\begin{aligned} q^+ &= max(q,0) + diag(r) \\ q^- &= max(-q,0) + diag(r) \\ h^+ &= max(h,0) \\ h^- &= max(-h,0) \end{aligned}$$

No se ofrece mucha información en los artículos anteriores respecto al vector r. Nos dicen que han comprobado que funciona para valores cercanos a 0, pero no especifican cuál es exactamente su función o cómo afecta al algoritmo cambiar su valor.

Si nos fijamos bien en la ecuación 5.17, podemos ver que dentro de los corchetes se obtendría tanto arriba como abajo un vector. En realidad, esta división se realiza elemento a elemento, posteriormente actualizando a la variable de decisión correspondiente. Por esta razón es posible paralelizar el algoritmo, porque la actualización de cada una de las variables de decisión se realiza por separado.

Se planteó más adelante una posible mejora para el rendimiento del algoritmo invocando una actualización distinta cada cierto número de ejecuciones [15]. Sin embargo, no resultaba ser muy estable en algunas circunstancias. En algunos casos conseguía mejorar los tiempos mientras que en otros los empeoraba.

4. Revertir la solución a su forma primal. Una vez hecho esto, devolverlo a su forma primera. Es importante seguir el orden aquí especificado. Esto es debido a que hay que ser congruente con el método que se siguió a la hora de transformarlos en primer lugar. Es fácil darse cuenta de que se sigue el orden contrario al que se siguió a la hora de la conversión (figura 5.8).

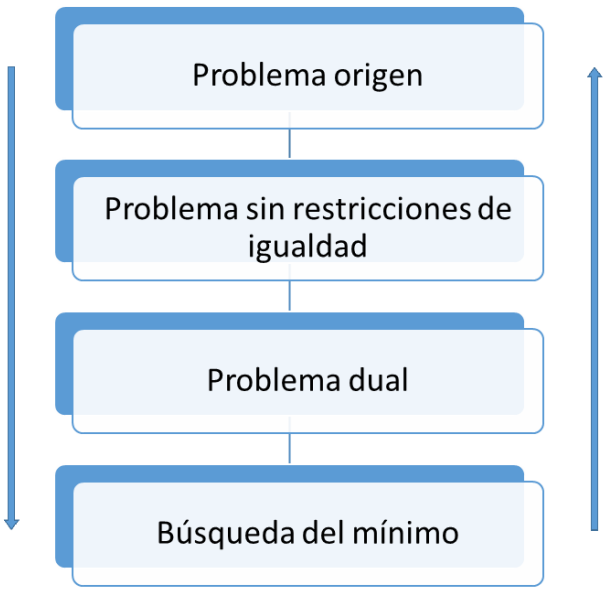


Figura 5.8 Conversión de restricciones del problema de optimización.

Como podemos ver, la carga computacional del algoritmo PQP es realmente pequeña. Sin embargo, nuestro problema concreto necesita pasar por 2 transformaciones consecutivas antes de poder resolverse. Además, algunas de las operaciones involucradas conllevan inversiones de matrices, las cuáles suelen ser computacionalmente costosas y en nuestro caso tendrían que realizarse para cada iteración del método de la bisectriz.

Por tanto, al tratar de comparar su rendimiento con el *solver* de Matlab (*quadprog*), hemos preferido este último por diversas razones:

- Quadprog resulta ser mucho más maduro que el algoritmo anteriormente presentado y, por tanto, no diverge para algunos casos patológicos como ocurría con el PQP.
- A pesar de que el PQP era generalmente más rápido que quadprog, las operaciones necesarias para devolver los resultados incrementaban este tiempo por encima del obtenido directamente con el *solver* de Matlab.
- Aunque quizá no parezca una razón al uso, el código se vuelve mucho más limpio y no es necesario generar una serie de variables auxiliares que estarían consumiendo memoria en la GPU.

Aún así, existen problemas en los que nos podríamos beneficiar de las limitaciones del PQP, como aquellos en los que la conversión del problema dual pueda realizarse fuera de línea debido a que permanece invariable. En esos casos, no tendríamos la carga computacional asociada a las inversiones de matrices sino únicamente la ejecución del algoritmo que, como hemos dicho antes, suele ser más rápida que *quadprog*.

5.2.3 Resultados

De igual manera que la bondad de los resultados que se obtienen al modelar un determinado sistema a través de una función de transferencia depende de factores muy diversos como el orden del modelo, las dinámicas que no se tienen en cuenta por simplicidad, etc, existen en nuestro caso factores que favorecen o empeoran el comportamiento. Algunos de estos parámetros serán:

- En primer lugar, como es obvio, la cantidad de información que tenemos en la base de datos. Es evidente la razón por la que esto es así, no es posible predecir el estado de un determinado sistema si no conozco mínimamente como ha evolucionado anteriormente. Además de la cantidad, es importante la selección que hemos hecho, es decir, qué información hemos decidido almacenar. Se puede dar el caso de tener una base de datos enorme, pero que no está muestreada correctamente (trayectorias repetidas o redundantes o mala selección de las trayectorias a almacenar en general), de manera que no es capaz de predecir adecuadamente.
- El ε_{max}. Si no existiera N_{max}, ε_{max} afectaría muchísimo al rendimiento al aumentar su valor, puesto que no habría límite de estados candidatos. Diciéndolo de esta manera parece que no tendría sentido tener un valor grande de ε_{max}, esto no siempre es así. En función de la zona en la que estemos trabajando y de como esté construida la base de datos, podemos encontrarnos para una misma trayectoria con que para un mismo valor de ε_{max} y distintos instantes de tiempo, tenemos algunos casos en los que no se encuentra ningún candidato (posible en los límites del espacio de trabajo) mientras que en otros se supera el máximo (esto ocurre sobre todo cuando nos acercamos al régimen permanente).
- N_{max}. Se encarga principalmente de impedir que se acumulen demasiados estados candidatos, de manera que impide que se pueda perder demasiado tiempo a la hora de resolver el problema de programación cuadrática. Se complementa bien con ε_{max}, puesto que ahora nos podemos permitir aumentar su valor sin miedo a que la ejecución se haga muy lenta.

Para comprobar el funcionamiento del modelo, se han generado trayectorias que parten de estados iniciales aleatorios con referencias igualmente aleatorias.

En las figuras 5.9 y 5.12 puede verse el funcionamiento del modelo con una base de datos suficientemente amplia. El error de la predicción es realmente pequeño, por lo que se puede combinar con el uso de sensores industriales sin problema.

El funcionamiento para el caso de la figura 5.10 resulta hasta sorprendente. Hemos reducido la cantidad de trayectorias hasta 1000, es decir, 100 veces menos que en el anterior. Aún así, seguimos obteniendo resultados

bastante satisfactorios.

Se ha realizado además la predicción de un conjunto de 1000 trayectorias para los dos conjuntos de bases de datos. La comparativa de errores queda reflejada en la tabla 5.2.

Tabla 5.2 Comparativa de la bondad para diferentes tamaños de la base de datos.

Nº trayectorias	Error máximo	Error mínimo	Error medio
1000	0.0336	$2.8228 \cdot 10^{-6}$	0.0019
100000	0.0195	0	$4.336 \cdot 10^{-4}$

Como era de esperar, los resultados son mejores para una base de datos más grande. Sin embargo, para la precisión que tienen gran cantidad de sensores industriales, el conjunto de 1000 trayectorias funcionaría sin problemas.

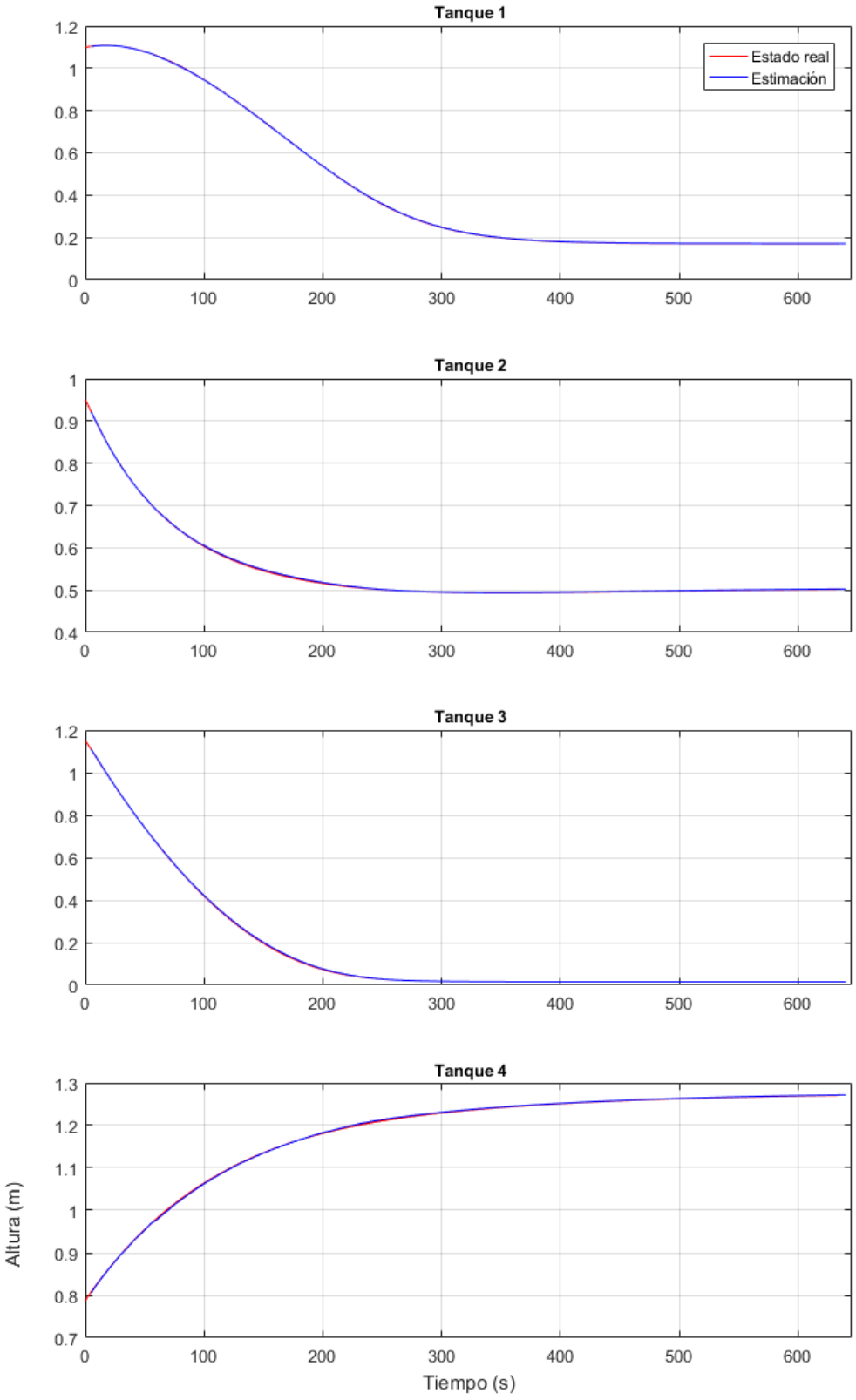


Figura 5.9 Modelo con una base de datos que almacena 100000 trayectorias. Ejemplo sin ruido.

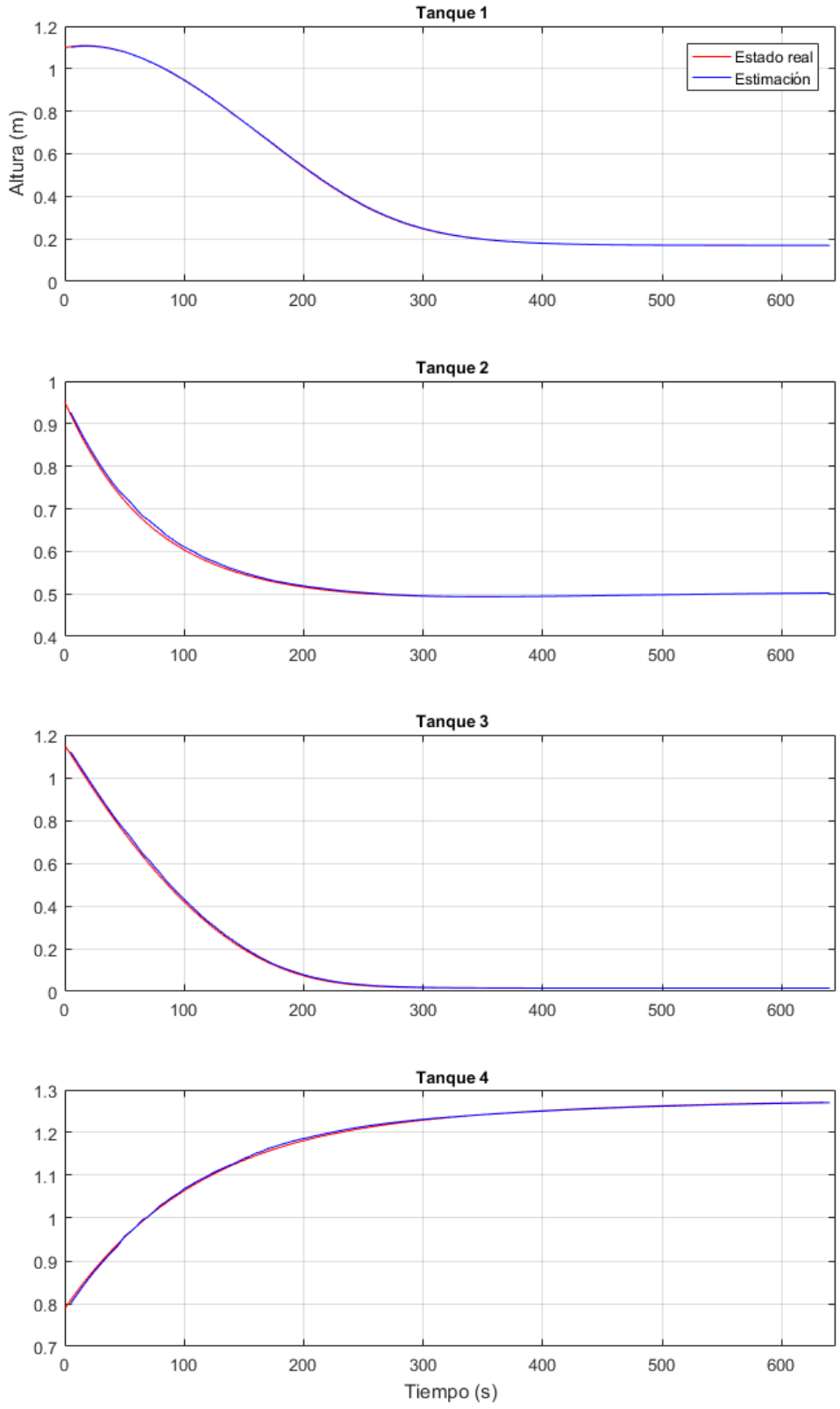


Figura 5.10 Modelo con una base de datos que almacena 1000 trayectorias. Ejemplo sin ruido.

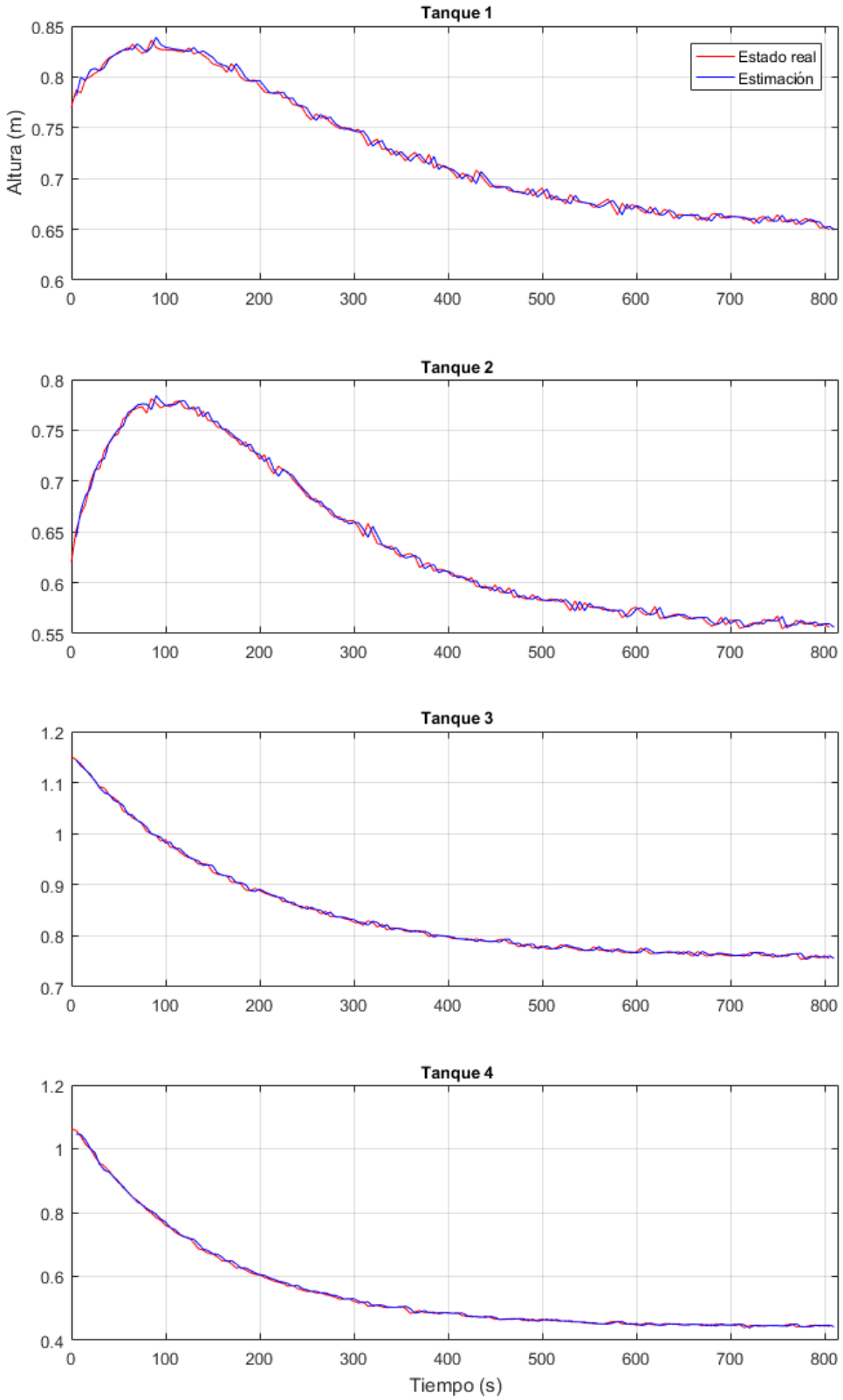


Figura 5.11 Modelo con una base de datos que almacena 1000 trayectorias. Ejemplo con ruido.

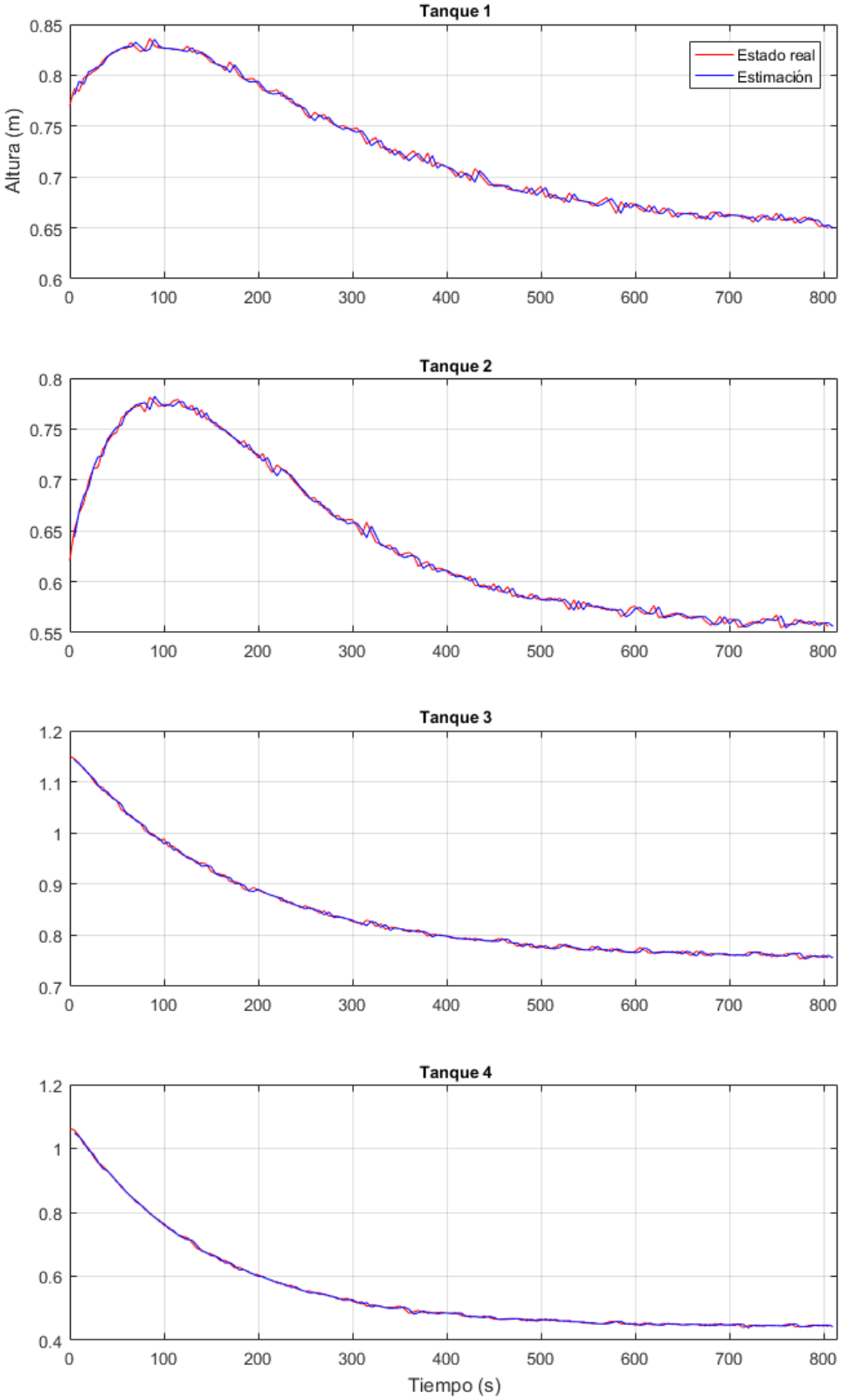


Figura 5.12 Modelo con una base de datos que almacena 100000 trayectorias. Ejemplo con ruido.

6 Control predictivo

Antes de hablar de nuestra propuesta, haremos una breve reseña histórica además de una pequeña explicación general sobre en qué consiste la estrategia de control que vamos a aplicar.

6.1 Estado del arte

El control predictivo surge a finales de los años 70 cuando empiezan a usarse modelos de sistemas para predecir los efectos de las acciones de control en las salidas futuras, las cuáles se determinaban minimizando el error predicho con unas determinadas restricciones de operación.
En cada instante de muestreo, la optimización se repetía con la información actual del proceso. La mayoría de las formulaciones eran de naturaleza heurística e intentaban aprovechar el creciente potencial de los computadores de la época.

Rápidamente, el MPC (*Model Predictive Control*) adquirió gran popularidad, sobretodo entre las industrias de procesos químicos. Esto fue debido a la simplicidad del algoritmo y al uso de determinados modelos que resultaban más intuitivos y necesitaban menos información para identificar.

En realidad, el control predictivo no es propiamente una estrategia de control específica, sino que aglutina un campo muy amplio de métodos desarrollados en torno a ideas comunes. Resumidamente, estas ideas pueden expresarse como:

- Uso de algún tipo de información para predecir la salida del proceso en instantes futuros de tiempo. Como hemos comentado anteriormente, esto se viene haciendo tradicionalmente con el uso de modelos matemáticos. Uno de los objetivos de nuestra propuesta es el uso de una base de datos que sustituye al modelo del sistema, como explicamos en el capítulo anterior.
- Cálculo de las acciones de control minimizando una cierta función objetivo.
- Estrategia de horizonte deslizante, de manera que en cada instante el horizonte se va desplazando hacia el futuro, aplicando la acción actual y desechando el resto. Por tanto, es necesario repetir este cálculo en cada instante de muestreo.

La mayoría de los algoritmos suelen diferir en los modelos utilizados y en las funciones objetivo que se desean minimizar.

Por otro lado, el control predictivo presenta una serie de ventajas interesantes respecto a otros métodos, entre las que podemos destacar:

- Resulta atractivo para personal sin gran conocimiento de control debido a que los conceptos son intuitivos.
- Puede usarse para controlar gran cantidad de sistemas, desde aquellos con dinámica simple a otros mucho más complejos incluyendo sistemas con retardo, de fase no mínima o inestables. En concreto, es fácil darse cuenta de que posee una compensación intrínseca del retardo.
- Permite tratar con facilidad el caso multivariable.

- La adición de restricciones adicionales al controlador resulta especialmente sencillo.
- También es especialmente útil cuando se conocen las futuras referencias, lo cual sucede en diversos campos como, por ejemplo, la robótica.
- Se trata de una metodología abierta únicamente basada en algunos principios básicos, pero permite gran cantidad de extensiones.

Sin embargo, también presenta diversos inconvenientes como:

- La carga de cálculo necesaria para la resolución de algunos algoritmos. En nuestro caso, al estar basado en datos, esto supone una gran limitación cuando se usan bases de datos suficientemente grandes. Más adelante veremos cómo la GPU es capaz de sobreponerse en estos casos.
- La necesidad de un modelo (o, en nuestro caso, de una base de datos) apropiado del proceso, puesto que nuestras predicciones las haremos a partir de él y, en caso de tenga gran cantidad de errores, es fácil darse cuenta de que no estaremos prediciendo correctamente, lo que llevará a que no tendremos un buen control.

A grandes rasgos, el funcionamiento de los controladores predictivos es el siguiente:

- En cada instante t, haciendo uso de la información que disponemos (modelo, base de datos, etc) se predicen las salidas futuras hasta un tiempo N, denominado horizonte de predicción (figura 6.1). Estas predicciones dependen de los valores conocidos hasta el instante t y de las señales de control futuras, por lo que se denominan de la siguiente manera:

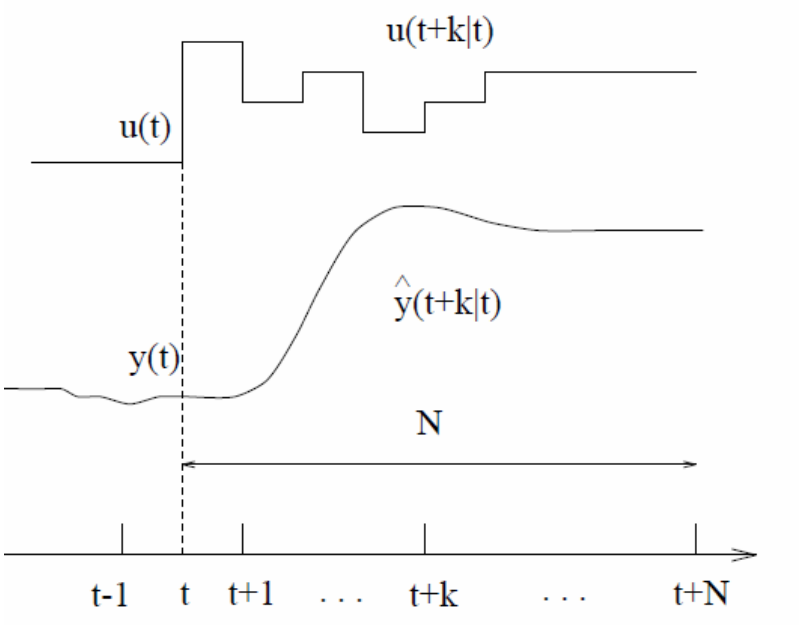


Figura 6.1 Estrategia de los controladores predictivos.

$$\hat{y}(t+k\,|\,t) \quad \text{Para } k = 1\ldots N \tag{6.1}$$
$$u(t+k\,|\,t) \quad \text{Para } k = 1\ldots N-1 \tag{6.2}$$

Es decir, $(t+k\,|\,t)$ significa los valores en el instante $t+k$ calculados en el instante t.

- El conjunto de acciones de control futuras (que es lo que nos interesa obtener) se calcula optimizando una determinada función de costes. Esta puede tener muy diversas formas. Sin embargo, lo más común es que se trate de una función cuadrática que posee dos términos: uno relativo al error y otro al coste de la acción de control.

- Por último, la señal de control $u(t\,|\,t)$ es enviada al sistema, mientras que el resto son desechadas. De esta manera, en el siguiente instante de muestreo haremos uso de la información obtenida en el instante $t+1$ para calcular nuevamente la secuencia óptima de acciones de control.

En resumen, el esquema de un controlador predictivo puede verse en la figura 6.2.

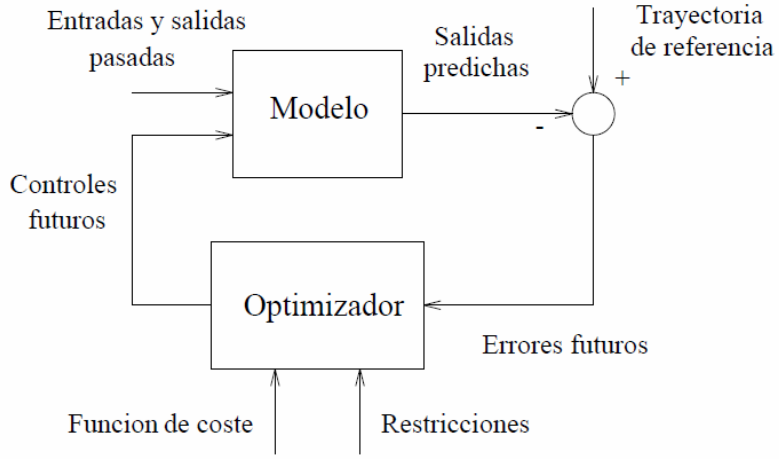


Figura 6.2 Esquema MPC.

6.2 Controladores predictivos basados en datos

La principal diferencia del controlador propuesto consiste en la manera de guardar la información. En lugar de identificar el sistema con los parámetros de un determinado modelo matemático, guardamos el histórico de las trayectorias llevadas a cabo por el sistema en bucle cerrado para diferentes controladores en una base de datos cuya forma se explicó en el apartado de modelado.

De esta manera, un posible esquema para nuestro controlador se vería reflejado en la figura 6.3.

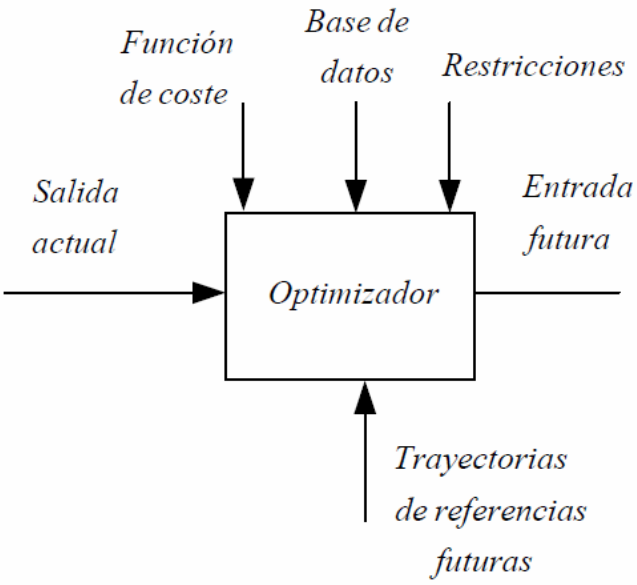


Figura 6.3 Esquema DbPC.

Lo que planteamos a partir de ahora se complementa perfectamente con el modelo visto anteriormente, cuyo algoritmo presenta ciertas similitudes. Partiendo del siguiente modelo no lineal discreto:

$$x_{k+1} = f(x_k, u_k) \tag{6.3}$$

Donde $x_k \in \mathbb{R}^n$ es la salida del sistema y $u_k \in \mathbb{R}^m$ es la entrada en el tiempo de muestreo k respectivamente. f representa el modelo del sistema, el cual se supone desconocido.

Queremos diseñar una ley de control que permita resolver el problema de regulación de manera que podamos partir de un x^0 arbitrario para alcanzar una referencia x^{ref} cualquiera (figura 6.4).

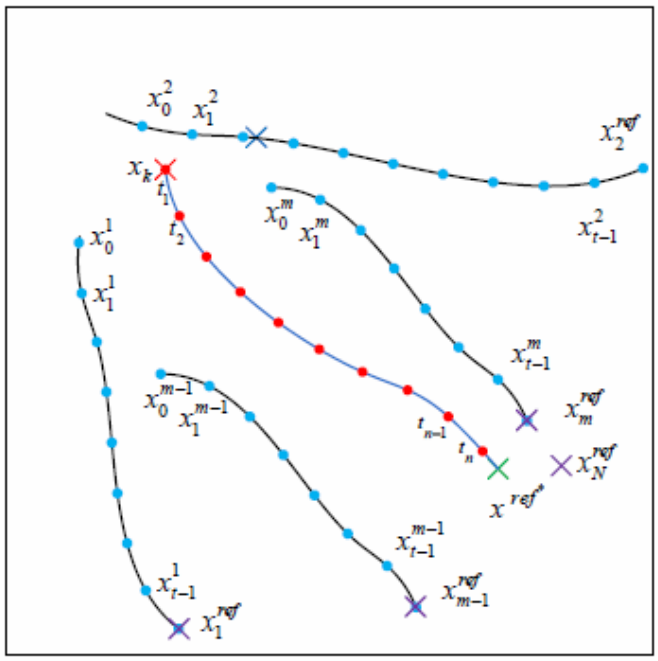


Figura 6.4 Problema de regulación en $\mathbb{R}^2$.

Buscamos hallar la entrada actual u_k de la manera:

$$u_k = DbPC(x_k, x^{ref}, BD, ID) \tag{6.4}$$

Los *ID* (Índices de desempeño) representan la bondad de las trayectorias candidatas y están definidos de la forma siguiente:

$$ID_k = \sum_{i=0}^{N} J_i(h, q, c, p) \tag{6.5}$$

Siendo ID_k el índice de desempeño correspondiente a una trayectoria candidata k, N el horizonte de predicción y J_i el coste de etapa. La definición de J^i viene dada por:

$$J_i(h, q, c, p) = q_{a\,i}^2 + c \cdot q_{b\,i}^2 + p \frac{V_{min}}{A(h_{1\,i} + h_{2\,i})} \quad \forall i = 1, \ldots, N \tag{6.6}$$

Los valores c y p son grados de libertad que nos permiten dar un mayor peso a unos u otros términos.

Estos índices de desempeño no se encuentran almacenados en la base de datos, sino que se calculan en línea. Sin embargo, no se trata de una operación computacionalmente dominante, ni siquiera en la CPU.

Si nos remontamos al apartado anterior donde explicábamos el *MPC*, es fácil darse cuenta de la similitud de lo que suponen estos índices con la función de costes cuya optimización nos resultaba de gran importancia para obtener las secuencias de control óptimas.

6.2.1 Algoritmo de control

A pesar de ser realmente parecido al usado anteriormente para el modelado, conviene pararse brevemente en algunos aspectos concretos del algoritmo. El esquema viene representado en la figura 6.5.

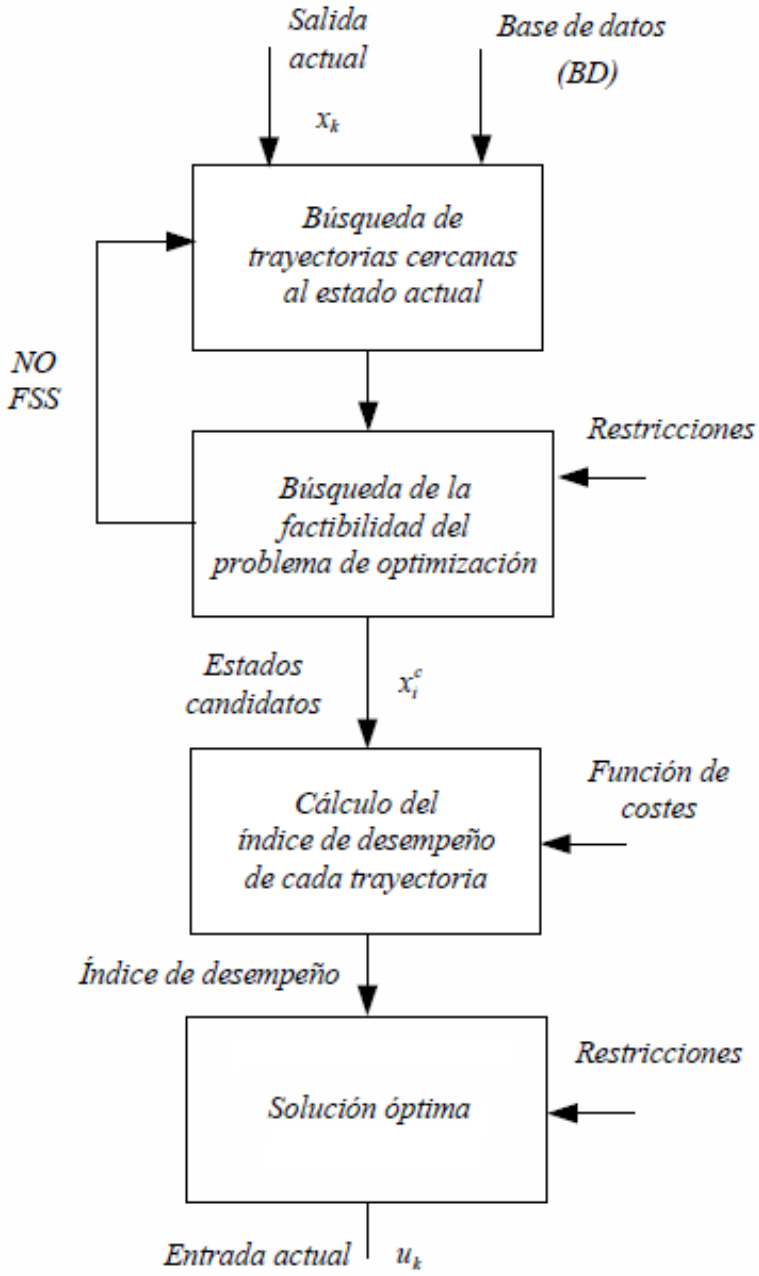


Figura 6.5 Esquema del algoritmo de control basado en datos.

- La búsqueda de estados candidatos es prácticamente la misma con ciertas salvedades. Antes se trataba de una búsqueda en $\mathbb{R}^6$ con un vector del tipo $\tilde{x}_k = [x_k\ , u_k]$. En el caso anterior, se añadían las u_k al

modelo para poder tener información de la tendencia del sistema. Sin embargo, para el control, lo que nos interesa es conocer el punto objetivo, puesto que con el estado actual y las referencias, tenemos toda la información necesaria. Eso quiere decir que tendremos $\tilde{x}_k = [x_k\ , h_{ref}]$.

Por otro lado, antes sólo era necesario tener un estado siguiente para poder predecir. En nuestro caso actual, esto dependerá del horizonte de predicción. Esto quiere decir que ahora la candidatura está más restringida.

- Una vez hemos calculados los candidatos, se comprueba que existe una envoltura convexa. Esto no quiere decir que ya estemos calculando una posible solución al problema de optimización sino que simplemente comprobamos la factibilidad.

Además, las restricciones que venían impuestas anteriormente han cambiado (junto al problema de optimización). Las nuevas restricciones vienen definidas por las siguientes ecuaciones:

$$\sum_{i=1}^{nc} \alpha = 1 \quad \forall i = 1,2,...,nc \tag{6.7}$$

$$\sum_{i=1}^{nc} \alpha_i x_i^c = x_k \quad \forall i = 1,2,...,nc \tag{6.8}$$

$$\sum \alpha_i x_i^{ref} = x^{ref} \quad \forall i = 1,2,...,nc \tag{6.9}$$

$$\alpha_i \geq 0 \quad \forall i = 1,2,...,nc \tag{6.10}$$

A la vista de las ecuaciones anteriores, vemos que el problema se descompone en la búsqueda de dos envolturas: una de $\mathbb{R}^4$ que envuelva las alturas de los tanques y otra de $\mathbb{R}^2$ que cobije a las referencias. Una simplificación gráfica de esto puede verse en la figura 6.6.

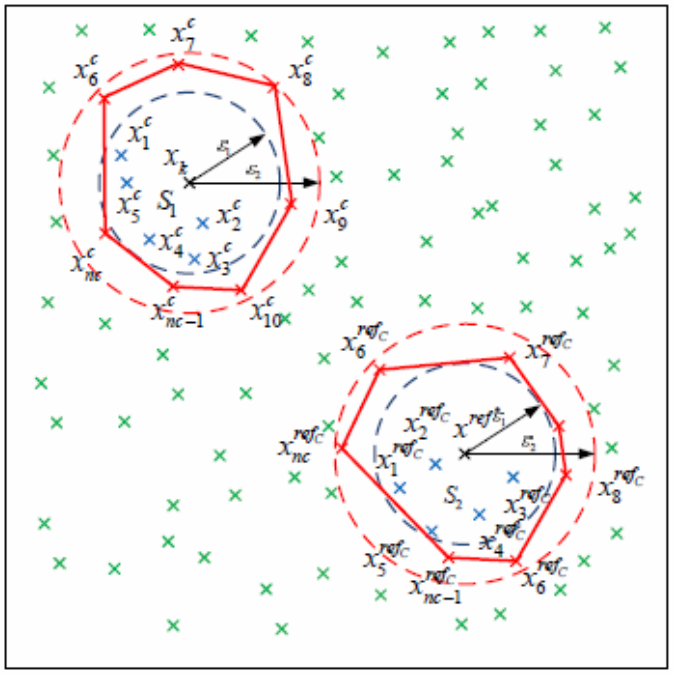


Figura 6.6 Ejemplo simplificado de búsqueda de envoltura convexa para el problema de control. Caso de solución factible encontrada.

Si no se encuentra solución factible al conjunto de ecuaciones anterior (figura 6.7), será necesario repetir la búsqueda anterior con un ε mayor. Además, seguiremos usando el método de la bisectriz

para calcular los valores de ε para la siguiente iteración. De igual manera, aplicamos igualmente los criterios de ε_{max} y N_{max}, con el mismo objetivo que en casos anteriores.

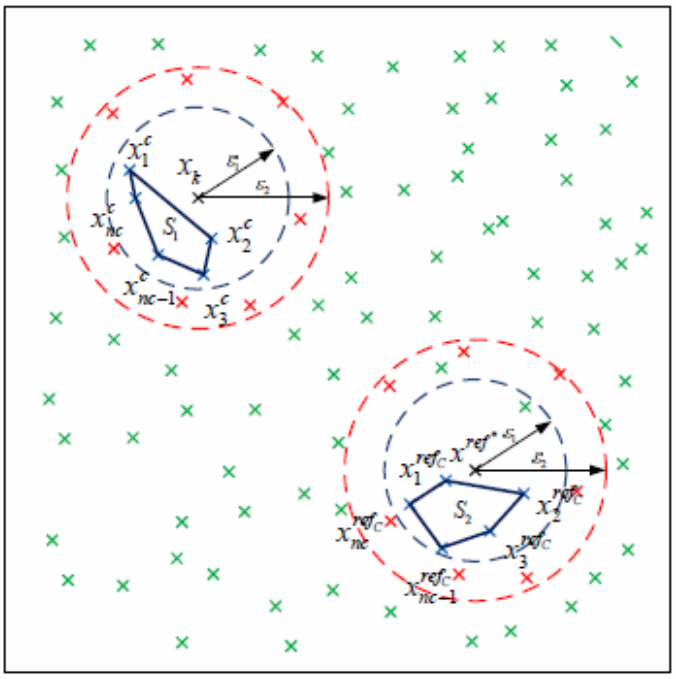


Figura 6.7 Ejemplo simplificado de búsqueda de envoltura convexa para el problema de control. Caso de solución factible no encontrada.

- El paso siguiente es completamente nuevo y ha sido esbozado previamente. Se trata del cálculo de los índices de desempeño. Estos serán parte imprescindible del problema de optimización que es inherente al control predictivo.
- El nuevo problema de minimización viene definido por las ecuaciones siguientes:

$$
\begin{aligned}
&\text{Minimizar} \quad \sum_{i=1}^{nc} \alpha_i ID_i \\
&\text{Sujeto a} \\
&\sum_{i=1}^{nc} \alpha = 1 \quad \forall i = 1,2,...,nc \\
&\sum_{i=1}^{nc} \alpha_i x_i^c = x_k \quad \forall i = 1,2,...,nc \\
&\sum \alpha_i x_i^{ref} = x^{ref} \quad \forall i = 1,2,...,nc \\
&\alpha_i \geq 0 \quad \forall i = 1,2,...,nc
\end{aligned}
\tag{6.11}
$$

Vemos como hemos cambiado el problema anterior por uno de programación lineal donde están incluidos los índices de desempeño que hemos calculado en el paso anterior. De aquí obtendremos los pesos necesarios que nos permitirán calcular la u_k óptima para este instante de muestreo.

Sin embargo, existe un caso que no hemos estudiado, el correspondiente a que no existe una envoltura convexa para cualquier valor de ε que satisfaga $\varepsilon \leq \varepsilon_{max}$ y $N \leq N_{max}$. Para los puntos en los que esto suceda, hemos decidido quedarnos con la acción de control correspondiente al punto más cercano (según la distancia $\mathbb{R}^6$ previamente definida).

6.2.2 Tareas en la GPU

Gran parte del objetivo de este proyecto es aplicar la GPU en posibles problemas de control pero, aún no hemos hablado mucho del papel que esta realiza tanto en el modelo como en el control.

En los algoritmos que hemos presentado anteriormente diferenciamos básicamente los siguientes problemas:

- Búsqueda de candidatos. Este problema incluye el cálculo de la distancia de una nube de puntos a otro y el cálculo de los mínimos de cada trayectoria.
- Optimización de una determinada función para hallar unos pesos α. Incluimos también aquí la comprobación de las restricciones.
- Cálculo de los índices de desempeño.

Resulta claro que la resolución de problemas de minimización no se va a ver recompensada por el uso de la GPU debido a lo que comentábamos del PQP en el capítulo anterior. Además, para que este se viera mejorado, necesitaríamos problemas con gran cantidad de variables de decisión, lo cual no es nuestro caso.

Por otro lado, el cálculo de los índices de desempeño es un problema muy fácilmente paralelizable debido a que el *ID* de cada trayectoria es independiente del de la siguiente. Sin embargo, debido a que la cantidad de *IDs* a calcular no es suficientemente grande y que las operaciones que se realizan son realmente sencillas, no se experimenta mejoría en este aspecto, pero tampoco empeora. Igualmente, en el código final se encuentra programado en la GPU.

Por tanto, es fácil ver que el problema que se va a ver más beneficiado de la paralelización va a ser la búsqueda de los candidatos. Esto va a ser así principalmente por el gran volumen de información que hay que procesar. Es decir, para una base de datos ”pequeña” de 1000 trayectorias, tenemos aproximadamente unas 2 millones de filas. Por tanto, eso supone que hay que calcular la distancia del estado actual a unos 2 millones de puntos. Es fácil darse cuenta de que si esto es así para el caso ”pequeño”, el grande tendrá un volumen mucho mayor, lo que llevará a un mejor rendimiento en comparación a la CPU.

6.2.3 Resultados de simulación

Vamos a estudiar distintas características del controlador en función de los tamaños de las bases de datos, como la evolución de los índices de desempeño, la factibilidad del problema de optimización, el coste computacional y la mejora con respecto a la CPU. Para obtener estos resultados, se han realizado ensayos con las referencias propuestas en el concurso de la CEA del año 2014. Además, se ha implementado sobre un modelo realista de la planta, de manera que presenta ruido en la medida. Las figuras siguientes muestran los resultados de ambos ensayos (con ruido y sin él) para el caso de 100000 trayectorias.

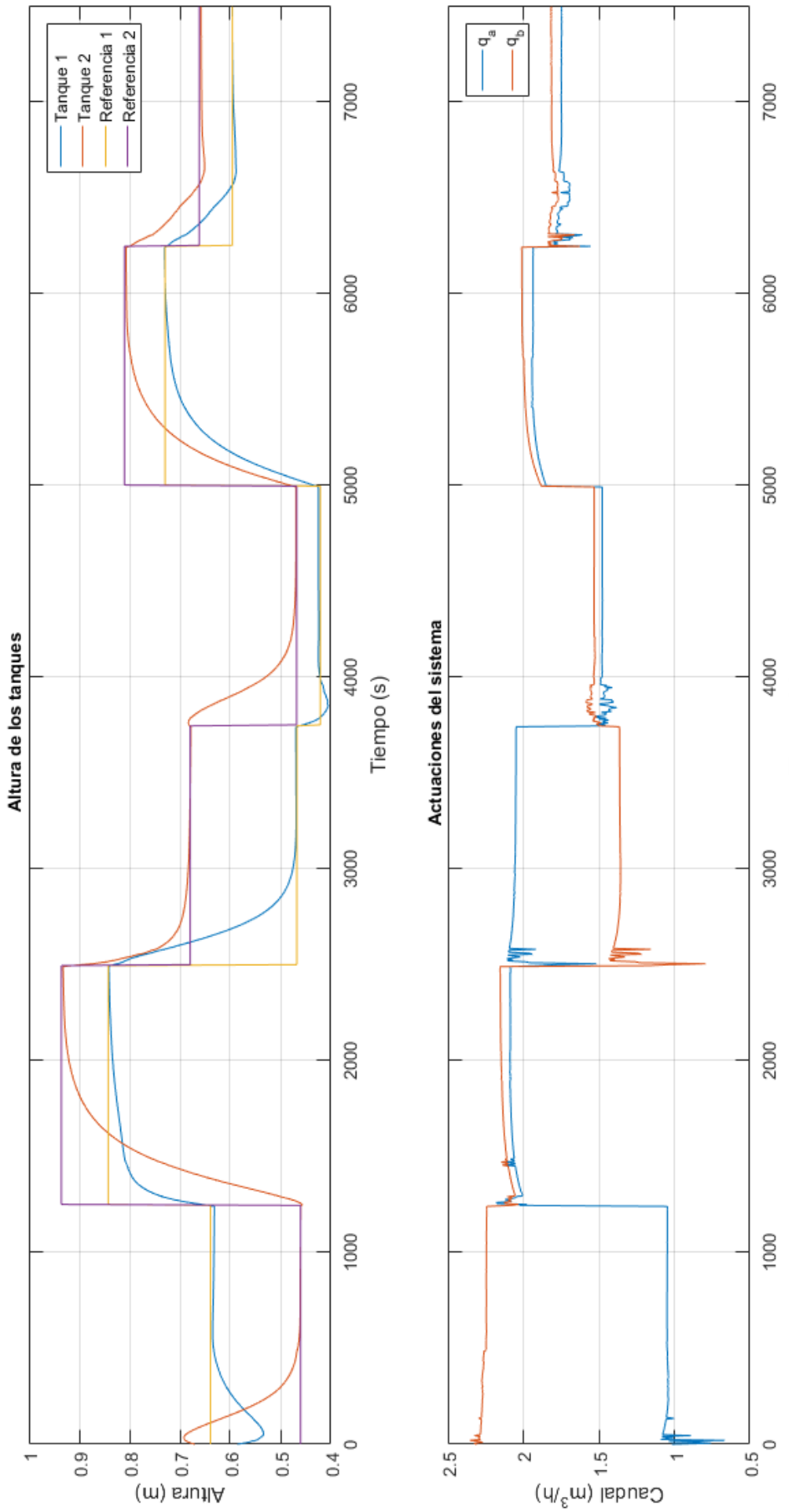


Figura 6.8 Resultados de la simulación con un modelo sin ruido.

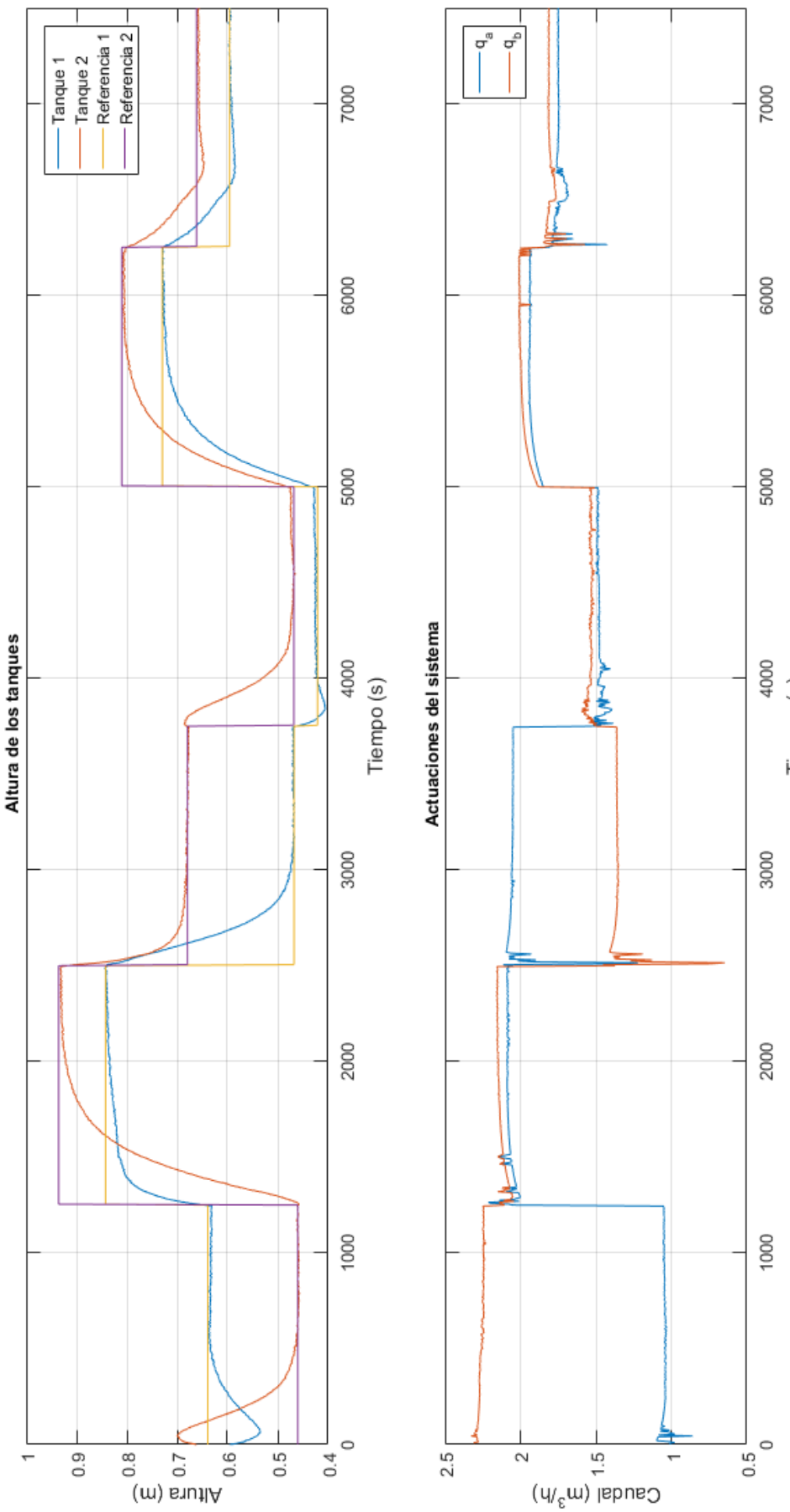


Figura 6.9 Resultados de la simulación con un modelo ruidoso.

Índices de desempeño

La evolución de los índices de desempeño puede verse en el gráfico de barras de la figura 6.10.

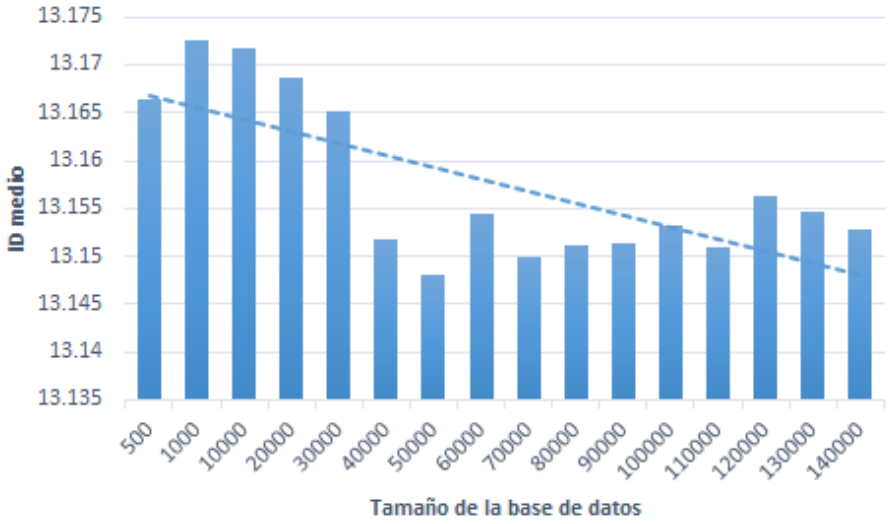


Figura 6.10 Evolución del valor medio de los índices de desempeño.

Debido a la sencillez del problema, aumentar el tamaño de la base de datos no mejora en gran medida el rendimiento del controlador, sino que permanece oscilando en torno a un punto. Sin embargo, la recta de regresión de los datos proporciona una pendiente decreciente, lo cual nos podría estar indicando una mejoría.

Factibilidad

La aparición de puntos no factibles tiene lugar cuando la base de datos no está bien muestreada o bien no tiene información suficiente. Como vemos en la figura 6.11, existe siempre una solución factible al problema de optimización llegados a un determinado tamaño.

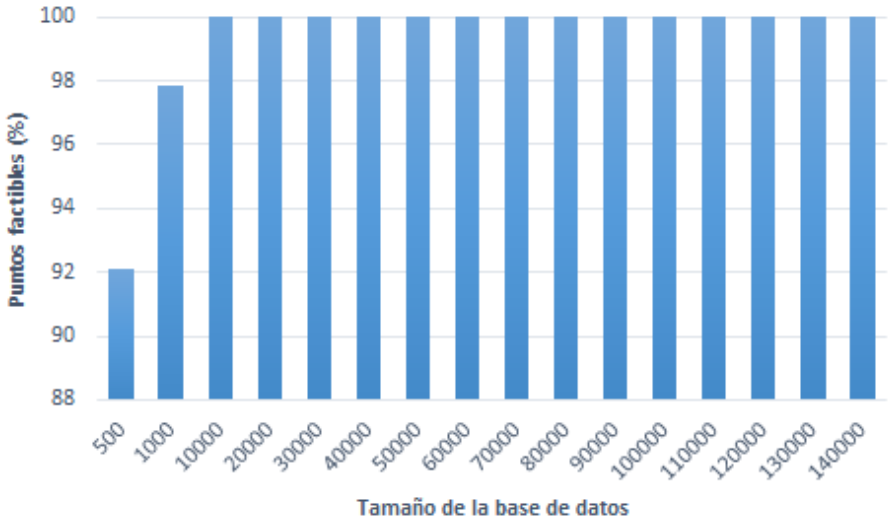
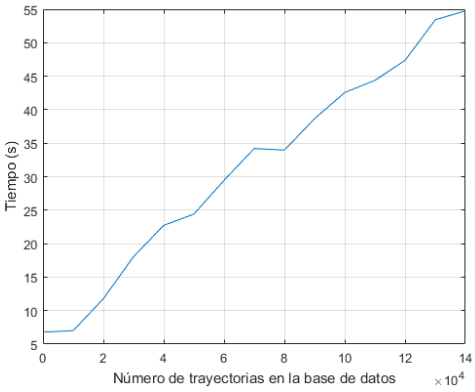


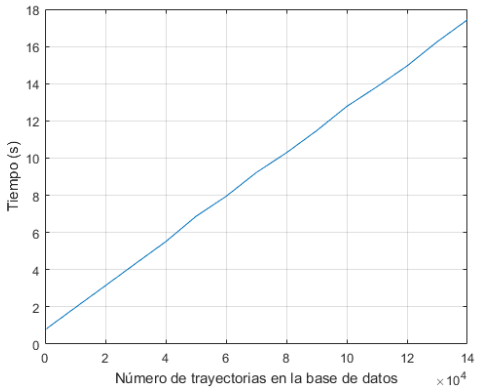
Figura 6.11 Porcentaje de puntos factibles.

Coste computacional y mejora

Vamos a centrarnos en el estudio del coste de la búsqueda de estados candidatos. Esto incluye tanto el problema del cálculo de la distancia como el de la búsqueda de los mínimos. Además, veremos la mejora respecto a su versión no paralelizada.



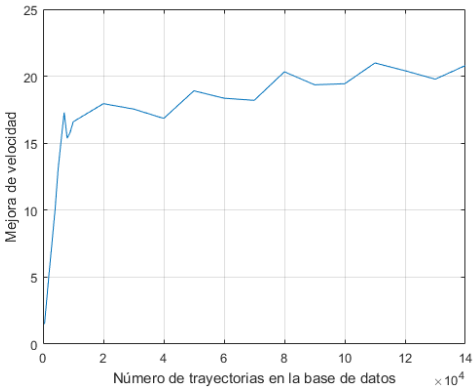
(a) Mínimo.



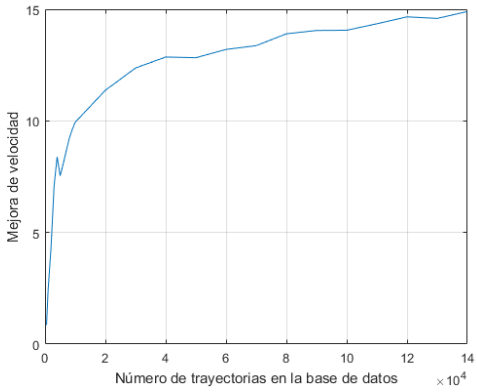
(b) Distancia.

Figura 6.12 Coste computacional (GPU).

Como era de esperar, el tiempo necesario para realizar los cálculos aumenta conforme se añaden trayectorias a la base de datos. En realidad, la tendencia debería ser en ambos problemas lineal pero, debido al método de la bisectriz (cuyas iteraciones dependen del número de trayectorias), la cantidad de veces que se ejecuta la búsqueda de los mínimos no es constante, provocando por tanto que el cálculo del mínimo no presente una tendencia lineal.



(a) Mínimo.



(b) Distancia.

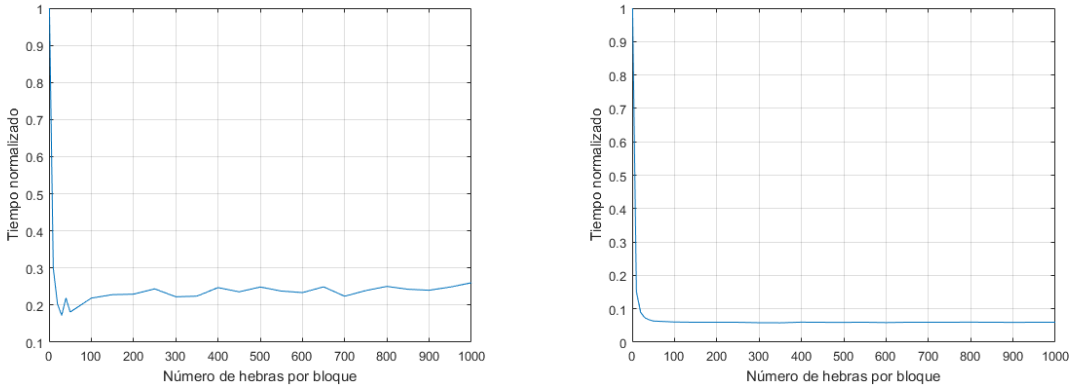
Figura 6.13 Mejora de velocidad.

Puede verse que la tendencia de estas curvas no es lineal. Esto es debido a que al principio se experimenta una mejora muy grande pero, conforme vamos avanzando esta comienza a mantenerse. Debería llegarse a un punto en el que la GPU sature y que, por tanto, deje de incrementar la mejoría.

Aunque no se hayan representado los tiempos de todas las funciones que actúan en el programa, cabe aclarar que la ejecución del programa es lo suficientemente rápida como para cumplir con el tiempo de muestreo, tanto en la versión de la GPU como de la CPU. Es más, las gráficas anteriores se refieren a 1500 instantes de muestreo. Es fácil ver que se supera con creces la imposición temporal de 5 segundos.

Hebras y bloques

Por otro lado, presentamos cómo afecta la cantidad de hebras por bloque a la hora de la ejecución de los kernels. En un principio, con 1 hilo por bloque, la ejecución se hace realmente lenta y posteriormente cae en picado hasta permanecer relativamente constante, lo cual es completamente congruente con la arquitectura de las GPUs y su funcionamiento.



(a) Mínimo. **(b)** Distancia.

Figura 6.14 Tiempo normalizado de computación respecto a la cantidad de hebras por bloque.

Cuando nos referimos a tiempo normalizado queremos decir que hemos dividido todos los valores por el máximo de todos ellos, con lo que obtenemos un resultado adimensional. En nuestro caso, hemos elegido una cantidad de 500 hebras por bloque para las pruebas anteriores.

6.2.4 Pruebas en la planta real

Una vez visto el funcionamiento en simulación, nos planteamos la posibilidad de implementar el controlador anterior en la planta de los 4 tanques presente en los laboratorios de la escuela.

El envío de las señales de control, medidas de los sensores, etc, se hace a partir de la conexión de un ordenador y un autómata programable. Esta comunicación se lleva a cabo a través del protocolo OPC, aunque resulta transparente al usuario. Una vez dicho esto, cabe aclarar que para poder controlar la planta existen 2 posibilidades:

- Uso del entorno labview.
- Uso de Matlab y Simulink.

En nuestro caso, al tener ya definido gran parte del código en Matlab, nos decantamos por este último. Simplemente sería necesario adaptar nuestro fichero ".m" a un bloque de Simulink. Esto es así porque únicamente se necesita conectar las salidas de lo que sería nuestro controlador (q_a y q_b) a la entrada del bloque OPC encargado de la comunicación, y lo contrario para el caso de las entradas (h).

Sin embargo, existe un problema, este no es más que el hecho de que el ordenador que se encarga de realizar estas tareas no disponía de tarjeta gráfica dedicada con arquitectura CUDA. Por ello, usamos un ordenador portátil (que sí dispone de CUDA) el cual conectamos usando un cable ethernet cruzado al ordenador encargado de gestionar la planta, creando una pequeña red de área local (LAN).

Pero claro, es necesario transmitir la información de un Matlab a otro, puesto que nuestro objetivo es que el ordenador "remoto" sea el encargado de calcular las acciones de control usando el algoritmo anterior. Esto quiere decir que en cada instante de muestreo debe enviarse el estado del sistema y sus referencias y, una vez realizados los cálculos, deben devolverse las acciones de control que deben aplicarse al sistema.

Un esquema de funcionamiento puede verse en la figura 6.15.

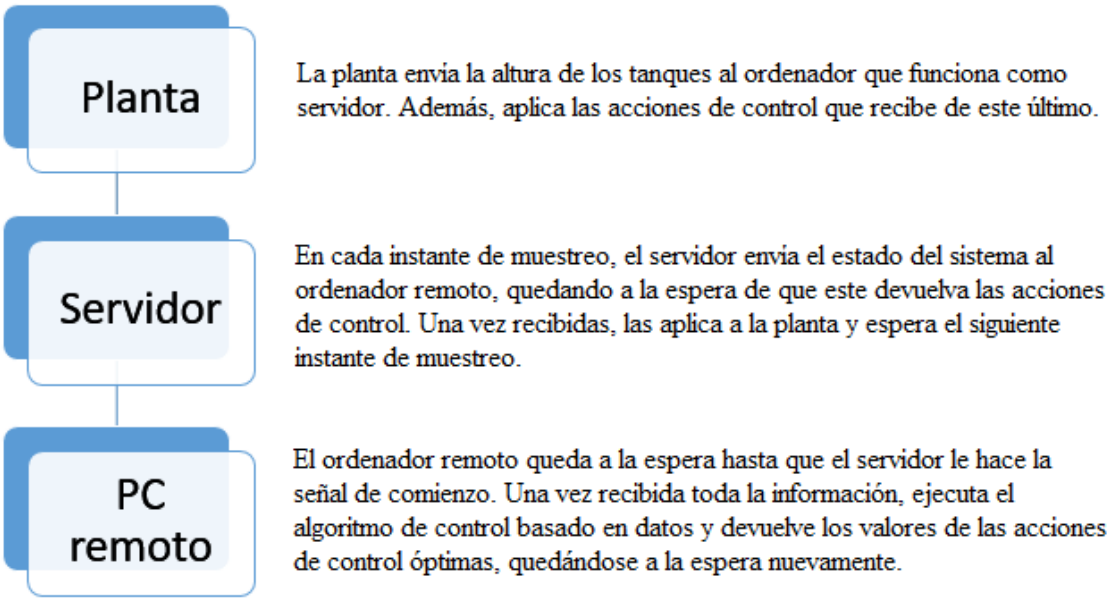


Figura 6.15 Esquema de conexiones de la LAN.

Para ello, es necesario crear un objeto UDP en el workspace de cada uno de los Matlab en ejecución, el cual nos permitirá realizar este intercambio de información. Esto implica que anteriormente hemos asignado direcciones IP de manera manual para el puerto ethernet en uso y que, además, tenemos algún puerto abierto que podamos usar. Una vez creado estos objetos, es posible la transmisión de cadenas de caracteres de manera bidireccional entre ellos. Puesto que nosotros enviamos valores numéricos, no hay más que convertir los mensajes recibidos al asignarlos a cada variable. Un ejemplo de un programa sencillo para servidor y cliente puede verse en la figura 6.16.

```
%Ordenador B
ipA='192.168.10.10'; portA=9090;
ipB='192.168.10.12'; portB=9091;
udpB=udp(ipA,portA,'LocalPort',portB);
fopen(udpB);

disp('Indicamos que vamos a transmitir');
fprintf(udpB,'Preparado');
pause(0.1);

%La sentencia num2str escribe el número que recibe de argumento como
%una cadena de caracteres, que es lo que escribimos con fprintf.
disp('Enviamos el número 4.5 como cadena de caracteres');
fprintf(udpB,num2str(4.5));
pause(0.1);
```

```
%Ordenador A
ipA = '192.168.10.10';   portA = 9090;
ipB = '192.168.10.12';   portB = 9091;
udpA = udp(ipB,portB,'LocalPort',portA);
fopen(udpA);

disp('Esperando confirmación del servidor');
%Quitamos el espacio en blanco que se genera al transmitir.
auxiliar=fscanf(udpA);
auxiliar(end)=[];

%Con la llamada strcmp nos aseguramos de que el mensaje que ha
%llegado ha sido exactamente "Preparado". Una vez hecho esto esperamos
%el número, que es lo que realmente nos interesaría.
    if strcmp(auxiliar,'Preparado')==0
        disp('Fallo en la recepción. Fin del programa')
    else
        %La sentencia str2double convierte un número escrito como cadena
        %de caracteres a un tipo double.
        disp('Convertimos la cadena de caracteres a un número')
                a=str2double(auxiliar);
    end
```

Figura 6.16 Ejemplo de comunicación.

El resultado de un ensayo con seguimiento en escalón puede verse en la figura 6.17.

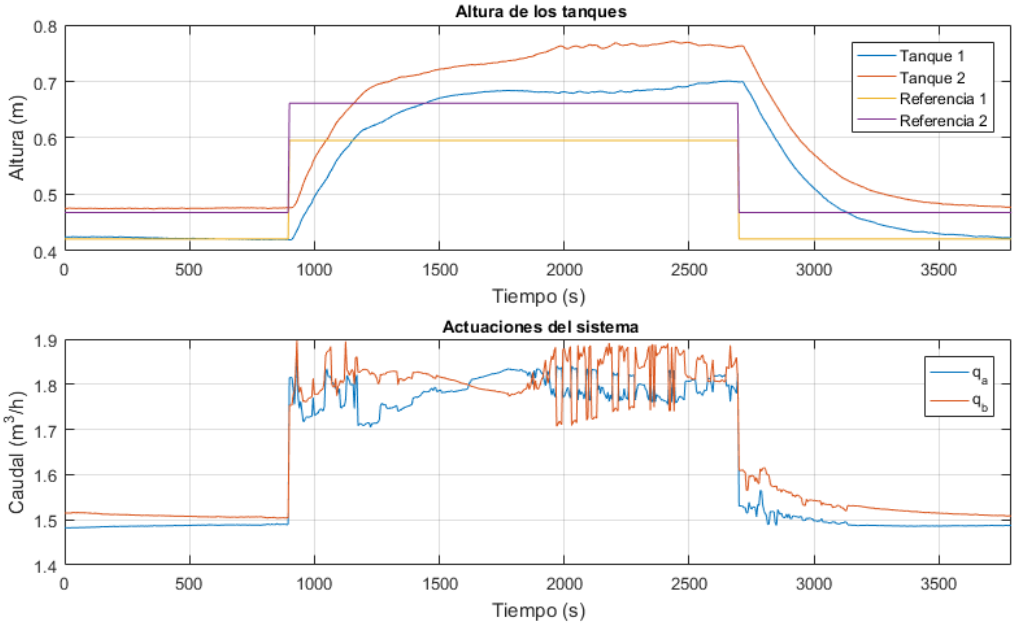


Figura 6.17 Ensayo en la planta real.

La apertura de las válvulas fue ajustada para que en uno de los puntos de operación las alturas de los tanques correspondieran a las del modelo usado en las simulaciones. Sin embargo, vemos como el algoritmo es realmente muy sensible ante las perturbaciones (en este caso, a la apertura de las válvulas), llevando a un error en régimen permanente realmente grande debido a las discrepancias con el modelo y que, también lleva como consecuencia, un funcionamiento anómalo en el cálculo de las acciones de control, como puede verse en las fuertes oscilaciones de los caudales.

Después de haber obtenido este resultado y para tratar de comprobar que se trata realmente de un problema del algoritmo y de ninguna manera de algo relacionado con la planta real, realizamos una simulación en la que perturbamos en el modelo de Simulink de manera voluntaria la apertura de las válvulas aproximadamente en un 10 %. El resultado puede verse en la figura 6.18.

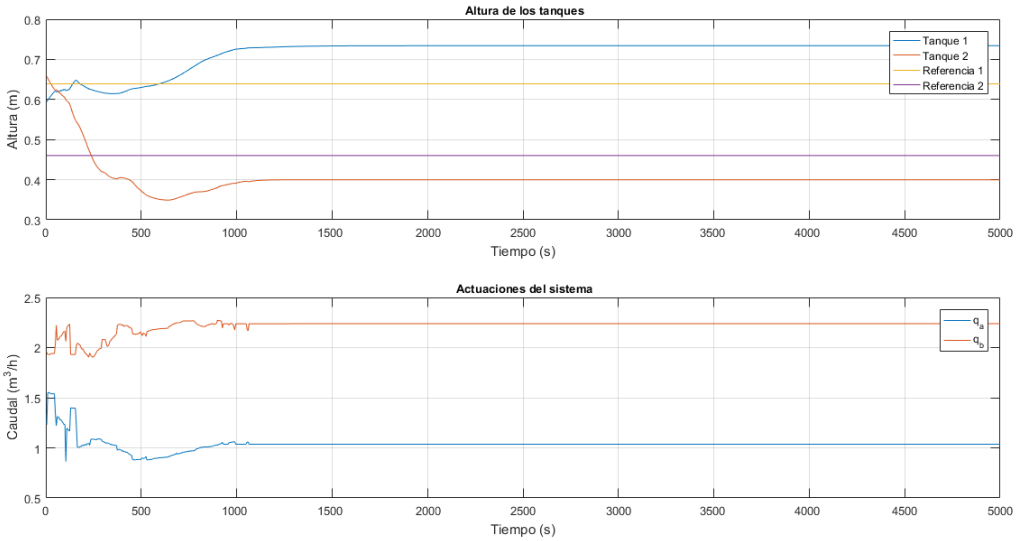


Figura 6.18 Simulación con modelo discrepante.

La simulación nos lleva a errores en régimen permanente muy parecidos a los obtenidos en el ensayo real, por lo que concluimos que el algoritmo no tiene capacidad para hacer frente a las discrepancias del modelo. Es decir, el algoritmo aplica las acciones de control óptimas para alcanzar la referencia especificada, sin embargo, no tiene ninguna manera de corregir esta diferencia de alturas de ninguna manera, puesto que al consultar en la base de datos las trayectorias cercanas para las referencias especificadas, encuentra acciones de control muy similares a las que está aplicando.

7 Conclusiones y posibles avances

Hemos visto que con el uso de las herramientas actuales es relativamente simple la paralelización de un problema. Además, las mejoras de rendimiento que se pueden obtener respecto a las versiones en la CPU son muy grandes en algunos casos. Por ejemplo, en algunas situaciones, estas mejoras hacen que sea posible resolver en tiempo real problemas que antes no eran factibles debido a su tiempo de computación.

Respecto al algoritmo PQP que tratamos en el capítulo de modelado, podría resultar interesante una implementación que realice en un fichero MEX de Matlab todas las operaciones matriciales necesarias (inversiones, multiplicaciones, etc) y no sólo unas cuantas. Además, usando librerías como CUBLAS, quizá sea posible vencer la barrera por la que se tuvo que abandonar la idea.

En cuanto al control, hemos visto cómo las discrepancias entre el modelo utilizado y la planta real afectan fuertemente al algoritmo, provocando la existencia de un error en régimen permanente importante. Sería interesante plantear la posibilidad de estimar esta discrepancia a partir del vector de estados, con el objetivo de paliar de alguna manera este comportamiento, por ejemplo, estableciendo un set point modificado para el que el sistema realmente alcance la referencia que queremos.

Por otro lado, el método de la bisectriz para la búsqueda de estados candidatos funciona, pero es probablemente mejorable. Por otra parte, el algoritmo aquí planteado tiene una formulación genérica. Esto implica que existe la posibilidad de estudiar su efecto en problemas muy diversos dentro del ámbito del control.

Índice de Figuras

Índice de Tablas

Bibliografía

[1] *NVIDIA Corporation*. http://www.nvidia.es/object/cuda-parallel-computing-es.html , 14 de Junio de 2017.

[2] J. Sanders y E. Kandrot, *CUDA by example*, 2007.

[3] *NVIDIA Corporation*. http://www.nvidia.com/object/uavs-drones-technology.html , 14 de Junio de 2017.

[4] *MathWorks*. https://es.mathworks.com/products/parallel-computing.html , 14 de Junio de 2017.

[5] Grupo de estimación, predicción, optimización y control, *Transparencias de programación en CUDA con Matlab.*

[6] *MathWorks*. https://es.mathworks.com/help/distcomp/gpuarray.html , 14 de Junio de 2017.

[7] *MathWorks*. https://es.mathworks.com/help/distcomp/gather.html , 14 de Junio de 2017.

[8] *MathWorks*. https://es.mathworks.com/help/matlab/ref/bsxfun.html , 14 de Junio de 2017.

[9] *MathWorks*. https://es.mathworks.com/help/distcomp/pagefun.html , 14 de Junio de 2017.

[10] *MathWorks*. https://es.mathworks.com/help/matlab/ref/arrayfun.html , 14 de Junio de 2017.

[11] K. H. Johansson, *"The quadruple-tank process"*, IEEE Transactions on Control Systems Technology, vol 8 (2000).

[12] V. S. Gómez, D. Muñoz de la Peña y T. Álamo, *"Modelo basado en optimización y bases de datos"*, 2015.

[13] M. Brand, V. Shilpiekandula, C. Yao y S.A. Bortoff, *"A Parallel Quadratic Programming Algorithm for Model Predictive Control"*, Mitsubishi Electric Research Laboratories, Agosto de 2011.

[14] M. Brand y D. Chen, *"Parallel Quadratic Programming for Image Processing"*, Mitsubishi Electric Research Laboratories, Septiembre de 2011.

[15] S. Di Cairano, M. Brand y S.A. Bortoff, *"Projection-free Parallel Quadratic Programming for Linear Model Predictive Control"*, International Journal of Control (2013), 86:8, 1367-1385.

[16] S. Boyd y L. Vandenberghe, *Convex Optimization*, Séptima edición, 2009.

[17] D. Rodríguez y C. Bordóns, *Apuntes de Ingeniería de Control*, Revisión del año 2007.

[18] V. S. Gómez, *"Control predictivo basado en datos. Aplicación a un sistema de cuatro tanques."* TFG , 2015.

[19] V. S. Gómez, D. Muñoz de la Peña y T. Álamo, *"Control predictivo basado en datos"*, 2015.