UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Instituto Universitario de Microelectrónica Aplicada
Sistemas de información y Comunicaciones

Máster en
Tecnologías de Telecomunicación

Trabajo Fin de Máster

Hardware synthesis Methodology of N-FINDR algorithm

Autor: **Anabella Medina Machín**
Tutor(es): **Gustavo Marrero Callicó**
Sebastián López Suárez
Fecha: **27 -julio - 2011**

IUMA
t +34 928 451 086 | iuma@iuma.ulpgc.es | Campus Universitario de Tafira
f +34 928 451 083 | www.iuma.ulpgc.es | 35017 Las Palmas de Gran Canaria

Versión: 2.0 – 14 jun 2011

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Instituto Universitario de Microelectrónica Aplicada
Sistemas de información y Comunicaciones

Máster en
Tecnologías de Telecomunicación

Trabajo Fin de Máster

Hardware synthesis Methodology of N-FINDR algorithm

HOJA DE FIRMAS

Alumno/a: Anabella Medina Machín Fdo.:

Tutor/a: Gustavo Marrero Callicó Fdo.:

Tutor/a: Sebastián López Suárez Fdo.:

Fecha: 27 -julio - 2011

IUMA
t +34 928 451 086 | iuma@iuma.ulpgc.es | Campus Universitario de Tafira
f +34 928 451 083 | www.iuma.ulpgc.es | 35017 Las Palmas de Gran Canaria

Versión: 2.0 – 14 jun 2011

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Instituto Universitario de Microelectrónica Aplicada
Sistemas de información y Comunicaciones

Máster en
Tecnologías de Telecomunicación

Trabajo Fin de Máster

Hardware synthesis Methodology of N-FINDR algorithm

HOJA DE EVALUACIÓN

Calificación: .....

Presidente: Tomás Bautista Delgado Fdo.:

Secretario: Gustavo Marrero Callicó Fdo.:

Vocal: Roberto Esper-Chaín Falcón Fdo.:

Fecha: 27 -julio - 2011

IUMA
t +34 928 451 086 | iuma@iuma.ulpgc.es | Campus Universitario de Tafira
f +34 928 451 083 | www.iuma.ulpgc.es | 35017 Las Palmas de Gran Canaria

Versión: 2.0 – 14 jun 2011

Índice

1. Objetivos

En los últimos años, la evolución en los sensores hiperespectrales ha supuesto un salto cualitativo en las aplicaciones orientadas a la observación remota de la tierra. Estos instrumentos se caracterizan por su capacidad para medir la radiación reflejada en una amplia gama de longitudes de onda, pudiendo registrar información en cientos de canales espectrales. Así, cada material se caracteriza por tener una firma espectral propia y diferente del resto de materiales [1].

El resultado de la toma de datos por parte de un sensor hiperespectral sobre una determinada escena puede ser representado en forma de un cubo de datos, con dos dimensiones para representar la ubicación espacial de un píxel, y una tercera dimensión que representa la singularidad espectral de cada píxel en diferentes longitudes de onda.

Debido a la resolución espacial disponible en los sensores de observación remota de la tierra, la mayor parte de los píxeles registrados por el sensor constituyen una mezcla de diferentes substancias a nivel sub-píxel. La Figura 1.1. se representa una imagen

hiperespectral, dónde se pueden ver las diferentes bandas y estos píxeles mezcla. Para solucionar este problema de mezclado, una de las técnicas más ampliamente utilizadas a la hora de llevar a cabo el análisis y clasificación de imágenes hiperespectrales es la separación espectral o *spectral unmixing.*

Estas técnicas suelen consistir en dos pasos claramente diferenciados: en primer lugar, se identifican firmas espectrales asociadas a componentes espectralmente puros en la imagen, denominados *Endmembers*. A continuación, el resto de componentes de la imagen se expresan mediante combinaciones de *Endmembers*, evitando así el habitual problema de la mezcla espectral y permitiendo realizar una cuantificación a nivel sub-píxel [2].

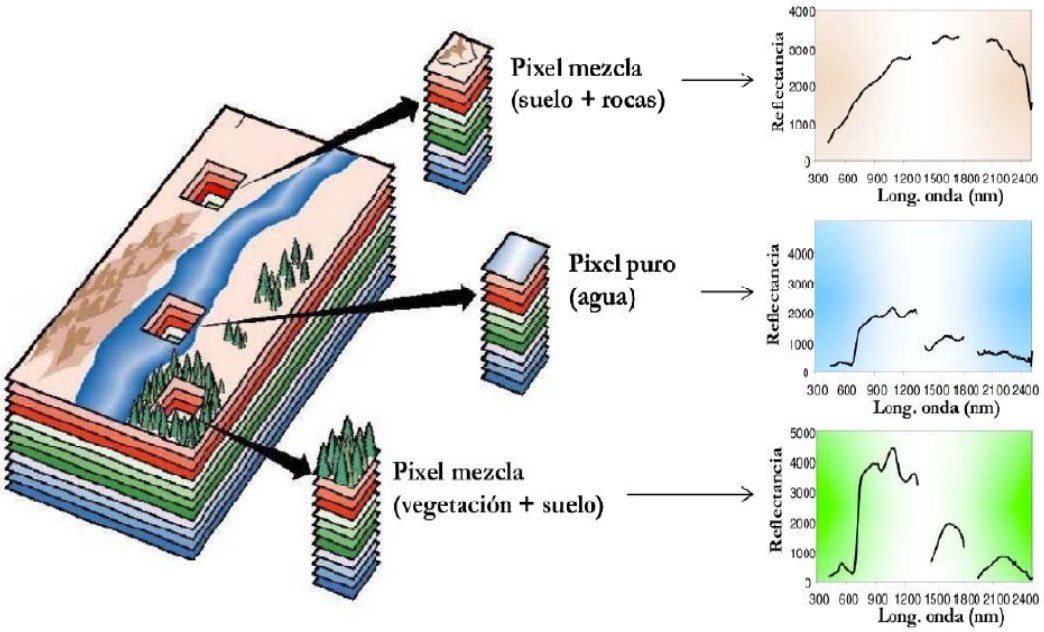


Figura 1.1. Tipos de píxeles y su firma espectral de una imagen hiperespectral.

A lo largo de los últimos años se han desarrollado diversos métodos para este propósito. Uno de los métodos más utilizados es el N-FINDR [3], que utiliza una técnica basada en identificar los *Endmembers* como los vértices del simplex de mayor volumen que puede formarse en la nube de puntos dada por todos los píxeles de la imagen hiperespectral. Se define un símplex o n-símplex como el análogo en n dimensiones de un triángulo. Más

exactamente, un símplex es la envoltura convexa de un conjunto de (n + 1) puntos independientes afines en un espacio euclídeo de dimensión n o mayor [4].

Los algoritmos basados en estas técnicas son altamente costosos desde el punto de vista computacional, lo que resulta de gran interés la síntesis hardware de estos algoritmos implementados de forma más generalizada en Matlab [5].

El objetivo de este Trabajo Fin de Máster (TFM) es desarrollar una metodología para la conversión de un código Matlab a un lenguaje de descripción hardware como Verilog o VHDL [6]. Para llevar a cabo este proceso se hará uso de diferentes herramientas software.

En la fase inicial de esta conversión se utilizará la *Toolbox* proporcionada por Matlab para la obtención de código C/C++ embebido (*Embedded C Toolbox*) [7], así como también la *Toolbox Fixed-Point* [8] que permite la utilización de aritmética de punto fijo. También se ha recurrido a otro tipo de librerías de punto fijo desarrolladas en lenguaje C/C++ [9].

Mediante estas herramientas se pretende lograr un código C/C++ independiente descrito mediante aritmética de punto fijo que resulte adecuado para su implementación en dispositivos electrónicos tales como los DSPs y las FPGAs.

Para la implementación sobre FPGAs, se utilizará la herramienta Catapult C para obtener el código de descripción hardware ejecutable en sistemas embebidos [10], que permitirá convertir el código C/C++ en un lenguaje de descripción hardware como Verilog o VHDL.

A modo de conclusión, el objetivo principal de este trabajo fin de máster es desarrollar la metodología para la generación del código de descripción hardware a partir de la función N-FINDER desarrollada en código Matlab.

Además, se llevará a cabo un análisis comparativo de diferentes parámetros, como eficiencia, precisión, recursos, ect., generando el código en punto fijo, a partir de las librería de Matlab de punto fijo. Comparando los resultados si se genera el código en punto fijo a partir de librerías en C/C++. Por otro lado, también se contrastarán los resultados obtenidos para aritmética de parte entera y para aritmética de coma flotante.

Las medidas se tomarán, tanto a nivel software como a nivel de sintetizado hardware.

2. Resumen del estado del arte

La práctica totalidad de las técnicas de análisis hiperespectral desarrolladas hasta la fecha presuponen que la radiación obtenida por un sensor en un determinado píxel viene dado por la contribución de diferentes materiales que residen a nivel sub-píxel. Las técnicas basadas en este modelo son altamente costosas desde el punto de vista computacional. A continuación se detallan las características genéricas de estas técnicas basadas en el modelo lineal de mezcla.

2.1. Modelo lineal de la mezcla

El modelo lineal de mezcla es el más utilizado en el análisis de imágenes hiperespectrales, debido a su mayor sencillez. Los píxeles mezcla se representan como una combinación lineal de firmas asociadas a componentes espectralmente puros. Este modelo ofrece resultados satisfactorios cuando los componentes que residen a nivel subpíxel aparecen

espacialmente separados, situación en donde los fenómenos de absorción y reflexión de la radicación incidente pueden ser caracterizados siguiendo un modelo lineal.

El modelo lineal de mezcla se puede interpretar como un espacio bidimensional tal como se presenta en la Figura 2.1. En este plano se puede apreciar como todos los puntos de la imagen quedan englobados dentro del triángulo formado por los tres puntos más externos (correspondientes con los elementos puros). Los vectores asociados a dichos puntos constituyen un nuevo sistema de coordenadas con origen en el centroide de la nube de puntos, de forma que cualquier punto de la imagen puede expresarse como combinación lineal de los puntos más externos, siendo estos puntos los candidatos a ser *Endmembers*. Para ello se identifican los elementos extremos de la nube de puntos N-dimensional.

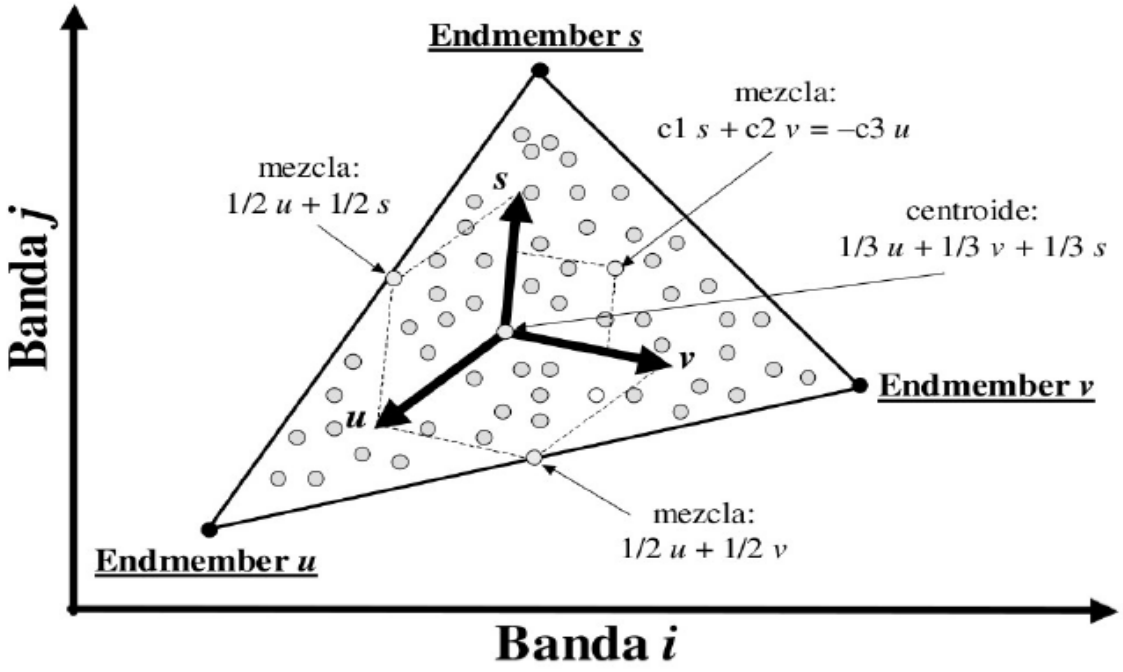


Figura 2.1. Interpretación del modelo lineal de mezcla

2.2. Procesado de imágenes hiperespectrales

El procesado de imágenes hiperespectrales tal y como se ve en la Figura 2.2. se realiza en varias etapas, primero se realiza una adaptación de los datos obtenidos por el sensor para obtener un formato de datos normalizado como la reflectancia, posteriormente se realiza una etapa de procesado en donde se realiza una reducción de bandas mediante algoritmos como el *Principal Components Analysis* (PCA) que generan combinaciones lineales de intensidades de los píxeles mutuamente incorreladas y de máxima variancia. La salida de esta etapa es utilizada en la extracción de *Endmembers* que obtiene las firmas espectrales de los posibles píxeles puros. Finalmente mediante la segmentación y clasificación de los *Endmembers* se obtienen los píxeles puros de la imagen.

La extracción de *Endmembers* es la parte más importante del proceso ya que su utilización proporciona como resultados los posibles píxeles puros, en este TFM nos vamos a centrar en la implementación de un algoritmo de extracción de *Endmembers* sin tener en cuenta las demás partes del proceso.

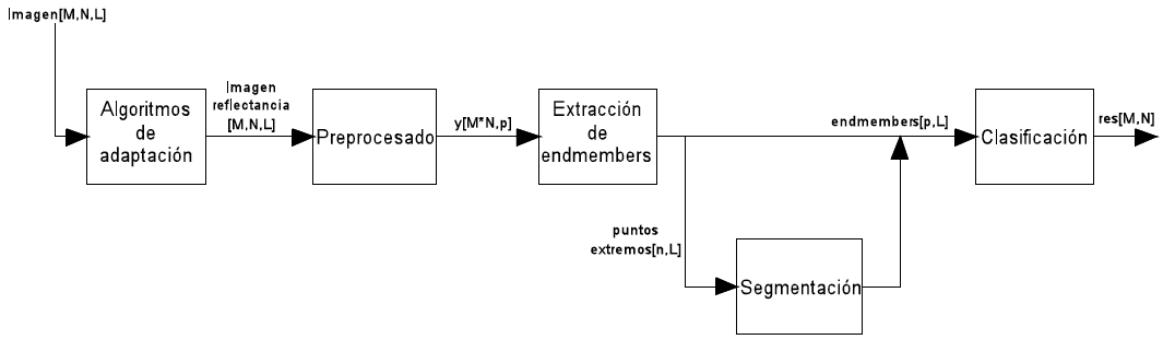


Figura 2.2. Diagrama de bloques del procesamiento de imágenes hiperespectrales

La extracción de *Endmembers* se basa en el modelo de composición lineal que se puede representar de la siguiente manera:

$$r = Ms + n$$

- Donde r es una matriz con cada uno de los píxeles de la imagen

- n es la componente de ruido aleatorio que existe en cada píxel
- M es la matriz de *Endmembers* M = [m1, m2, …, m_p] (siendo p el número de *Endmembers*

Por lo tanto es necesario obtener los valores de los *Endmembers* en el cálculo de este modelo. De esta manera han aparecido numerosos algoritmos matemáticos de los cuales destacan los tres siguientes:

- *Pixel Purity Index* (PPI) [11]. En este algoritmo se proyecta la imagen un número de veces muy elevado (entre 1000 y 10000), sobre vectores aleatorios que se construyen para cada proyección almacena, al valor mayor de la proyección en valor absoluto. Mediante el cálculo del valor más votado se designan los puntos extremos. Una vez que obtenemos estos *Endmembers* se realiza una clasificación para reducir el número de *Endmembers*.
- *N-FINDR* [12]. Partiendo de la base de que un símplex es equivalente en n dimensiones a un triángulo. Este algoritmo se basa en buscar un simplex de volumen máximo y un número de vértices p, basándose en la suposición de que en aquel simplex de mayor volumen que encierre a todos los puntos, sus p vértices serán los p *Endmembers* buscados.
- *Vertex Component Analysis* (VCA) [13]. Se obtiene cada uno de los *Endmembers* al realizar una proyección completa de la imagen sobre un vector perpendicular al anterior *Endmember* obtenido, realizándose por tanto p proyecciones hasta encontrar p *Endmembers*.

A continuación, se procede a explicar con más detalle el algoritmo N-FINDR, puesto que es del que se va a partir para el desarrollo de este trabajo fin de máster.

2.2.1. Algoritmo N-FINDR

El algoritmo N-FINDR, desarrollado por M. E. Winter [14], es uno de los algoritmos más ampliamente utilizado para la extracción de *Endmembers* en imágenes hiperespectrales, debido principalmente, a una racional facilidad de diseño, contrapuesta a otras desventajas, como son el coste computacional y la inicialización aleatoria de unos

Endmembers que toma como datos de partida. Lo cual implica una ejecución no eficiente, así como también, la generación de distintos resultados en cada ejecución.

El fundamento principal del algoritmo es calcular el máximo volumen formado por un símplex, con un número de vértices p. Se parte de la asunción, de que el símplex de mayor volumen, será aquel que esté formado por los vértices que representan a los píxeles más puros. A su vez, estos vértices, representarán por tanto a los *Endmembers*.

Son múltiples las modificaciones que se han presentado sobre el algoritmo, y que mejoran considerablemente su ejecución, sin embargo en lo que sigue se presentará la versión fundamental de cuya implementación se hará uso a lo largo de este trabajo fin de máster.

Antes de realizar la ejecución del algoritmo, es fundamental llevar a cabo la fase de preprocesado, que incluye la determinación del número de *Endmembers* a calcular, p, y la transformación espacial de la imagen, para realizar una reducción dimensional espectral de la misma de L a p bandas.

Una vez se ha ejecutado este preprocesado, se presenta el paso principal del algoritmo, que consiste en realizar una búsqueda exhaustiva para un conjunto arbitrario p de vectores e1, e2, ...,ep y calcular el volumen generado por el simplex, S (e1, e2, ..., ep), a partir de la siguiente expresión:

$$V(e_1, \dots, e_p) = \frac{\det\begin{bmatrix} 1 & 1 & \dots 1 \\ e_1 & e_2 & \dots e_p \end{bmatrix}}{(p-1)!}$$

Se encuentra un sistema de vectores de muestras, designado por $\{e^*_1, e^*_2, \dots, e^*_p\}$, que forman el simplex de p-vértices para alcanzar el máximo valor:

$\{e^*_1, e^*_2, \dots, e^*_p\} = \arg\{\max_{\{e\}} V(e_1, e_2, \dots, e_p)\}$

Se destaca que la búsqueda necesita de $\binom{N}{p} = (N!)/(p!\,(N-p)!)$ iteraciones, por lo que el coste del algoritmo, como ya se adelantó, es muy elevado. Una vez encontrado el conjunto de vectores que maximicen el volumen, se concluye el proceso, y ese conjunto de vectores es el conjunto de p *Endmembers* buscados.

Una de las principales líneas es que se investiga, para mejorar este algoritmo, se basa en buscar una parada mucho más temprana del algoritmo, una vez que una iteración y la anterior produzcan resultados muy parecidos. Para ello, se inicializa el N-FINDR con vectores aleatorios en numerosas ocasiones, de ahí que reciba el nombre de Random N-FINDR (RN-FINDR) [15].

2.3. Implementación hardware

Las técnicas de análisis hiperespectral, como ya se ha adelantado, se basan en la realización de operaciones matriciales que resultan muy costosas desde el punto de vista computacional. En cambio, el carácter repetitivo de estas operaciones las hace altamente susceptibles de ser implementadas en arquitecturas paralelas, mejorando de una forma importante su rendimiento computacional. Las técnicas de computación paralela se han utilizado ampliamente en el tratamiento de grandes imágenes, obteniendo tiempos de respuesta muy reducidos. En la actualidad existen diferentes plataformas que permiten realizar el procesado paralelo necesario de estos algoritmos, como las GPUs, o hardware dedicado como las FPGAs y los circuitos integrados de aplicación específica (ASIC). La implementación hardware es la que mejores resultados proporciona, pero es importante destacar que la complejidad de la tarea de implementación aumenta considerablemente, por lo que se hace necesario el estudio de alternativas software que permitan una implementación hardware a partir de algoritmos escritos en lenguajes de alto nivel como Matlab.

2.3.1. Uso de FPGAs en la implementación de algoritmos hiperespectral

Las FPGAs (*Field Programmable Gate Array*) son dispositivos programables ideales para la realización de prototipos, en el que los cambios en la implementación son elevados permitiendo realizar múltiples pruebas una vez que se ha programado la placa. Las FPGAs son unos dispositivos que proporcionan un gran equilibrio entre flexibilidad y eficiencia, por lo que se ha elegido este tipo de dispositivo para la implementación del algoritmo N-FINDR.

Las FPGAs están compuestas por una matriz bidimensional de bloques configurables que se pueden conectar mediante recursos generales de interconexión. Estos recursos incluyen segmentos de pista de diferentes longitudes, más unos conmutadores programables para enlazar bloques a pistas o pistas entre sí. Por lo tanto, lo que se programa en una FPGA son los conmutadores que sirven para realizar las conexiones entre los diferentes bloques, más la configuración de los propios bloques. En la Figura 2.3 se muestra una imagen con la estructura básica de una FPGA.

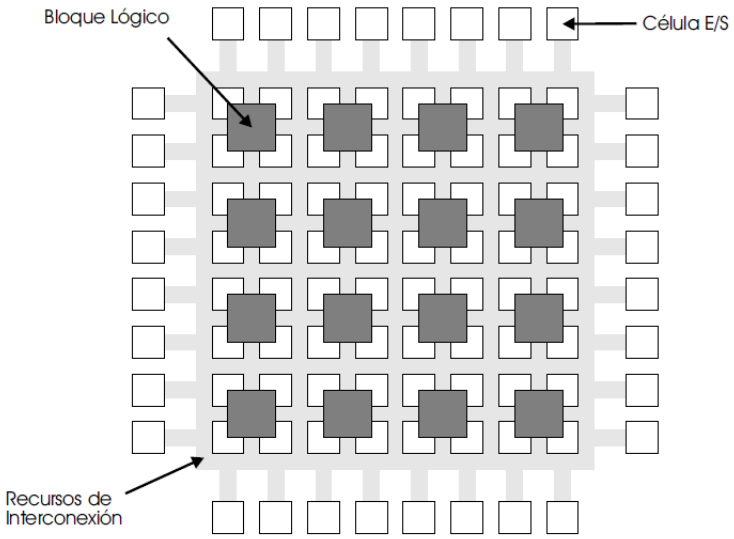


Figura 2.3. Estructura básica de una FPGA

Los elementos básicos constituyentes de una FPGA, son los siguientes

- Bloques lógicos, cuya estructura y contenido se denomina arquitectura
- Recursos de interconexión, cuya estructura se denomina arquitectura de enrutado
- Memoria RAM, que es cargado en el RESET para configurar bloques y contenidos

Por lo tanto, las ventajas más importantes que proporciona el uso de FPGAs son el bajo coste de prototipado y el corto tiempo de producción. Los inconvenientes residen en la baja velocidad de operación y la baja densidad lógica frente a los dispositivos ASIC.

Las FPGAs que existen en el mercado, se pueden clasificar en cuatro grandes familias dependiendo de la estructura que adoptan los bloques lógicos que tienen definidas.

1. Matriz simétrica, como son las de XILINX
2. Basada en canales, como son las de ACMEL.
3. PLD jerárquica, como son las de ALTERA o las CPLD's de XILINX
4. Mar de Puertas, como es el caso de ORCA.

2.3.2. Sintetizado hardware desde lenguajes de alto nivel

Los lenguajes de descripción hardware utilizados para la síntesis de las FPGAs tienen una mayor dificultad que los lenguajes de alto nivel como Matlab o C/C++, por lo que desde hace unos pocos años han aparecido herramientas que permiten la síntesis hardware a más alto nivel como el System-C y otras herramientas que permiten convertir código C/C++ para dispositivos empotrados en código de descripción hardware como el VHDL o Verilog. Gracias a estas nuevas herramientas, se logra realizar la síntesis de algoritmos con menor esfuerzo pagando como contrapartida una menor eficiencia en los resultados obtenidos.

La implementación de algoritmos matemáticos, suelen realizarse en lenguajes de muy alto nivel, en dónde destaca la utilización del Matlab por gran parte de la comunidad investigadora. Matlab permite una programación rápida de prototipos que están muy alejados de implementaciones hardware. Por este motivo, los desarrolladores de Matlab han trabajado en la realización de una *Toolbox* que permite la generación de código C/C++ para sistemas empotrados. Este hecho abre la puerta a metodologías de sintetizado rápido de códigos en Matlab ya que este código puede ser compilado a C/C++ y posteriormente sintetizado para FPGAs mediante la utilización de herramientas como el Catapult C.

3.Herramientas utilizadas

Para la síntesis hardware del algoritmo N-FINDR programado en Matlab se ha hecho uso de múltiples herramientas software y de programación que se van a describir en este capítulo. Entre estas herramientas destacan la IDE de programación matemática Matlab con sus *Toolbox*es, la librería para punto fijo de C++ *fixed*, la IDE de programación Code::Block y la herramienta para la síntesis hardware a partir de código C/C++ Catapult-C.

3.1. Matlab

Matlab constituye actualmente un estándar de facto dentro de las herramientas del análisis numérico, tanto por su gran capacidad y sencillez de manejo como por su enorme versatilidad y difusión.

Matlab (*Matrix Laboratory*) es un lenguaje de programación técnico-científico que básicamente trabaja con variables vectoriales y matriciales. Es fácil de utilizar debido a que contiene varias cajas de herramientas con funciones incorporadas (*Toolbox* de

procesamiento de señales, herramientas de C embebido, matemática aritmética de punto fijo, etc.). Dos *Toolbox*es esenciales en el desarrollo de este TFM son el *Embedded* y el *Fixed-Point*, estas herramientas permiten la generación de código C/C++ para sistemas empotrados a partir de código Matlab compatible y la utilización de aritmética de punto fijo en los algoritmos. A continuación se va a proceder a describir estas *Toolboxes* con más detalle.

3.1.1. Toolbox Embedded Matlab

La *Toolbox* de *Embedded* de Matlab permite generar código C embebido para dispositivos empotrados a partir de un subconjunto de funciones compatibles de Matlab [16]. El código generado en C cumple con los requisitos de tiempo real para la memoria y de variables locales.

Para hacer uso del *Embedded* C de Matlab es necesario instalar y configurar un compilador de C++, dependiendo del sistema operativo en que se ejecuta. Para el caso de Windows 64 bits es necesario instalar la versión Visual Studio de Microsoft.

En este TFM se ha utilizado el Visual Studio 2008. Para poder integrar el compilador en Matlab ha sido necesario instalar el Microsoft Visual Studio 2008 Service Pack 1 y posteriormente el Microsoft Windows SDK for Windows 7 y el .NET Framework 3.5 SP1. Una vez instalado el software se ha procedido a la configuración desde Matlab para su uso.

Para configurar el compilador que va a ser utilizado en el *Embedded* de Matlab es necesario configurar las utilidades de compilación de C en Matlab, esto se realiza con el comando *mex*. Tecleando *mex –setup* podemos configurar el compilador que se va a utilizar. Matlab detecta automáticamente los compiladores compatibles instalados en el sistema operativo y te permite seleccionar cual va a ser utilizado.

En la Figura 3.1. se muestra el prompt de Matlab dónde se puede ver el proceso de configuración del compilador de Visual Studio con el comando *mex*.

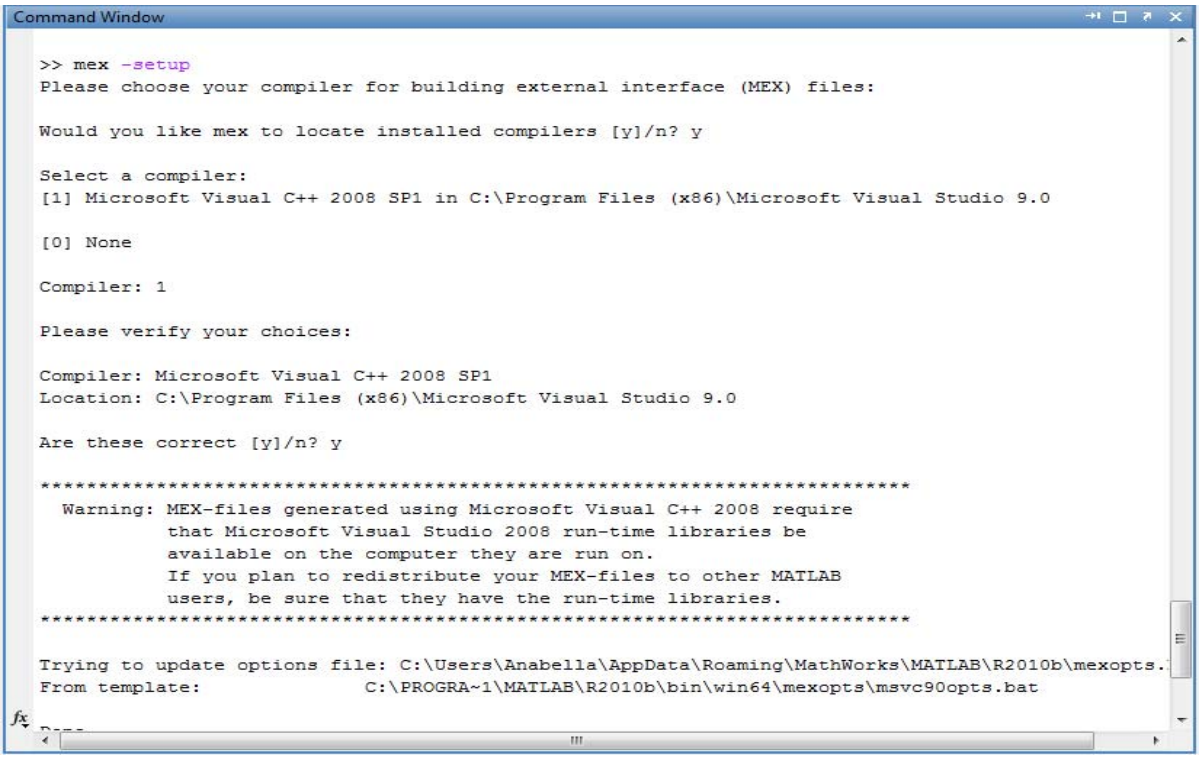


Figura 3.1. Configuración del compilador de C/C++ utilizado en Matlab mediante el comando mex

Una vez configurado el compilador, ya se está en disposición de poder utilizar la *Toolbox* de *Embeddeb* Matlab. En posteriores capítulos se describe con detalle los pasos a seguir para la generación del código C.

3.1.2. Toolbox Fixed-Point

Un número en punto fijo representa un valor de tipo real mediante un número entero en el que se ha fijado un número de dígitos antes del punto decimal y otro número de dígitos después de este punto. El uso del punto fijo, en lugar del coma flotante, es útil para la representación de valores reales cuando el hardware que procesa los datos carece de una unidad de procesado de números en coma flotante (FPU). El punto fijo proporciona mejor rendimiento que la coma flotante en estos dispositivos pequeños de bajo rendimiento.

Un ejemplo que ilustra el funcionamiento del punto fijo es la utilización de enteros decimales para representar números reales de dos decimales. Por lo tanto podemos definir la variable A y B.

Real A = 2,5 → Entero 250 (escala 100)

Real B = 10,00 → Entero 1000 (escala 100)

Para realizar sumas y restas se realiza de la siguiente forma.

Real C = 2,5 + 10 = 12,5 → Entero 250 + 1000 = 1250 (escala 100)

Real C = 10 – 2,5 = 7,5 → Entero 1000 – 250 = 750 (escala 100)

Por lo tanto la suma y las restas en punto fijo son idénticas a su equivalente en entero. Para operar multiplicaciones y divisiones se realiza de la siguiente forma.

Real C = 2,5 * 10 = 250 → Entero 250 * 1000 = 250000/100 = 25000 (escala 100)

Real C = 2,5 / 10 = 0,25 → Entero (250/1000)/*100 = 25 (escala 100)

Por lo que podemos observar cómo, para el producto se ha de dividir el resultado por la escala y en la división, se ha de multiplicar previamente por este valor. Estas operaciones han de ser realizadas en el punto fijo para mantener el número de dígitos decimales almacenados.

Para el caso binario el sistema es idéntico, el único cambio reside en la aritmética binaria. Para una variable de 64 bits se suele dejar un valor de escala suficiente que permita almacenar datos enteros con suficiente resolución. Por ejemplo, si se utiliza un 64x16, indica que los últimos 16 bits están dedicados para la parte decimal por lo tanto podemos tener una resolución máxima de 2^{-16} = 0,0000152588. Por el lado contrario se puede almacenar en la parte entera 48 bits, o lo que es lo mismo a 2^{48-1} = 140.737.488.355.328, además de un bits de signo. De esta manera se describen los dos problemas de la aritmética de punto fijo, el primero es que hay una resolución máxima para la parte decimal, por debajo de ese valor no se puede medir. El segundo problema es el desbordamiento, que indica que si el número es superior al máximo almacenamiento se producirá un error en el uso del punto fijo.

Para definir la escala del punto fijo se utilizan números potencia de dos, dado que se está ejecutando en un dispositivo electrónico el que multiplicar y dividir en potencias de dos se traduce en desplazamientos binarios que son operaciones simples para los dispositivos electrónicos.

Las *Toolbox* de Matlab *Fixed-Point* proporciona el uso de la aritmética de punto fijo para una multitud de funciones compatibles. El punto fijo de Matlab se implementa en una clase que puede almacenar números en punto fijo de tamaños casi ilimitados. Además el punto fijo de Matlab gestiona las operaciones entre las diferentes variables que pueden tener tamaños y escalas diferentes. *Fixed-Point* proporciona herramientas de configuración del comportamiento de los *Fixed-Point* y de la aritmética asociada.

Finalmente el uso de *Fixed-Point* está soportado, aunque con ciertas limitaciones, por la *Toolbox* de *Embedded* Matlab, por lo que se puede hacer uso del punto fijo en la generación de código C desde Matlab. A continuación, se muestran varias forma de inicialización de las variables de punto fijo:

```
A1 = fi(3.1416)
A2 = fi(pi, 1, 8) % word length 8 bits, and fraction length best Precision
```

3.2. Librería C++ de punto fijo

En este TFM se ha explorado la posibilidad de hacer uso de librerías de punto fijo en el lenguaje C/C++ que nos permita eliminar el uso de la coma flotante, que es complicada de sintetizar en los dispositivos electrónicos programables FPGAs. Realizando una búsqueda por las librerías existentes en Internet nos encontramos con un amplio conjunto de ellas, de las que destaca la librería en C/C++ Libfixmath y la librería de C++ *fixed* [17].

Estas librerías utilizan el potencial de C++ para proporcionar una envoltura a la aritmética de punto fijo gracias a que C++ permite programar y sobrecargar los diferentes operadores matemáticos y lógicos como la suma, la resta, el producto, la división, la comparación, etc.

De esta forma se puede utilizar la aritmética de punto fijo de la misma forma que se utiliza la aritmética entera o flotante. Las librerías de punto fijo proporcionan operaciones de mayor complejidad como la raíz cuadrada, la potencia u operaciones trigonométricas, lo que permite realizar la gran mayoría de los algoritmos matemáticos.

La limitación que introduce el punto fijo respecto al *overflow* y la falta de resolución también está presente en las implementaciones realizadas en C++, más aún si se tiene en

cuenta que se utilizan enteros de 32 bits para almacenar los datos. Este tamaño no es adecuado para los cálculos que se pretenden realizar en nuestro algoritmo por lo que se reescribió la librería de C++ para que utilice enteros largos de 64 bits, con esto se pueden obtener mejores resultados en la implementación del algoritmo.

Se ha elegido la librería *fixed* para su modificación dado que su código es el más sencillo en cuanto a la programación y a que ha sido programado íntegramente en C++. La librería consta únicamente de dos archivos fuentes (.cpp y .h) que define el total funcionamiento del tipo de dato numérico en punto fijo. A continuación se muestran los prototipos de métodos que proporciona la librería para su uso.

```
fixed(const fixed& fixedVal);
fixed(const fixed* fixedVal);
fixed(long long int nVal);
fixed operator++(void);
fixed operator++(int);
fixed operator--(void);
fixed& operator=(fixed fixedVal);
fixed& operator=(int intVal);
bool operator==(fixed fixedVal);
bool operator==(int intVal);
bool operator!=(fixed fixedVal);
bool operator!=(int intVal);
bool operator<(fixed fixedVal);
bool operator<(int intVal);
bool operator<=(fixed fixedVal);
bool operator<=(int intVal);
```

```
bool operator>(fixed fixedVal);
bool operator>(int intVal);
bool operator>=(fixed fixedVal);
bool operator>=(int intVal);
operator double(void);
operator float(void);
operator int(void);
fixed floor(void);
fixed ceil(void);
fixed operator+(int b);
fixed operator-(int b);
fixed operator*(int b);
fixed operator/(int b);
fixed operator+(fixed b);
fixed operator-(fixed b);
fixed operator*(fixed b);
```

Gracias a la implementación de los operadores anteriores, la sintaxis de la aritmética en punto fijo para esta librería en C++ queda de la siguiente manera.

```
fixed a = 3.1416;
fixed b = (a+4.5)/2.14;
if(a <= b) b = sin(a);
float c = (float) b;
```

3.3. IDE de programación C++ Code::Block

Code::Block es un entorno de programación integrado libre y multiplataforma para el desarrollo de aplicaciones realizados en C++. Permite la utilización de diversas librerías gráficas multiplataforma para la implementación de programas con interfaz gráfica en diferentes sistemas operativos.

Code::Block, al igual que las demás IDEs, soporta varias utilidades de apoyo a la programación como el coloreo de sintaxis, auto completado de los métodos, tabulación inteligente de código, etc. Además, presenta un navegador de proyectos: vista de archivos, símbolos (heredados, etc.), clases y recursos. También dispone de menús de configuración de los proyectos C++, su compilación y ejecución.

Una de las características más importantes de Code::Block son los complementos (*plugins*) que permite que la IDE sea muy dinámica y potente. Code::Block permite enlazarse a una gran variedad de compiladores, desde compiladores como el Microsoft Visual Studio Toolkit, pasando por el compilador de Borland C++ Compiler, Intel C++ Compiler y terminando por el compilador GNU (GCC, G++). Este último en sus versiones para Linux/Unix y Windows (MinGW). En la siguiente Figura 3.2. se muestra la IDE de programación Code::Block.

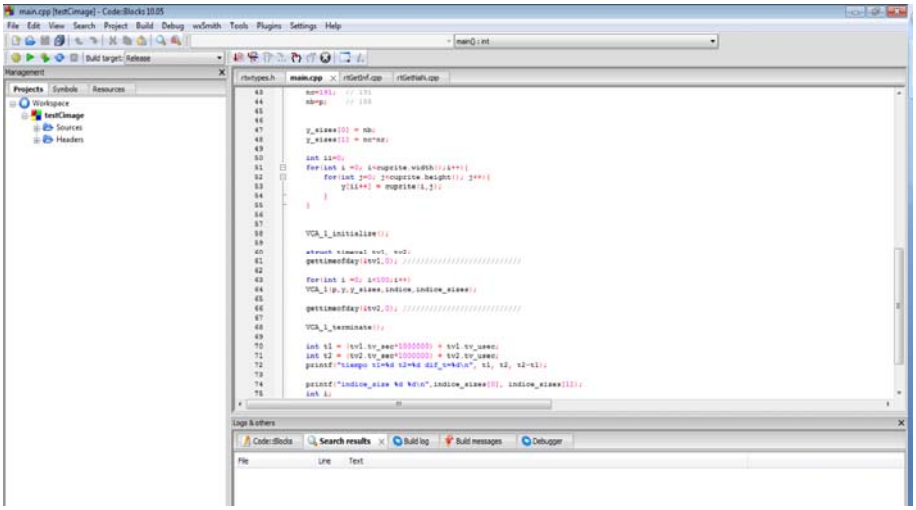


Figura 3.2. IDE de programación Code::Block

3.4. Herramienta se síntesis de alto nivel Catapult C

Catapult C [18] es una herramienta de sintetizado hardware de alto nivel. Es un software comercial producido por Mentor Graphics para la utilización de código C/C++ y genera salidas RTL en código VHDL y Verilog, entre otros, para la síntesis hardware de FPGAs y ASICs. Catapult C permite a los usuarios indicar las restricciones de tiempo y área indicando la frecuencia de reloj a utilizar y la tecnología de destino.

La utilización básica de Catapult no resulta complicada y sólo se requiere de la inserción de los archivos .cpp para iniciar el proceso. Posteriormente, Catapult C comprueba la corrección del código fuente teniendo que tener especial cuidado con el uso de funciones matemáticas incluidas en la cabecera math.h. En el caso de que alguna función no esté soportada, como por ejemplo *fabs()* se ha de implementar para su posterior uso. Una vez que el código es correcto, se continúa con la configuración del diseño, en donde se selecciona el dispositivo a sintetizar y la frecuencia de reloj que se quiere utilizar. El siguiente paso, Catapult genera las restricciones de la arquitectura seleccionada. Una vez finalizado este paso se pasa al organizador de la síntesis y para finalizar se realiza el paso de generación del lenguaje de descripción hardware RTL seleccionado.

Catapult C proporciona integrado un paquete llamada *Precision* que permite realizar los pasos necesarios para la síntesis del código RTL generado. Gracias a que obtiene los datos necesarios del dispositivo seleccionado de Catapult C solo se ha de pulsar el botón de sintetizado para generara automáticamente todo el proceso.

Se debe destacar que los tiempos de generación del código RTL y de la síntesis posterior son muy prolongados pudiéndose necesitar días para la finalización de este proceso.

En la figura 3.3 y 3.4 se muestra el aspecto que presenta el programa Catapult C y *Precision*.

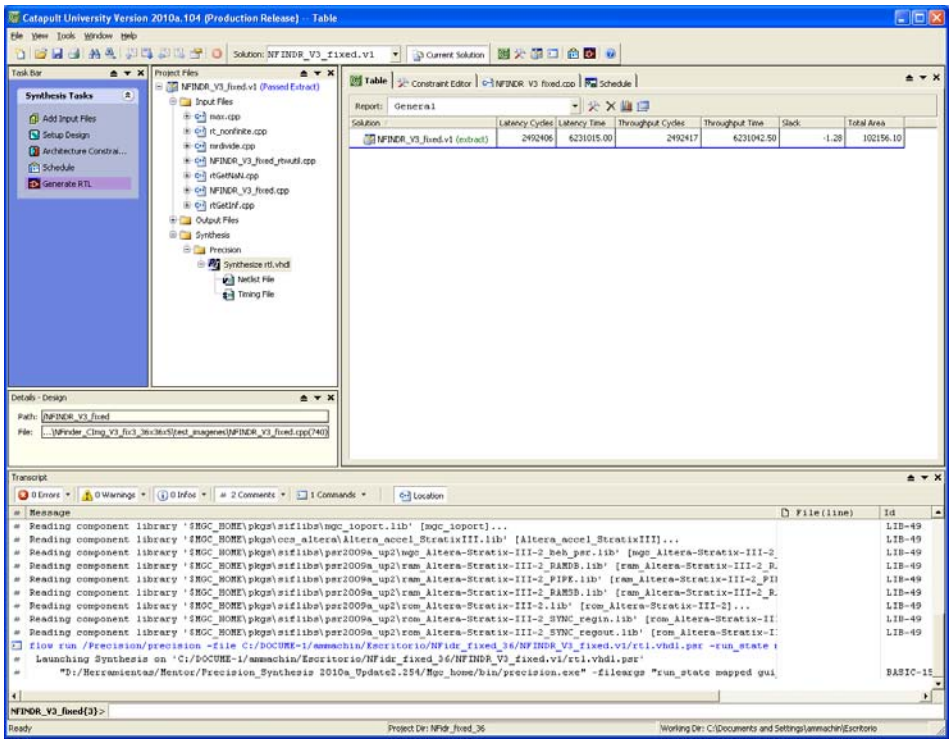


Figura 3.3. Interfaz de la herramienta Catapult C

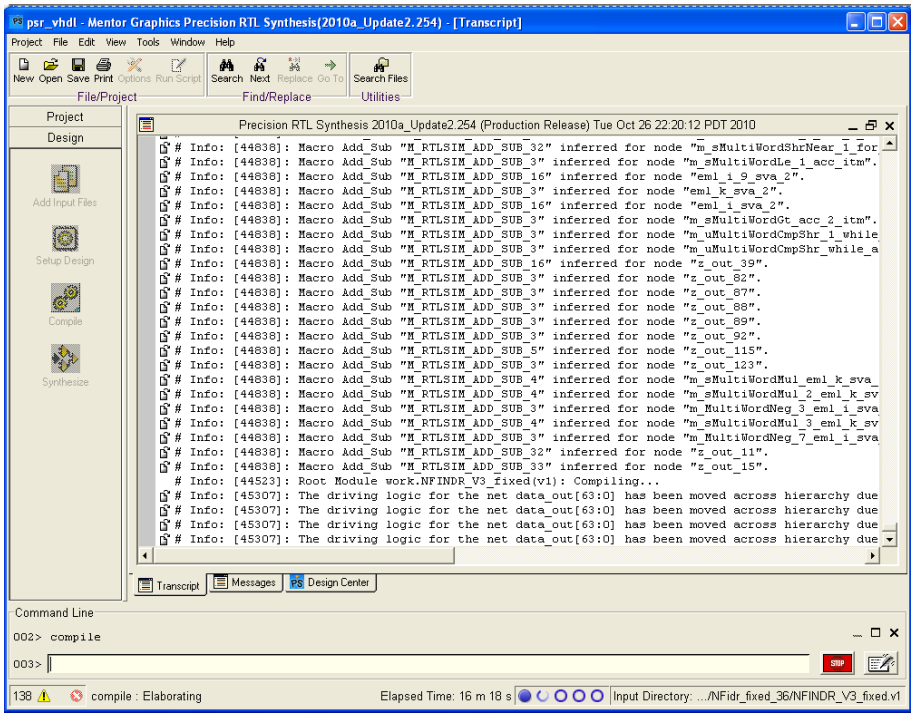


Figura 3.4. Interfaz de la herramienta *Precision*

4. Trabajo realizado y descripción de la metodología utilizada

En este capítulo, se va a describir de forma detallada, el trabajo realizado en este TFM para llevar a cabo el síntesis, en un lenguaje de descripción hardware, de un algoritmo de extracción de *Endmembers* como es el N-FINDR.

Para ello, se parte de un código Matlab dado, que implementa dicha algoritmo. El objetivo, es desarrollar una metodología alternativa, a la implementación directa del código en lenguajes a más bajo nivel hardware como VHDL o Verilog.

Con el objeto de plantear esa metodología, se estudian diferentes estrategias, todas fundamentadas en obtener un algoritmo en código C++, a partir del código Matlab, que introducido en la herramienta Catapul C, nos genere un código VHDL o Verilog con distintas características.

El trabajo se ha enfocado en buscar el mejor camino para generar dicho código C++, variando los tipos de datos y la aritmética de operación, que en el proceso final ofrezca mejores prestaciones, en cuanto a frecuencias de funcionamiento, *throughput*, área, etc. en la implementación sobre un dispositivo FPGA.

4.1. Descripción del trabajo realizado

El trabajo realizado en este TFM, se ha desarrollado de una manera incremental. El punto de partida ha sido una implementación del algoritmo N-FINDR desarrollada en código Matlab, el cual se ha versionado de cuatro maneras diferentes. A continuación, se describen los fundamentos principales de cada versión.

1. ***Versión 1. Algoritmo N-FINDR original***. En esta versión se parte del código original. Se han eliminado las inicializaciones aleatorias, se ha reestructurado el código y se utiliza el tipo *double* para los datos de entrada. El código C++ se genera con *Embeddeb* Matlab.
2. ***Versión 2. Algoritmo N-FINDR en fixed point***. En esta versión, se parte del código utilizado en el punto anterior, pero en este caso, se redefine para la utilización de aritmética de punto fijo, utilizando la librería *Fixed Point* de Matlab.
3. ***Versión 3. Algoritmo N-FINDR en aritmética de parte entera.*** En esta versión, también se parte del código del punto uno, pero se redefinen los tipos de datos y funciones, para utilizar aritmética entera.
4. ***Versión 4. Algoritmo N-FINDR en fixed de C++.*** Por último, en esta versión, se parte del mismo código C++ generado por *Embeddeb* Matlab, y se redefinen los tipos de datos a punto fijo, utilizando para ello la clase *fixed* de C++.

4.1.1. Versión 1. Sintetizado hardware del algoritmo N-FINDR original.

En este apartado, como ya se ha adelantado, se va a describir el trabajo realizado para conseguir la síntesis en un lenguaje de descripción hardware, del algoritmo de extracción de *Endmembers* en imágenes hiperespectrales, N-FINDR. Para ello, se ha partido de una función dada del algoritmo en código Matlab.

El procedimiento a seguir es generar el código C++ de dicha función a partir de la utilidad que ofrece la *toolbox* de Matlab, llamada *Embeddeb* Matlab. Una vez generado el código C++, se valida a través de la IDE de programación Code::Block. Este procedimiento no es inmediato, y requiere de una reestructuración del código Matlab.

Finalmente, conseguido el código en lenguaje C++, se procede a realizar la síntesis hardware del mismo. Para ello, utilizando la herramienta Catapult C, se generara el código de descripción hardware como VHDL o Verilog y a partir del paquete *Precision* se realiza la síntesis optimizada del archivo RTL generado por Catapult C, en el dispositivo seleccionado.

A continuación se describe con más detalle el trabajo realizado.

***Generación del código C++ usando Embeddeb* Matlab**

El código Matlab del N-FINDR del que se parte originalmente es el de la Figura 4.1:

```
function [P,vStart,vEnd,it,ptime] = N-FINDR(MNFHIM,p,P,FN,maxit,Print);
% AUTHORS: Maciel Zortea & Antonio Plaza, University of Extremadura, Spain
% mailto: mzortea@gmail.com
disp(' === Start N-FINDR_WinterInputReduction_v4 run ===')
% Obtener tiempo CPU actual
t1=cputime;
[nr,nc,nb] = size(MNFHIM);
MatrixTest = zeros(p,p);
MatrixInicial = zeros(p-1,p-1);
for k = 1:p
    %     row = floor(rand*nr+1);
    %     line = floor(rand*nc+1);
    row = P(k,1);
    col = P(k,2);
    disp(sprintf('Start with random pixel at position row = %0.5d | col %0.5d:',row,col))
    MatrixInicial(:,k) = squeeze(MNFHIM(row,col,1:p-1));
end
MatrixTest(1,:) = 1;
MatrixTest(2:p,:) = MatrixInicial;
vStart = abs(det(MatrixTest)); %should be volumeactual = abs(det(MatrixTest))/(factorial(p-
       1));
disp(sprintf('. Maximun # of iterations allowed (classic) = %d',maxit))
it = 0;
v1 = -1;
v2 = vStart;
volumeactual = vStart;
Matrix = MatrixTest;
while and(it<maxit,v2>v1)
    it = it+1;
    disp(' ')
    disp(sprintf('. Start iteration # [%d] of max = [%d], with abs(det(E)) =
```

```
        %4.8g',it,maxit,v2))
    disp(sprintf('  Functional abs(det(E)): @ v1 = %8.8g  @ v2 = %8.8g , Ratio: v2/v1 =
        %8.8g ',v1,v2,v2/v1))
    for i = 1:nr
        for j = 1:nc
            %pixelactual = squeeze(MNFHIM(i,j,1:p-1));
            pixelactual = reshape(MNFHIM(i,j,1:p-1),[p-1 1]);
            volume = zeros(1,p);
            for k = 1:p
                MatrixTest = Matrix;
                MatrixTest(2:p,k) = pixelactual;
                volume(1,k) = abs(det(MatrixTest));  % should be volume =
        abs(det(MatrixTest))/(factorial(p-1));
            end
            [volume,pos]=max(volume);
            if volume > volumeactual
                disp(sprintf('---> update endmember -%d- with pixel @            (%5d,%5d)
        | abs(det(E)) = %8.8g ',pos,i,j,volume))
                volumeactual = volume;
                P(pos,1) = i;
                P(pos,2) = j;
                Matrix(2:p,pos) = pixelactual;
            end
        end
    end
    disp(sprintf('. End of iteration # [%d] of max = [%d]: abs(det(E)) =
        %8.8g',it,maxit,volumeactual))
    v1 = v2;
    v2 = volumeactual;
    if Print == 1
        figure
        imagesc(mean(MNFHIM(:,:,1:p-1),3)); colormap(gray);
        set(gca,'DefaultTextColor','black','xtick',[],'ytick',[],'dataaspectratio',[1 1 1])
        for i=1:size(P,1)
            drawnow;
        text(P(i,2),P(i,1),'o','Margin',1,'HorizontalAlignment','center','FontSize',22,'Font
        Weight','light','FontName','Garamond','Color','green');
        %text(P(i,2),P(i,1),num2str(i),'Margin',1,'HorizontalAlignment','center','FontSize',
        22,'FontWeight','light','FontName','Garamond','Color','green');
        end
        % print map
        FNdisk = sprintf('%s_pos_end_N-FINDR_iteration_%d',FN,it);
        print('-dpng','-r300',strcat(FNdisk,'.png'))
    end
end
if it<maxit
    disp(sprintf(' ===>>> Convergence @ iteration # [%d] of [%d]. Final abs(det(E)) =
        %8.8g',it,maxit,volumeactual))
    disp(sprintf(' ===>>> Final abs(det(E)) = %8.8g',volumeactual))
else
    %    disp(sprintf(' End, NO convergence @ iteration # %d. The abs(det(E)) =
        %8.8g',nit,volumeactual))
end
 disp(sprintf('. The N-FINDR solution [row x col] after %d iteration(s) is:',it))
% Obtener tiempo CPU actual
```

```
t2=cputime;
ptime = t2-t1;
vEnd = volumeactual;
 % Mostrar tiempo total en ejecucion del algoritmo
disp(sprintf('. Total CPU processing time ..... %6.3f [s]  ',ptime));
disp(' === End N-FINDR_WinterInputReduction_v4 ===');
```

Figura 4.1. Código Matlab del N-FINDR original.

Antes de generar el código C++, la librería *Embeddeb* presenta ciertas incompatibilidades con el código inicial, lo cual implica realizar ciertas modificaciones sobre el mismo para eliminar todas aquellas funciones que no estén soportadas.

En primer lugar, lo más evidente es eliminar la líneas dónde se hacen llamadas a la función "*disp*" que muestran una cadena de texto sobre el *prompt* de Matlab, así como todas aquellas llamadas a funciones que tengan que ver con la generación de gráficas o figuras.

Es necesario inicializar todas las variables que se asignan directamente en el código, ya que si no generarán errores en la compilación con *Embeddeb*.

Además, si se utilizan variables de entrada con tamaños variables, es necesario establecer una condición de tamaño máximo. Esto es posible utilizando la función *assert* que evalúa el tamaño de la variable y genera un error en caso de que se vulnere la condición.

El siguiente paso es algo más complejo, y viene determinado por las características concretas de este código, que para el cálculo del algoritmo del N-FINDR, trabaja con imágenes en tres dimensiones, es decir imágenes de MxNxP, dónde M representa las filas de una matriz, N las columnas y P las bandas de la imagen.

Después de realizar ciertas pruebas con el código inicial, se comprobó que el código generado en C++ no estaba interpretando los datos siguiendo la misma estructura que Matlab, así que se tomó la decisión de sobreescribir el código para trabajar con las imágenes en dos dimensiones.

Finalmente, el código queda en la Figura 4.2:

```
function [indice] = N-FINDR_V3(MNFHIM,p,maxit) %#eml
assert (p <= 7);
assert (maxit <= 10);
[nb np] = size(MNFHIM);
```

```
% No random initialization of the matrix with the position of endmember
indice = zeros(1,p,'int32');
for i=1:p
    indice(i) = (i*(np/p));
end
MatrixInicial = zeros(p-1,p-1);
for c = 1:p
    for f = 1:p-1
        MatrixInicial(f,c) = MNFHIM(f,indice(c));
    end
end
MatrixTest    = zeros(p,p);
for c = 1:p
    for f = 1:p
        if f > 1
            MatrixTest(f,c) = MatrixInicial(f-1,c);
        else
            MatrixTest(f,c) = 1;
        end
    end
end
vStart = abs(det(MatrixTest));
%volumenes auxiliares
v1 = -1;
v2 = vStart;
volumeactual = vStart;
Matrix = MatrixTest;
it = uint16(0);
pixelactual = zeros(p-1,1);
while and(it<maxit, v2>v1)
    it = it+1;
    for ii = 1:np
        %pixelactual = reshape(MNFHIM(1:p-1,ii),[p-1 1]);
        for f = 1:p-1
            pixelactual(f) = MNFHIM(f,ii);
        end
        volume = zeros(1,p);
        for k = 1:p
            MatrixTest = Matrix;
            for f=2:p
                MatrixTest(f,k) = pixelactual(f-1);
            end
            volume(1,k) = abs(det(MatrixTest));
        end
        [volume,pos]=max(volume);
         if volume > volumeactual
           volumeactual = volume;
           indice(pos) = ii;
           for f=2:p
               Matrix(f,pos) = pixelactual(f-1);
           end
       end
    end
    v1 = v2;
    v2 = volumeactual;
```

```
end
end
```

Figura 4.2. Código Matlab del N-FINDR modificado.

Como se puede comprobar, el código ha sido reestructurado casi en su totalidad. Cabe añadir, que no es conveniente utilizar las referencias *":"* que se utilizan en Matlab para dirigirse a todos los elementos de un vector fila o columna de una matriz, porque puede dar problemas al generar el código en C++.

Además, en el algoritmo inicial se inicializaba aleatoriamente un vector de índices. En este caso, hemos eliminado esa aleatoriedad inicializando el vector con valores equidistanciados con el fin de obtener siempre resultados comparables en cada ejecución.

El siguiente paso para generar el código C++ es generar un objeto de configuración *Real Time Workshop* (*RTW*), sobre el que editamos las opciones que queremos utilizar para generar nuestro código. En este caso, se ha seleccionado el lenguaje C++, se ha desactivado la generación de un *makefile* y se han mantenido activadas las opciones de tamaño variable de las variables y la opción de saturación de enteros en caso de *overflow*. El resto de opciones se han mantenido en su estado por defecto. Además, ante de generar el objeto, tenemos que crear la función *main.cpp*, que posteriormente llamará a la función principal. Las rutinas utilizadas son las siguientes:

```
rtwcfg = emlcoder.RTWConfig        % Se crea el RTW y se llama rtwcfg
rwcfg.CustomSource = 'main.cpp'    % Se define la función main
open rtwcfg                        % Se edita el objeto de configuración
```

A continuación, para poder compilar la función con *Embeddeb* Matlab es necesario añadir al principio del código la directiva *%#eml*. El propósito de esta directiva es que Matlab ofrece, en el caso de que la compilación no haya ido bien, un diagnóstico detallado de los fallos ocurridos durante la compilación, comprueba todas las funciones incompatibles y los posibles errores de sintaxis.

Finalmente, el último paso para generar el código C++ a partir del código Matlab, es hacer la llamada a *Embedded* Matlab. Esto se realiza con la rutina *emlc* seguida de la especificación de una serie de parámetros. Estos parámetros indican el objeto de

configuración utilizado, los archivos que se desean generar (*.cpp, *.h, *.lib, *.obj), el nombre y ruta de los archivos que se vayan a generar, las opciones de optimización del código y los tipos y tamaños de las variables de entrada a la función. Esta última opción suele se especificada introduciendo una serie de variables de ejemplo en la llamada.

La sentencia utilizada en este caso es la siguiente:

```
emlc -s rtwcfg -d nfinder N-FINDR.m -eg {emlcoder.egs('double', [188 47750]), uint16(0),uint16(0)} -report
```

dónde,

- `-s rtwcfg`, indica el objeto de configuración utilizado.
- `-d nfinder`, indica el nombre del fichero donde se almacenarán todos los archivos generados.
- `N-FINDR.m`, indica el nombre la de función en código Matlab que se quiere generar en código C++.
- `-eg{emlcoder.egs('uint8',[188 47750]),uint16(0),uint16(0)}`, indica un ejemplo del tipo y tamaño de las variables de entrada a la función. En este caso, se trata de una matriz de de datos tipo uint8 y dimensiones 188x47750, y dos enteros de tipo uint16.
- `-report`, indica que se muestre un informe de resultados.

A continuación, la Figura 4.3 el aspecto que presenta el *report* devuelto por *Embeddeb* Matlab.

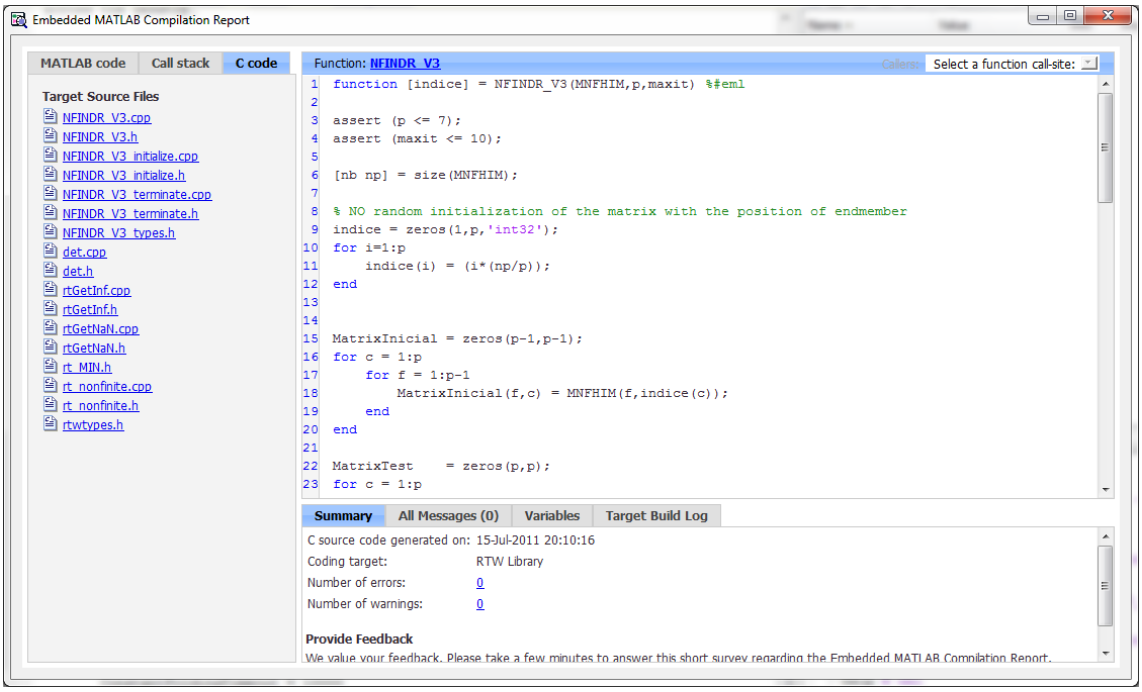


Figura 4.3. Informe de la compilación con *Embeddeb* Matlab.

Una vez compilado el código con éxito, se generan los archivos de código fuente con extensión .cpp y los archivos de cabecera con extensión .h. De estos, caben destacar los siguientes:

- N-FINDR_V3.cpp
- N-FINDR_V3.h
- N-FINDR_V3_initialize.cpp
- N-FINDR_V3_initialize.h
- N-FINDR_V3_terminate.cpp
- N-FINDR_V3_terminate.h
- N-FINDR_V3_types.h
- det.cpp
- det.h
- rtwtypes.h

Estos ficheros definen el código de la función principal, así como de las funciones auxiliares que se utilizan en su implementación, como es el caso del determinante, dónde podemos ver que se ha generado un código independiente.

En los archivos *initialize* y *terminate* se define el código que se quiere ejecutar antes y después de llamar a la función principal desde el *main.cpp*. Por ejemplo, para generar o liberar recursos.

Por otro lado, existen ficheros, como el *rtwtypes.h*, que definen los tipos de datos y su equivalencia en la conversión del lenguaje. Un ejemplo de estas definiciones, extraídas de ese archivo, se muestra a continuación:

```
/*=======================================================================*
 * Generic type definitions: real_T, time_T, boolean_T, int_T, uint_T,
 *                           ulong_T, char_T and byte_T.
*=======================================================================*/
typedef double real_T;
typedef double time_T;
typedef unsigned char boolean_T;
typedef int int_T;
typedef unsigned uint_T;
typedef unsigned long ulong_T;
typedef char char_T;
typedef char_T byte_T;
/*=======================================================================*
```

Compilación del código usando Code::Block

Un vez se he generado el código C++ a partir del *Embeddeb* Matlab, procedemos a compilar los archivos en la IDE de programación Code::Block. Para ello, creamos un nuevo proyecto en el que añadimos todos los archivos .cpp y .h. Además creamos la función *main*, desde la cual llamamos a la función principal.

Definimos los tipos y tamaños de las variables de entrada y salida a la función. Los tipos se definen en función de los tipos de datos definidos en la cabecera de la función principal generada por *Embeddeb*.

En las funciones de código C++ generado por Matlab, los nombres de las variables se mantienen igual, pero se les añade la extensión *eml_* delante. Además, la disposición de la variables cambia y se crean nuevas variable que definen los tamaño de las mismas. En este caso, a la entrada *MNFHIM* le corresponden dos variables, una llamada

MNFHIM_data que almacena los datos de la imagen de entrada y otra llamada *MNFHIM_sizes,* dónde se define el tamaño de la imagen de entrada. Este último es un vector que almacena en el primer elemento el número de filas de la matriz de entrada y en el segundo elemento las columnas. Los mismo ocurre en el caso de la variables de salida *indice_data*, en la que se almacenan la posición de los *Endmembers* clasificados por el algoritmo.

Otro aspecto importante a tener en cuenta, es la utilización de los datos desde Code::Block, ya que podemos no estar tratando la lectura de los datos de la matriz en la misma disposición que en Matlab. Así, en el primer caso se hace una lectura vectorial de la imagen con una disposición en columnas.

A continuación, se muestra la cabecera de la función principal extraída del archivo N-FINDR.h y el correspondiente código de la función *main*.cpp.

```
//N-FINDR.h

void N-FINDR_V3 (real_T eml_MNFHIM_data[8977000], int32_T eml_MNFHIM_sizes[2], uint16_T
eml_p, uint16_T eml_maxit, int32_T eml_indice_data[30], int32_T eml_indice_sizes[2]);

//main.cpp

// función principal de la aplicación
int main()
{
    real_T y[8977000];
    int32_T y_sizes[2];
    uint16_T nr;
    uint16_T nc;
    uint16_T nb;
    uint16_T p;
    uint16_T maxit;
    int32_T indice[30];
    int32_T indice_sizes[2];
    p=5;
    maxit=10;
    nr=36;  // 250              //<---------
    nc=36;  // 191              //<---------
    nb=p;  // 188               //<---------
    y_sizes[0] = nb;
    y_sizes[1] = nc*nr;
    N-FINDR_V3_initialize();
    N-FINDR_V3(y,y_sizes,p,maxit,indice,indice_sizes);
    N-FINDR_V3_terminate();
}
```

Para comprobar los resultados obtenidos, se generó desde Matlab una serie de imágenes a tomar como referencia y se guardaron en formato .tiff con valores normalizados entre 0 y 255.

De estas imágenes, tres se generaron con valores aleatorios de dimensiones, 100x100x5, 100x100x10, 100x100x15 y una cuarta imagen real llamada cuprite procedente del sensor AVIRIS con dimensiones 250x191x19. A esta última, se le aplicó un procesado para reducir la dimensionalidad de los datos, ya que inicialmente constaba de 224 bandas. El último paso fue reestructurar su información, ya que la entrada a la función del algoritmo N-FINDR debe de ser una matriz, cuyas filas se correspondan con los valores de un mismo píxel en cada una de las bandas y las columnas se correspondan con cada unos de los píxeles que forman la imagen. Es decir, que finalmente las imágenes quedan con las siguientes dimensiones: 10000x5, 10000x10, 10000x15 y 47750x19.

Finalmente, se comprobó que datos obtenidos para los índices concordaban con lo esperado, de acuerdo a los resultados obtenidos desde la ejecución en Matlab.

Síntesis hardware con Catapult C

El siguiente paso fue la síntesis del código C++ en un lenguaje de descripción hardware como VHDL o Verilog. Para ello utilizaremos la herramienta Catapult C.

El procedimiento seguido, ha sido generar un nuevo proyecto al que hemos añadido los códigos fuente generados a partir de *Embeddeb* Matlab. Una vez introducidos todos los .cpp, procedemos a seguir los pasos preestablecidos en la herramienta para la síntesis, no sin antes indicar a Catapult cuál es la función principal que se desea sintetizar, para ello se añade el *pragma hls_design top* sobre la misma.

Configuramos las opciones de diseño, entre las que elegimos la tecnología utilizada. Se ha elegido como herramienta de síntesis *Precision* 2009a_up2 y como dispositivo una *Altera Stratix III*, hemos seleccionado todas las librerías compatibles, y como frecuencia de diseño hemos puesto 400MHz.

Antes de pasar al siguiente paso, nos sale un error debido a que Catapult no proporciona implementación de todas las funciones *math.h*. Así, en este caso, es necesario definir la

función *fabs*. Para resolver este problema, añadimos la definición de la misma en los archivos que se utilicen esta función. La definición utilizada, se muestra en la siguiente línea.

```
#define fabs(a) (((a) < 0) ? -(a) : (a))
```

Se comentan los *includes* al *math.h* original para evitar errores por múltiples definiciones de las funciones.

A continuación, procedemos a realizar el siguiente paso de *Architecture Constraint*, que tiene en cuenta las restricciones en la arquitectura. El siguiente paso es generar el esquema de diseño y finalmente se procede a generar el archivo RTL. Cabe decir que estos últimos pasos son muy lentos, y la herramienta Catapult ha necesitado aproximadamente 24 horas para su finalización.

Para obtener información de la bondad del código generado es necesario realizar la síntesis hardware del código mediante la utilidad incluida en Catapult denominada *Precision*. Este software permite sintetizar, de forma optimizada para un dispositivo concreto, tanto código VHDL como Verilog para las FPGAs de Xilinx y Altera. Los paso necesarios son la compilación y la síntesis del código. Los datos de configuración por defecto son los definidos anteriormente en Catapult. Este proceso puede durar hasta 6 horas. Una vez finalizado, podemos obtener información detallada sobre la frecuencia y latencia de la síntesis, así como del área y recursos utilizados de la FPGA.

4.1.2. Versión 2. Sintetizado hardware del algoritmo N-FINDR en fixed point.

En este apartado se va a describir el procedimiento seguido para la síntesis en un lenguaje de descripción hardware como VHDL o Verilog, del algoritmo N-FINDR. Partimos del mismo algoritmo utilizado en el apartado anterior. El procedimiento a seguir es muy similar, sin embargo, en este caso se van a tratar los datos en aritmética de punto fijo. Para ello, se va a utilizar la *toolbox* que ofrece Matlab de punto fijo, llamada *Fixed Point Toolbox*.

Matlab ofrece una amplia variedad de funciones para realizar operaciones con este tipo de datos, sin embargo también presenta muchas limitaciones.

El objetivo de esta implementación en punto fijo es facilitar el cómputo en los dispositivos hardware empotrados que no disponen de unidades de procesamiento de datos en aritmética de coma flotante. Por lo tanto, se pretende reducir el coste computacional y aumentar la velocidad de las operaciones.

Generación del código C++ usando Embeddeb* Matlab *con fixed point

En este apartado, vamos a exponer el trabajo realizado para generar el código C++ en aritmética de punto fijo.

El primer inconveniente que encontramos a la hora de compilar el código C++ con Matlab, es la limitación que se presenta en las funciones que no están implementadas para este tipo de datos en punto fijo. En el caso del algoritmo N-FINDR se ha tenido que buscar una alternativa al cálculo del determinante, ya que la función *det()* de Matlab, no está implementada para aritmética de punto fijo.

La alternativa elegida para generar el determinante ha sido la transcripción a código Matlab de un código C++ implementado para el cálculo del determinante de una matriz bidimensional. El código utilizado se muestra en la Figura 4.4 :

```
function de = deter(n)
   [nf nc] = size(n);
   if (nf ~= nc)
        %disp( 'error: Matrix must be square' );
        return;
   end;
   de = n(1,1);
   for k=1:nf-1
      l=k+1;
      for i=l:nf
        for j=l:nf
            n(i,j) = ( n(k,k)*n(i,j) - n(k,j)*n(i,k) )/n(k,k);
        end
      end
      de = de*n(k+1,k+1);
   end
end
```

Figura 4.4. Cálculo del determinante.

El siguiente paso, es generar los datos de tipo *fixed point.* Para ello se hace uso de la función *fi()* de Matlab. En la generación de un objeto de punto fijo, se pueden configurar

una serie de parámetros. Para ello, se editan las propiedades de unos objetos, *fimath* y *numerictype*, que se generan automáticamente cuando se crea un objeto de tipo *fixed point*.

A continuación, se muestran las principales propiedades editables de estos objetos:

Propiedades del objeto fimath

Estas propiedades son siempre editables.

- *MaxProductWordLength*, se define la máxima longitud de palabra permitida para el producto de datos.
- *MaxSumWordLength*, se define la máxima longitud de palabra permitida para la suma de datos.
- *ProductFractionLength*, se define la longitud de la parte decimal, en bits, en el producto de datos.
- *ProductMode*, se define como es determinado el producto de datos.
- *ProductWordLength*, se define la longitud de palabra, en bits, del producto de datos.
- *SumWordLength*, se define la longitud de palabra, en bits, de la suma de datos.
- *SumMode*, se define como es determinado la suma de datos.

Propiedades del objeto numerictype

- Estas propiedades no son editables una vez generado el objeto de tipo *fixed point*. Sin embargo, se puede crear una copia de este objeto asignando un *numerictype* distinto.
- WordLength, se define la longitud de palabra, en bits, de la parte entero almacenado del objeto *fixed point*.
- *FractionLength*, se define la longitud de la parte decimal, en bits, del valor entero almacenado del objeto *fixed point*.

En el caso de no editar ningunas de las propiedades mencionadas, Matlab utiliza por defecto una configuración, en la que recalcula a tiempo real, el valor asignado a cada una

de las propiedades para optimizar los resultados en las operaciones de aritmética de punto fijo. Sin embargo, a la hora de generar un código C++ con *Embeddeb* Matlab, es necesario predefinir de antemano los valores de estas propiedades. Estos valores deben ser asignados con precisión para evitar errores en la compilación con *Embeddeb*, debido a que se producen *overflows* en las operaciones de suma, producto y división. Así, ha sido necesario una fase de experimentación hasta dar con una combinación adecuada de valores asignados a cada una de estas propiedades, para evitar, por un lado, la pérdida excesiva de precisión y por otro lado los errores de *overflows*.

Finalmente, para este caso, el códido Matlab generado, con los objetos de tipo *fixed point* es el mostrado en la Figura 4.5.

```
function [indice] = N-FINDR_V3_fixed(MNFHIM,p,maxit) %#eml
assert (p <= 7);
assert (maxit <= 10);
fixsize1 = 128;
fixsize2 = 16;
[nb np] = size(MNFHIM);
T = numerictype('WordLength',fixsize1,'FractionLength',fixsize2);
% NO random initialization of the matrix with the position of endmember
indice = zeros(1,p,'int32');
for i=1:p
    indice(i) = (i*(np/p));
end
MatrixInicial = fi(zeros(p-1,p-1),1,fixsize1,fixsize2,'SumMode','KeepLSB', 'ProductMode',
'KeepLSB', 'ProductWordLength', 128, 'ProductFractionLength', 64, 'SumWordLength', 128,
'SumFractionLength', 64);
for c = 1:p
    for f = 1:p-1
        MatrixInicial(f,c) = MNFHIM(f,indice(c));
    end
end
MatrixTest = fi(zeros(p,p),1,fixsize1,fixsize2,'SumMode','KeepLSB', 'ProductMode',
'KeepLSB', 'ProductWordLength', 128, 'ProductFractionLength', 64, 'SumWordLength', 128,
'SumFractionLength', 64);
for c = 1:p
    for f = 1:p
        if f > 1
            MatrixTest(f,c) = MatrixInicial(f-1,c);
        else
            MatrixTest(f,c) = 1;
        end
    end
end
%%det
n_ = MatrixTest;
[nf_ nc_] = size(n_);
   %de_ = n_(1,1);
```

```
   de_ = fi(n_(1,1),1,fixsize1,fixsize2,'SumMode','KeepLSB', 'ProductMode',...
          'KeepLSB', 'ProductWordLength', 128, 'ProductFractionLength',...
          64, 'SumWordLength', 128, 'SumFractionLength', 64);
   for k_=1:nf_-1
      l_=k_+1;
      for i_=l_:nf_
        for j_=l_:nf_
            %n_(i_,j_) = ( n_(k_,k_)*n_(i_,j_) - n_(k_,j_)*n_(i_,k_) )/n_(k_,k_);
            n_(i_,j_) = divide(T,(n_(k_,k_)*n_(i_,j_) - n_(k_,j_)*n_(i_,k_)),n_(k_,k_));
        end
      end
      de_ = fi(de_*n_(k_+1,k_+1),1,fixsize1,fixsize2,'SumMode','KeepLSB',
'ProductMode',...
          'KeepLSB', 'ProductWordLength', 128, 'ProductFractionLength',...
          64, 'SumWordLength', 128, 'SumFractionLength', 64);
   end
 vStart = abs(fi(de_,1,fixsize1,fixsize2,'SumMode','KeepLSB', 'ProductMode',...
          'KeepLSB', 'ProductWordLength', 128, 'ProductFractionLength',...
          64, 'SumWordLength', 128, 'SumFractionLength', 64));
%volumenes auxiliares
v1 = fi(-1,1,fixsize1,fixsize2,'SumMode','KeepLSB', 'ProductMode', 'KeepLSB',
'ProductWordLength', 128, 'ProductFractionLength', 64, 'SumWordLength', 128,
'SumFractionLength', 64);
v2 = vStart;
volumeactual = vStart;
Matrix = MatrixTest;
it = uint16(0);
pixelactual = fi(zeros(p-1,1),1,fixsize1,fixsize2,'SumMode','KeepLSB', 'ProductMode',
'KeepLSB', 'ProductWordLength', 128, 'ProductFractionLength', 64, 'SumWordLength', 128,
'SumFractionLength', 64);
while and(it<maxit, v2>v1)
    it = it+1;
    for ii = 1:np
        for f = 1:p-1
            pixelactual(f) = MNFHIM(f,ii);
        end
        volume = fi(zeros(1,p),1,fixsize1,fixsize2,'SumMode','KeepLSB', 'ProductMode',
'KeepLSB', 'ProductWordLength', 128, 'ProductFractionLength', 64, 'SumWordLength', 128,
'SumFractionLength', 64);
        for k = 1:p
            MatrixTest = Matrix;
            for f=2:p
                MatrixTest(f,k) = pixelactual(f-1);
            end
n_ = MatrixTest;
[nf_ nc_] = size(n_);
   %de_ = n_(1,1);
   de_ = fi(n_(1,1),1,fixsize1,fixsize2,'SumMode','KeepLSB', 'ProductMode',...
          'KeepLSB', 'ProductWordLength', 128, 'ProductFractionLength',...
          64, 'SumWordLength', 128, 'SumFractionLength', 64);
   for k_=1:nf_-1
      l_=k_+1;
      for i_=l_:nf_
        for j_=l_:nf_
            %n_(i_,j_) = ( n_(k_,k_)*n_(i_,j_) - n_(k_,j_)*n_(i_,k_) )/n_(k_,k_);
```

```
            n_(i_,j_) = divide(T,(n_(k_,k_)*n_(i_,j_) - n_(k_,j_)*n_(i_,k_)),n_(k_,k_));
        end
      end
      de_ = fi(de_*n_(k_+1,k_+1),1,fixsize1,fixsize2,'SumMode','KeepLSB',
'ProductMode',...
          'KeepLSB', 'ProductWordLength', 128, 'ProductFractionLength',...
          64, 'SumWordLength', 128, 'SumFractionLength', 64);
   end
  volume(1,k) = abs(fi(de_,1,fixsize1,fixsize2,'SumMode','KeepLSB', 'ProductMode',...
          'KeepLSB', 'ProductWordLength', 128, 'ProductFractionLength',...
          64, 'SumWordLength', 128, 'SumFractionLength', 64));
%%det
        end
        [volume,pos]=max(volume);
        if volume > volumeactual
           volumeactual = volume;
           indice(pos) = ii;
           for f=2:p
               Matrix(f,pos) = pixelactual(f-1);
           end
       end
    end
    v1 = v2;
    v2 = volumeactual;
end
end
```

Figura 4.5. Código en punto fijo de Matlab.

Otra cuestión a tener en cuenta, es que para realizar las divisiones de objetos en punto fijo, Matlab ofrece una función llamada *divide*. Para utilizar esta función, hay que generarse un objeto de tipo *numerictype*, con sus propiedades e introducirlo como parámetro de entrada a la llamada de la función, junto con las variables.

Inicialmente, se utilizó la división genérica de Matlab, con la "/", pero se comprobó que se generaba como resultado de la división un *fixed point* con parte decimal a cero, lo cual generaba grandes pérdidas de precisión en el cálculo del determinante.

Una vez estructurado el código y los tipos de datos, se compiló del código Matlab con *Embeddeb* para generar el código en C++. El comando utilizado en la llamada se muestra a continuación:

```
emlc -s rtwcfg -d nfinder_fixed3 N-FINDR_V3_fixed.m -eg {emlcoder.egs('uint8', [5 1296]),
uint16(0),uint16(0)} -report
```

Como se puede ver, las entradas no son de tipo *fixed*, pero internamente, se convierten los datos a tipo *fixed point*. No se han introducido directamente los datos de entrada de tipo *fixed point*, porque así se gestionara de forma interna cuando se genere el código en C++. En caso de no haberlo hecho así, se hubiera tenido que generar antes el dato *fixed point* para introducirlo en la llamada a la función, desde el *main* de la herramienta de Code::Block.

Finalmente, después de las pautas dadas, se genera el código C++ con éxito.

Compilación del código fixed point usando Code::Block

Un vez se he generado el proyecto en Code::Block y se ha creado la función *main*, definimos los tipos y tamaños de las variables de entrada y salida en función de los tipos de datos creados en la cabecera de la función principal.

A continuación se muestra la cabecera de la función principal extraída del archivo N-FINDR_*fixed*.h.

```
void N-FINDR_V3_fixed(char_T eml_MNFHIM_data[955000], int32_T eml_MNFHIM_sizes[2], uint16_T
eml_p, uint16_T eml_maxit, int32_T eml_indice_data[30], int32_T eml_indice_sizes[2]);
```

En base a estos tipos de datos hacemos la llamada a la función. En la figura 4.6 se muestra el código del *main* generado:

```
int main()
{
    char_T y[955000];
    int32_T y_sizes[2];
    uint16_T nr;
    uint16_T nc;
    uint16_T nb;
    uint16_T p;
    uint16_T maxit;
    int32_T indice[30];
    int32_T indice_sizes[2];
    p=5;
    maxit=10;
    nr=100;  // 250 para la cuprite
    nc=100;  // 191 para la cuprite
    nb=p;
    y_sizes[0] = nb;
    y_sizes[1] = nc*nr;
```

```
    N-FINDR_V3_fixed_initialize();
    N-FINDR_V3_fixed(y,y_sizes,p,maxit,indice,indice_sizes);
    N-FINDR_V3_fixed_terminate();
    return 0;
}
```

Figura 4.6. Código main que llama a la función de tipo *fixed* generada en C++.

Otro aspecto a tener en cuenta una vez analizado en el entorno Code::Block el código C++ generado y los tipos de datos definidos en el archivo *rtwtypes.h*, es que el tipo de datos de nuestra entrada es *char_T* sin signo, dado que se trata de una imagen *.tiff* normalizada con valores entre 0 y 255. Sin embargo, Matlab no ha asignado a la entrada un tipo de dato sin signo, con lo cual este hecho producirá un error en la lectura de los datos de entrada. Para solucionarlo se ha de redefinir el tipo como *unsigned char*.

A continuación, se muestra la definición de tipos del archivo *rtwtypes.h*.

```
/*===========================================================================*
 * Fixed width word size data types:                                         *
 *   int8_T, int16_T, int32_T     - signed 8, 16, or 32 bit integers         *
 *   uint8_T, uint16_T, uint32_T  - unsigned 8, 16, or 32 bit integers       *
 *   real32_T, real64_T           - 32 and 64 bit floating point numbers     *
 *===========================================================================*/
typedef signed char int8_T;
typedef unsigned char uint8_T;
typedef short int16_T;
typedef unsigned short uint16_T;
typedef int int32_T;
typedef unsigned int uint32_T;
typedef float real32_T;
typedef double real64_T;
 /*===========================================================================*
 * Generic type definitions: real_T, time_T, boolean_T, int_T, uint_T,       *
 *                           ulong_T, char_T and byte_T.                     *
 *===========================================================================*/
 typedef double real_T;
typedef double time_T;
typedef unsigned char boolean_T;
typedef int int_T;
typedef unsigned uint_T;
typedef unsigned long ulong_T;
typedef unsigned char char_T;
typedef char_T byte_T;
```

Síntesis hardware del código fixed point con Catapult C

En este punto se lleva a cabo el mismo procedimiento descrito anteriormente, en el correspondiente apartado dedicado a la síntesis del código C++.

Se recuerda, que el dispositivo seleccionado es una *Altera Stratix III* con una frecuencia de 400 MHz.

En este caso no fue necesario definir funciones propias de math.h, porque no se utiliza ninguna función que no esté implementada por Catapult C.

En este caso tanto Catapult como Precisión necesitaron también una gran cantidad de horas para terminar el proceso.

4.1.3. Versión 3. Sintetizado hardware del algoritmo N-FINDR en aritmética de parte entera

En esta versión 3, se generará el código en C++ con variables definidas en parte entera. El objetivo es comprobar si compensa la utilización de aritmética en punto fijo frente a la utilización de datos enteros.

El primer inconveniente que encontramos para desarrollar el algoritmo de parte entera se produce en el cálculo del determinante. Por un lado, Matlab no tiene implementada esta función para enteros, sólo para *double*. Por otro lado, utilizando el determinante del apartado anterior, al trabajar con enteros, las operaciones de multiplicación y división utilizadas para su cálculo hacen inviable trabajar con enteros en esta parte del código, ya que se pierde tanta precisión, que los resultados obtenidos no eran comparables. Así, se recurre a la aritmética de punto fijo, como alternativa, sólo para esta parte del código, ya que no se encontró ninguna otra solución, que no fuera, la generación de un nuevo algoritmo en parte entera que calculase el determinante de una matriz de valores, con la complejidad y tiempo que eso conllevaría.

Por otro lado, al ver los resultados de las operaciones, se comprueba que aún utilizando el tipo de datos entero *int32*, este no es suficiente y se pierde mucha precisión en los cálculos, ya que hablamos en algunos casos de determinantes con valores que alcanzan un exponentes de valor cuarenta. Para solventar esta limitación, se recurre a otro tipo de dato entero de mayor rango como el *int64*, que resulta ser una clase de Matlab, y sólo se

utiliza para almacenar datos enteros. Pero está restringido realizar operaciones con este tipo de datos.

Finalmente, la estrategia a seguir fue implementarlo en parte entera desde el código generado en C++. Para ello, se definieron todos los tipos de datos a tipo *double*, en Matlab, con el objetivo de que una vez generado el código C++ con *Embedded* Matlab, se redefiniera el tipo de dato *double* como tipo de dato *long long int*. Además, la ventaja que presenta utilizar double y no otro tipo de dato, es evitar que *Embeddeb* genere un código C++ programado a la defensiva, dónde esté comprobando en todo momento el *overflow* de los datos. Al pretender trabajar con datos de mayor tamaño, este tipo de programación nos estaría limitando en todo momento al tamaño especificado desde el código Matlab.

Así finalmente, el código Matlab para esta implementación se muestra en la Figura 4.7:

```
function [indice] = N-FINDR_V3_intdouble(MNFHIM,p,maxit) %#eml
assert (p <= 7);
assert (maxit <= 10);
T = numerictype('WordLength',128,'FractionLength',24);
[nb np] = size(MNFHIM);
% No random initialization of the matrix with the position of endmember
indice = zeros(1,p,'int32');
for i=1:p
    indice(i) = i*floor(np/double(p));
end
MatrixInicial = zeros(p-1,p-1,'double');
for c = 1:p
    for f = 1:p-1
        MatrixInicial(f,c) = MNFHIM(f,indice(c));
    end
end
MatrixTest    = zeros(p,p,'double');
for c = 1:p
    for f = 1:p
        if f > 1
            MatrixTest(f,c) = MatrixInicial(f-1,c);
        else
            MatrixTest(f,c) = 1;
        end
    end
end
n_ = fi(MatrixTest,1,128,24,'SumMode','KeepLSB', 'ProductMode', 'KeepLSB',
'ProductWordLength', 128, 'ProductFractionLength', 64, 'SumWordLength', 128,
'SumFractionLength', 64);
[nf_ nc_] = size(n_);
   de_ = n_(1,1);
   for k_=1:nf_-1
```

```
      l_=k_+1;
      for i_=l_:nf_
        for j_=l_:nf_
            n_(i_,j_) = divide(T, ( n_(k_,k_)*n_(i_,j_) - n_(k_,j_)*n_(i_,k_) ),
n_(k_,k_));
        end
      end
      de_ = fi((de_*n_(k_+1,k_+1)),1,128,24,'SumMode','KeepLSB', 'ProductMode', 'KeepLSB',
'ProductWordLength', 128, 'ProductFractionLength', 64, 'SumWordLength', 128,
'SumFractionLength', 64);
   end
 vStart = abs(double(de_));
v1 = double(-1);
v2 = vStart;
volumeactual = vStart;
Matrix = MatrixTest;
it = uint16(0);
pixelactual = zeros(p-1,1,'double');
while and(it<maxit, v2>v1)
    it = it+1;
    for ii = 1:np
        %pixelactual = reshape(MNFHIM(1:p-1,ii),[p-1 1]);
        for f = 1:p-1
            pixelactual(f) = MNFHIM(f,ii);
        end
        volume = zeros(1,p,'double');
        for k = 1:p
            MatrixTest = Matrix;
            for f=2:p
                MatrixTest(f,k) = pixelactual(f-1);
            end
        n_ = fi(MatrixTest,1,128,24,'SumMode','KeepLSB', 'ProductMode', 'KeepLSB',
'ProductWordLength', 128, 'ProductFractionLength', 64, 'SumWordLength', 128,
'SumFractionLength', 64);
        [nf_ nc_] = size(n_);
           de_ = n_(1,1);
           for k_=1:nf_-1
              l_=k_+1;
              for i_=l_:nf_
                for j_=l_:nf_
                    n_(i_,j_) = divide(T, ( n_(k_,k_)*n_(i_,j_) - n_(k_,j_)*n_(i_,k_) ),
n_(k_,k_));
                end
              end
              de_ = fi((de_*n_(k_+1,k_+1)),1,128,24,'SumMode','KeepLSB', 'ProductMode',
'KeepLSB', 'ProductWordLength', 128, 'ProductFractionLength', 64, 'SumWordLength', 128,
'SumFractionLength', 64);
           end
         volume(1,k) = abs(double(de_));
        end
        [volume,pos]=max(volume);
        if volume > volumeactual
           volumeactual = volume;
           indice(pos) = ii;
           for f=2:p
```

```
                Matrix(f,pos) = pixelactual(f-1);
            end
        end
    end
    v1 = v2;
    v2 = volumeactual;
    it
end
end
```

Figura 4.7. Código Maltab generado para la Versión 3.

El comando utilizado en este caso para generar el código en C++ es el siguiente:

```
emlc -s rtwcfg -d nfinder_intdouble N-FINDR_V3_intdouble.m -eg {emlcoder.egs('uint8', [5
1296]), uint16(0),uint16(0)} -report
```

La generación del código en C++ concluye con éxito.

Compilación del código C++ en aritmética de parte entera usando Code::Block

Siguiendo el mismo procedimiento ya descrito anteriormente, se crea el proyecto en Code::Block y se define la función *main* correspondiente. Se modifican nuevamente los tipos de datos en base a la cabecera de la función principal generada en código C++ por *Embeddeb* Matlab.

A continuación, se muestra la cabecera de la función principal extraída del archivo N-*FINDR_intdouble.h*.

```
void N-FINDR_V3_intdouble(char_T eml_MNFHIM_data[955000], int32_T eml_MNFHIM_sizes[2],
uint16_T eml_p, uint16_T eml_maxit, int32_T eml_indice_data[30], int32_T
eml_indice_sizes[2]);
```

El siguiente paso, como ya se comentó en el apartado anterior, es sustituir el tipo de dato *double* que Matlab define como real_T por un entero de tipo *long long int* de 64 bits. A continuación, se muestra como quedan definidos los tipos de datos en el archivo rtwtypes.h.

```
/*=======================================================================*
 * Fixed width word size data types:                                     *
 *   int8_T, int16_T, int32_T     - signed 8, 16, or 32 bit integers     *
 *   uint8_T, uint16_T, uint32_T  - unsigned 8, 16, or 32 bit integers   *
 *   real32_T, real64_T           - 32 and 64 bit floating point numbers *
 *=======================================================================*/
```

```
typedef signed char int8_T;
typedef unsigned char uint8_T;
typedef short int16_T;
typedef unsigned short uint16_T;
typedef long long int int32_T;
typedef unsigned int uint32_T;
typedef long long real32_T;
typedef long long real64_T;
 /*===========================================================================*
 * Generic type definitions: real_T, time_T, boolean_T, int_T, uint_T,       *
 *                           ulong_T, char_T and byte_T.                     *
 *===========================================================================*/
typedef long long int real_T;
typedef long long int  time_T;
typedef unsigned char boolean_T;
typedef long long int int_T;
typedef unsigned uint_T;
typedef unsigned long ulong_T;
typedef unsigned char char_T;
typedef char_T byte_T;
```

Además, como se puede comprobar, todos los tipos de datos enteros se han definido del tipo *long long int*, para evitar *overflow* en las operaciones.

Tras realizar estas modificaciones, surgió el inconveniente de la necesidad de crear una función que convirtiera datos de tipo entero a *fixed* en lenguaje C++. Ya que en el código generado se utiliza una función que convierte datos de tipo *double* a *fixed*, y al sustituir el tipo de datos double por tipo de datos entero, la función no opera con valores correctos.

Para subsanar este error, se creó en Matlab una función que convierte un tipo de dato *fixed point* a entero y posteriormente se generó el código en C++ a través de *Embeddeb* Matlab. A continuación se muestra la función generada en código Matlab, y la función obtenida en código C++.

```
%Código Matlab
function [ z ] = convert(x)  %#eml
 xx = fi(zeros(5,5),1,128,48,'SumMode','KeepLSB', 'ProductMode', 'KeepLSB',
'ProductWordLength', 128, 'ProductFractionLength', 64, 'SumWordLength', 128,
'SumFractionLength', 64);
 xx(1,1) = x(1,1)
 xx = xx+2;
 z = int32(xx);
end

%Código C++ generado
static void m_sLong2MultiWord(int32_T eml_u, uint32_T eml_y[], int32_T eml_n)
{
```

```
  uint32_T eml_yi;
  int32_T eml_i;
  int32_T eml_loop_ub;
  eml_y[0] = (uint32_T)eml_u;
  eml_yi = eml_u < 0 ? MAX_uint32_T : 0U;
  eml_loop_ub = eml_n - 1;
  for(eml_i = 1; eml_i <= eml_loop_ub; eml_i++) {
    eml_y[eml_i] = eml_yi;
  }
}
```

Por lo tanto, se sustituye en el código C++ generado la función llamada:

```
static void m_Double2MultiWord(real_T eml_u1, uint32_T eml_y[], int32_T eml_n)
```

por la función creada para tal fin *m_sLong2MultiWord.*

Tras realizar todas estas modificaciones, aquí argumentadas, se compila el código con la herramienta *Embeddeb* y se comprueba que el resultado obtenido coincide con el esperado.

Síntesis hardware del código en aritmética de parte entera con Catapult C

El siguiente paso es realizar la síntesis del código C++ en un lenguaje de descripción hardware como VHDL o Verilog, tal y como se ha indicado para versiones anteriores en el apartado equivalente.

En este caso fue necesario definir la función *ldexp*, ya que no está implementada por Catapult. El código generado se muestra a continuación.

```
static int32_T ldexp_(int32_T param, int32_T n){
    int32_T num = 1;
    for(int i=0; i<abs(n); i++) num *= 2;
    //printf("valor de param %d valor de num %d  %d n\n",(int)param, (int)num, (int)n);
    int32_T resul;
    if(n>0){
        resul = (int32_T) param*num;;
    }else{
        resul = (int32_T) param/num;
    }
    return resul;
}
```

En este caso Catapult ha necesitado aproximadamente más de 72 horas para su finalización.

4.1.4. Versión 4. Sintetizado hardware del algoritmo N-FINDR en fixed de C++

En este caso se procede a realizar la síntesis del lenguaje C++ pero se va a implementar el código en punto fijo realizando los cambios desde el propio código C++ generado con *Embeddeb* Matlab, para la versión original. Para ello se va a utilizar una librería de C++ llamada *fixed*. A continuación se describe con más detalle el trabajo realizado.

***Generación del código C++ en fixed usando Embeddeb* Matlab**

El procedimiento utilizado para generar el código C++, en este caso, es el mismo que el utilizado en apartados anteriores Se parte del código Matlab modificado para su correcto compilado con *Embeddeb* Matlab y se genera el código C++. El comando utilizado ya ha sido descrito.

Compilación del código C++ en fixed usando Code::Block

Se realiza el mismo procedimiento que para los casos anteriores, se crea un proyecto en Code::Block, en el que se ha cargado los archivos de código fuente y se ha generado la función *main.* Los tipos de datos utilizados en la llamada a la función son los mismos que están definidos en su cabecera.

A continuación, se muestra la cabecera de la función principal extraída del archivo N-FINDR*_fixed*cpp.h.

```
extern void N-FINDR_fixedcpp(real_T eml_MNFHIM_data[8977000], int32_T eml_MNFHIM_sizes[2],
uint16_T eml_p, uint16_T eml_maxit, int32_T eml_indice_data[30], int32_T
eml_indice_sizes[2]);
```

El siguiente paso, como ya se comentó, es incluir en el proyecto las fuentes de la clase *fixed*.h y *fixed*.cpp. Una vez hecho esto se ha de redefinir el tipo *real_T*, que hasta el momento corresponde a un tipo *double*, por el tipo de dato de la clase *fixed*. Además, a lo largo del código hay que modificar los números que aparecen en coma flotante por datos de tipo de la clase *fixed*, dado que no se permite la asignación directa de los valores en coma flotante en la clase *fixed*. También, hubo que sustituir algunas estructuras propias de C+ que no permiten trabajar con clases de C++ y que genera *Embedded* Matlab en el código, como es el caso de las clausulas *union* propias de C por las clausulas *struct de C++.*

Una vez realizados los cambios descritos, la definición de los tipos en el archivo rtwtypes.h queda de la siguiente manera:

```
/*=======================================================================*
 * Fixed width word size data types:                                     *
 *   int8_T, int16_T, int32_T     - signed 8, 16, or 32 bit integers     *
 *   uint8_T, uint16_T, uint32_T  - unsigned 8, 16, or 32 bit integers   *
 *   real32_T, real64_T           - 32 and 64 bit floating point numbers *
 *=======================================================================*/
typedef signed char int8_T;
typedef unsigned char uint8_T;
typedef short int16_T;
typedef unsigned short uint16_T;
typedef int int32_T;
typedef unsigned int uint32_T;
typedef fixed real32_T;
typedef fixed real64_T;
 /*=======================================================================*
 * Generic type definitions: real_T, time_T, boolean_T, int_T, uint_T,   *
 *                           ulong_T, char_T and byte_T.                 *
 *=======================================================================*/
typedef fixed real_T;
typedef fixed time_T;
typedef unsigned char boolean_T;
typedef int int_T;
typedef unsigned uint_T;
typedef unsigned long ulong_T;
typedef unsigned char char_T;
typedef char_T byte_T;
```

Una vez realizadas las pertinentes modificaciones, se procede a compilar el código en C++ y se comprueba que los resultados obtenidos son comparables a los que cabía esperar.

Cabe mencionar que la clase *fixed* hace uso de 64 bits para el almacenamiento de la variable, por lo que se tuvo que regular el número de bits dedicados a la parte decimal para evitar el *overflow* y no perder la precisión necesaria en los cálculos del determinante. Se comprobó que esta configuración era considerablemente dependiente del tamaño de la imagen de entrada, requiriendo ajustar este valor, en función de la entrada esperada.

Síntesis hardware del código C++ en *fixed* con Catapult C

El procedimiento llevado a cabo para la síntesis del código C++ es idéntico al descrito en apartados equivalentes de versiones anteriores.

4.2. Metodología de sintetizado hardware

La metodología a desarrollar comprende de forma genérica, las pautas seguidas a lo largo del trabajo realizado, y descritas en detalle en apartados anteriores. El objetivo es establecer una metodología de trabajo, determinando los pasos a seguir para llevar a cabo la síntesis hardware de una algoritmo de tratamiento de imágenes hiperespectrales, como es el N-FINDR, y el fin último deseado sería extrapolar esta metodología a la síntesis hardware de otros algoritmos, aunque esto último, no es objeto de análisis en este trabajo fin de máster.

En la Tabla 4.1 las principales acciones llevadas a cabo en cada una de las partes del proceso, para cada una de las versiones expuestas en el apartado anterior.

Tabla 4.1. Acciones llevadas a cabo para la síntesis de las diferentes versiones.

	Código Matlab	***Embedded*** Matlab	**Código C++ en Code::Block**	**Catapult C / *Precision***
V1. Código original	Simplificación del algoritmo Inicialización de variables	Uso directiva %#eml Configuración rtwcfg Uso de assert Parámetros de entrada de tipo *doubley salida int*	Adecuación de los tipos Eliminación del math.h Implementación de la función fabs	#pragma hls_design top Generación del archivo RTL Timing report Area report
V2. Código generado con *Fixed-Point* de Matlab	Simplificación del algoritmo Inicialización de variables Creación objetos fi() estáticos. Gestionar *overflow* en las operaciones	Uso directiva %#eml Configuración rtwcfg Uso de assert Parámetros de entrada de tipo *unsigend char y salida int*	Adecuación de los tipos (char_T) Eliminación del math.h	#pragma hls_design top Generación del archivo RTL Timing report Area report
V3. Código generado con aritmética entera	Simplificación del algoritmo Inicialización de variables Tipos de datos double Gestionar *overflow* en las operaciones	Uso directiva %#eml Configuración rtwcfg Uso de assert Parámetros de entrada de tipo *unsigend char y salida long logn int*	Redefinición de los tipos (double por long long int) Eliminación del math.h Implementación de la función ldexp	#pragma hls_design top Generación del archivo RTL Timing report Area report

V4. Código generado con *Fixed* C++	Simplificación del algoritmo Inicialización de variables Tipos de datos double Gestionar *overflow* en las operaciones	Uso directiva %#eml Configuración rtwcfg Uso de assert Parámetros de entrada de tipo *unsigend char y salida long logn int*	Incluir la clase *fixed* de C++ redefiniendo el tipo *real_T* como *fixed* Eliminación del math.h	#pragma hls_design top Generación del archivo RTL Timing report Area report

El propósito de las distintas implementaciones realizadas, es abarcar de alguna manera las distintas opciones que se pueden encontrar a la hora de proceder a desarrollar la síntesis en un leguaje de descripción hardware como VHDL o Verilog de un algoritmo. En todo momento, se parte de que no se ha analizado la opción de desarrollar directamente el código en este lenguaje, si no que se trata de analizar las alternativas.

Con las implementaciones y pruebas desarrolladas, se busca encauzar una metodología, que tenga en cuenta las características del algoritmo y de los dispositivos en que se quiere sintetizar.

El uso de aritmética de punto flotante, se utiliza en gran medida en los algoritmos de procesado de imágenes, puesto que requieren de gran precisión en los datos. Sin embargo, los tiempo de procesamiento con este tipo de datos son muy altos, en comparación con la aritmética entera. En los dispositivos como FPGAS, la diferencia a la hora de realizar la operaciones con un tipo u otro de datos son muy notables, ya que no suelen disponer de unidades en coma flotante específicas, y en algunos casos se suele recurrir a módulos adicionales. Esto, ralentiza considerablemente el cómputo de estos algoritmos, además de implicar un aumento del área ocupada.

Con el fin de evaluar las prestaciones que ofrece la *toolbox* de Matlab de aritmética en punto fijo para sistemas empotrados, y compararlo con la implementación en aritmética de coma flotante y parte entera, se han implementado las tres versiones. En base a las premisas anteriormente expuestas, es lógico pensar que la opción más adecuada para reducir el coste computacional es utilizar aritmética de parte entera, el inconveniente en este caso, vendría dado por las características especiales del algoritmo y las operaciones

desarrolladas, ya que en aritmética de parte entera se pierde mucha precisión en los resultados.

En el caso de requerir más precisión, pero buscando un compromiso de coste computacional, se puede recurrir a las aritmética en punto fijo. Esta última también podría ser combinada con la aritmética de parte entera, utilizándola sólo en aquellas partes del código más sensibles a la precisión.

En última instancia, cuando las características del algoritmo requieran de excesiva precisión, se puede recurrir al cálculo en aritmética de coma flotante, teniendo siempre presente el coste computacional que esta elección conllevaría.

Otra cuestión muy importante que se debe tener en cuenta es la gestión del *overflow*, ya que en las operaciones matemáticas los datos tratados pueden ser tan grandes que el tamaño del dato permitido no sea suficiente para representarlo con la adecuada resolución. Teniendo en cuenta esto, al trabajar con enteros hay más probabilidades de que se genere *overflow*. Por ejemplo, en Matlab, aparte de presentar restricciones para operar con estos datos, sólo permite operaciones con un tamaño máximo de enteros de 32 bits. En el caso del algoritmo NFIDR, se comprobó que no era suficiente. Para el caso del uso de aritmética de punto fijo con la clase *fixed* de C++, hay una limitación de 64 bits flexible, en la que se puede configurar el tamaño asignado a la parte entera y a la parte decimal.

En este sentido, la ventaja que presenta la *toolbox* de Matlab de *fixed point*, es que el tamaño del dato puede llegar a ser de 128 bits repartidos entre la parte entera y la parte fraccionaria, lo cual, va a permitir mayor precisión en los resultados obtenidos. El inconveniente que presenta *fixed point* de Matlab, es que resulta muy ineficiente, y el código C++ que se genera suele ser muy complejo y recargado.

Con el fin de plasmar las indicaciones anteriormente expuestas, se muestra en la Figura 4.8 el diagrama de flujo propuesto, dónde se indican los pasos a seguir, en función de una serie de pautas establecidas, para llevar a cabo la síntesis en un leguaje de descripción hardware de un algoritmo en código Matlab.

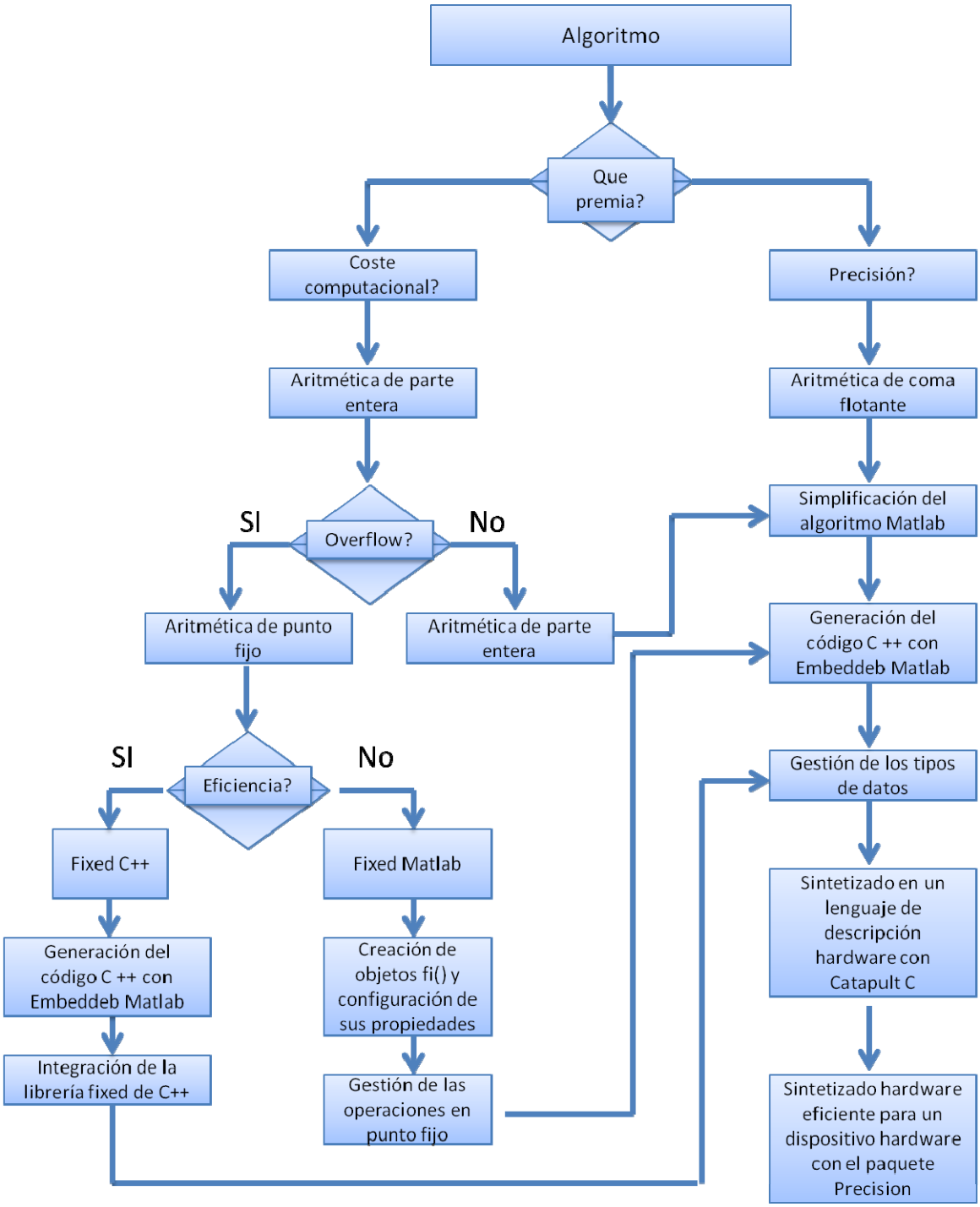


Figura 4.8. Diagrama de flujo de la metodología propuesta.

5.Resultados obtenidos

En este apartado se van a exponer los resultados obtenidos, para cada una de las cuatro implementaciones realizadas. Para ello se va a utilizar una demo, en la que se genera una imagen artificial, formada por píxeles mezclados a partir de la combinación de una serie de *Endmembers* previamente conocidos, a los que se les asignan unos valores de abundancias. A esta imagen se le añade ruido. Regulando la relación señal ruido se puede ajustar la componente de ruido gaussiano añadida a cada píxel. Finalmente, se introduce en el algoritmo de extracción de *Endmembers* NFIDR y se obtienen una serie de *Endmembers* estimados.

Para obtener información sobre la bondad de los resultados respecto a la precisión se emplean dos ángulos que se calculan a partir del método *spectral angle*. Se destaca que estos ángulos se pueden determinar ya que, como ya se dijo, se trata de imágenes artificiales donde se conoce la firma espectral pura que se debe obtener y por tanto se

puede comparar con la que se obtiene con el algoritmo. Las expresiones de los ángulos son los siguientes:

$$\theta_i \equiv (arccos \frac{\langle m_i, \hat{m}_i \rangle}{\|m_i\| \|\hat{m}_i\|})$$

Esta ecuación determina el ángulo espectral que existe entre el *Endmember* estimado $\hat{m}_i$ y el *Endmember* real m_i.

$$\beta_i \equiv (arccos \frac{\langle [S]_{i,:}, \widehat{[S]}_{i,:} \rangle}{\left\| [S]_{i,:} \right\| \left\| \widehat{[S]}_{i,:} \right\|})$$

En esta ecuación se determinan las diferencias entre las abundancias de cada píxel estimadas y las reales, siendo la fórmula para el cálculo de las abundancias $\hat{S} = \widehat{M^{\#}}[r_1, r_2, \dots, r_n]$, que representa la pseudoinversa de la matriz de *Endmembers* multiplicado por cada uno de los píxeles de la imagen.

Adicionalmente se emplea el *Spectral Information Divergence* (SID), que compara la similitud de las dos firmas espectrales.

$$SID_{m_i, \hat{m}_i} \equiv D(m_i | \hat{m}_i) + D(\hat{m}_i | m_i)$$

$$D(m_i | \hat{m}_i) \equiv \sum_{j=1}^{L} p_j \, log\left(\frac{p_j}{q_j}\right)$$

Siendo $p_j = \frac{m_{ij}}{\sum_{k=1}^{L} m_{ik}}$ y $q_j = \frac{\hat{m}_{ij}}{\sum_{k=1}^{L} \hat{m}_{ik}}$.

Es lógico deducir que cuanto más bajos son los valores de estas medidas mejores son los resultados estimados.

Estas medidas producen p valores (tantos como *Endmembers* se calculen) y por lo tanto para la representación gráfica se calcula el rms de los vectores de valores.

Par realizar la medida de estos ángulos se han generado cuatro imágenes con distintas dimensiones. Así, se han realizado las pruebas, para una imagen de dimensiones 100x100x224, a la que se ha realizado un reducción de bandas, a 5, 10 y 15 bandas, a través de una técnica de transformada en componentes principales, con el objeto de eliminar la información redundante. Realizar este tipo de reducción es habitual antes de

aplicar cualquier algoritmo de extracción de *Endmembers*. El número de *Endmembers* a obtener se ha hecho coincidir con el número de bandas.

Asimismo, también se ha aplicado las distintas metodologías para una imagen hiperespectral real, comúnmente utilizada entre la comunidad investigadora, del sensor AVIRIS, llamada Cuprite. La imagen Cuprite tiene unas dimensiones iniciales de 250x191x224, a la que también se ha hecho una reducción de bandas a 19. La característica de este imagen es que se conocen las respuestas espectrales de los materiales que existen en la imagen, de tal manera, que se pueden evaluar las respuestas espectrales de los *Endmembers* estimados con los ya originalmente conocidos.

A continuación, se procede a exponer en una serie de tablas y gráficas los resultados obtenidos, para cada una de las imágenes.

También se muestran unas figuras que representan gráficamente los simplex formados por los *Endmembers* estimados y los originales, para cada una de las imágenes. Un simplex es el análogo en n dimensiones de un triángulo.

5.1. Resultados para la imagen 100x100x5

A continuación se muestran los resultados obtenidos con la demo para la imagen de dimensiones 100x100x5, es decir, reducida a cinco bandas, y se han estimado cinco *Endmembers*.

En la Figura 5.1 se muestra la respuesta espectral de los *Endmembers* originales utilizados para generar la imagen artificial, y la respuesta espectral de los estimados. Además, de las imágenes mezcla generadas artificialmente y la representación de las abundancias. Podemos ver como los resultados estimados (en rojo) son muy similares a los originales (en azul).

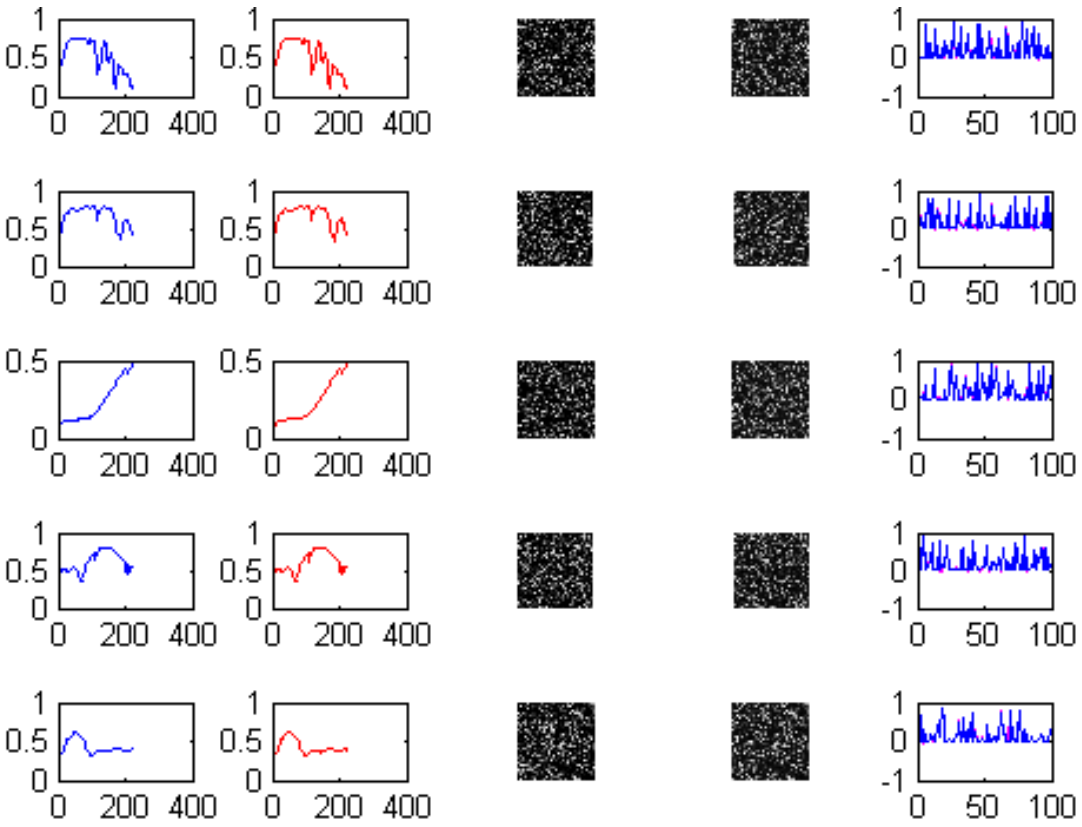


Figura 5.1. Respuestas espectrales generadas y estimadas.

Asimismo, en la Figura 5.2. se muestra el simplex formado por los cinco *Endmembers* originales y el simplex formado por los cinco *Endmembers* estimados. Se puede comprobar visualmente la similitud existente entre ambos y como se encuentra en su interior la nube formada por todos los puntos.

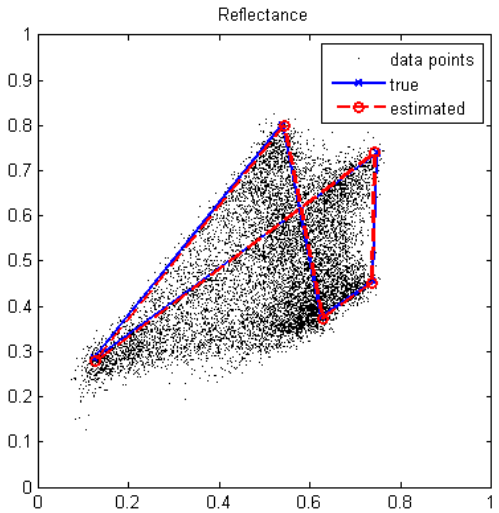


Figura 5.2. Símplex de los *Endmembers* generados y estimados.

Una vez comprobados los resultados gráficos obtenidos con la demo para la ejecución en Matlab del algoritmo N-FINDR original, procedemos a realizar diferentes medidas de tiempo y errores.

Las medidas se realizarán tanto en Matlab, como en Code::Block, de ahí que aparezcan dos resultados más, en vez de cuatro correspondientes a las versiones explicadas en el capítulo de Trabajo realizado.

En la Tabla 5.1, se indican, entre otros parámetros, los índices obtenidos. Estos índices indican la posición de los endmembers puros y se han utilizado como primera medida para comprobar el correcto funcionamiento del algoritmo en las distintas implementaciones. Los resultados no tienen por qué ser exactamente iguales en todos los casos, pero sí guardar cierta relación de similitud que demuestra que no se ha producido overflow u otro tipo de fallo que produzca valores extraños.

Asimismo, en la Tabla 5.1 también se muestran los tiempo de ejecución y las medidas de error de los dos ángulos descritos anteriormente y el parámetro SID. Dado que para las medidas de ángulos se obtienen valores de error menos significativos que se ven afectados por los valores de abundancias, sólo nos centraremos en el valor del SID:

Tabla 5.1. Resultados obtenidos para una imagen 100x100x5.

demo 1, p=5, Lines = 100 y Columns = 100					
Ejecuciones	**Índices**	**Tiempo (ms)**	**SID**	**θ (°)**	**B (°)**
Original en Matlab	7418 3071 9777 4434 3214	594	0,00241	2,29	7,29796
***Embeddeb* C en Code::Bock**	2158 3071 9777 4434 3214	42	0,00228	2,2	7,26585
***Fixed point* en Matlab**	147 3071 9777 4434 3214	15845000	0,00261	2,45	7,3023
***Fixed point* Matlab en Code::Block**	147 3071 9777 4434 3214	1492	0,00261	2,45	7,28279
***Fixed* de C++**	5580 4434 9777 3071 3214	1840	0,00227	2,18	7,2489
***Embeddeb* de parte entera**	192 3836 3908 9688 6256	1742	0,00479	3,1	7,39774

A tenor de los resultados de índices obtenidos podemos concluir que para todas las ejecuciones se obtienen valores comparables.

Para analizar en más detalle el Spectral Information Divergence (SID) se representa en la Figura 5.3 un diagrama con los resultados obtenidos para cada caso.

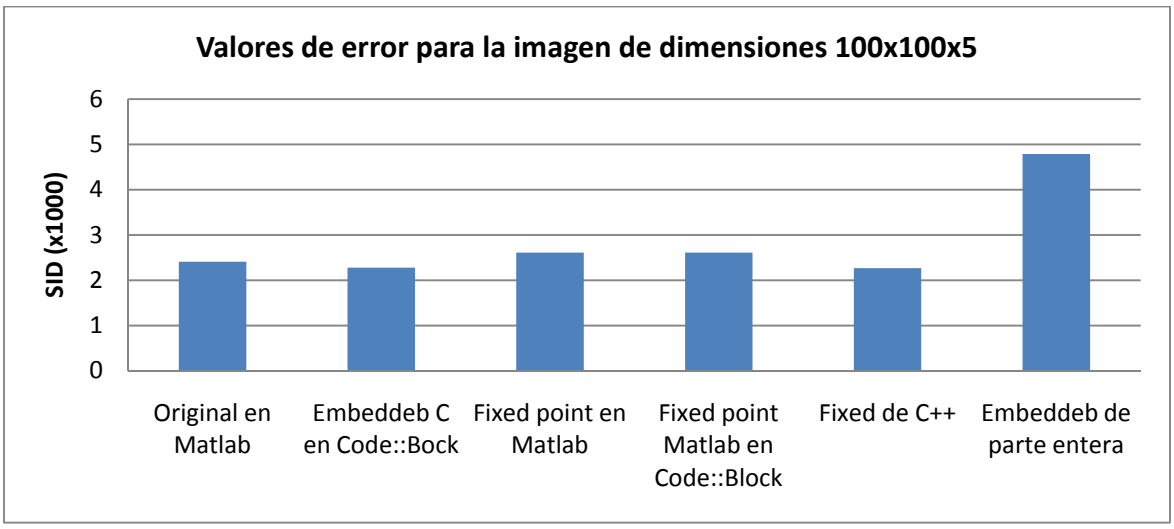


Figura 5.3. Representación gráfica del SID.

En cuanto a los valores de error, vemos que casi todos son comparables a los obtenidos con la versión original ejecutada en Matlab. Empeorando algo más en la versión de parte entera debido a la pérdida de precisión en los cálculos.

La siguiente Figura 5.4 muestra una representación gráfica de los tiempos de ejecución del algoritmo para cada una de las implementaciones. Se ha eliminado el tiempo de ejecución para la implementación de aritmética de punto fijo ejecutada en Matlab debido a que es desproporcionadamente más lenta. Esto último, también se ha realizado en las demás sucesivas gráficas que se muestran más adelante para los tiempos de ejecución.

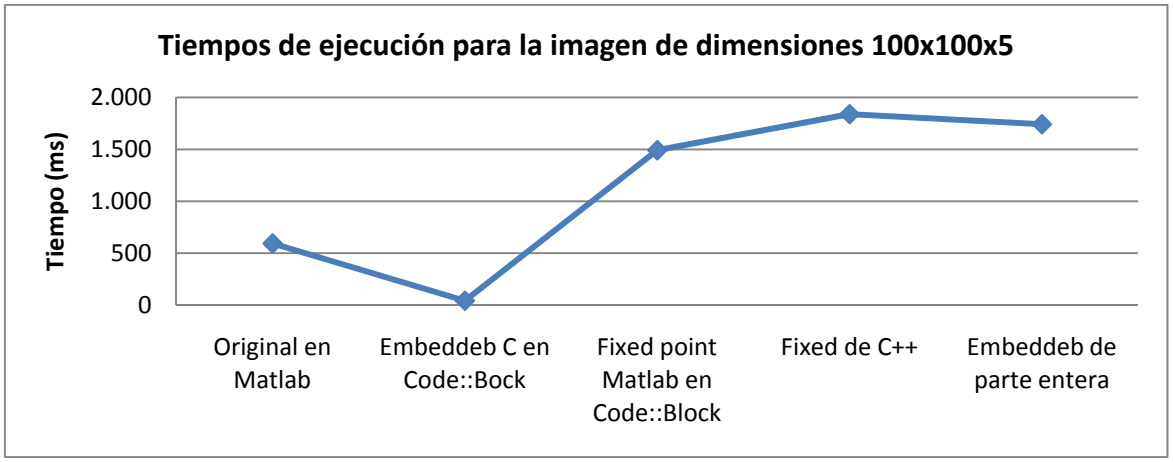


Figura 5.4. Representación gráfica de los tiempo de ejecución.

Vemos como se mejora el tiempo de ejecución del código C++ generado con *Embeddeb*, respecto al código ejecutado directamente en Matlab. Por otro lado, en las versiones de *fixed*, la que mejor tiempo de cómputo ofrece es la versión en *fixed point* de Matlab compilada en C++ con el *Embeddeb*.

5.2. Resultados para la imagen 100x100x10

A continuación, se muestran los resultados para la imagen de dimensiones 100x100x10, es decir, reducida a diez bandas, y se han estimado diez *Endmembers*.

Inicialmente se muestra en la Figura 5.5 el símplex formado por los diez Endmembers estimados y los originales, con el fin de ver que los resultados obtenidos por el algoritmo original N-FINDR ejecutado en Matlab son correctos para esta imagen.

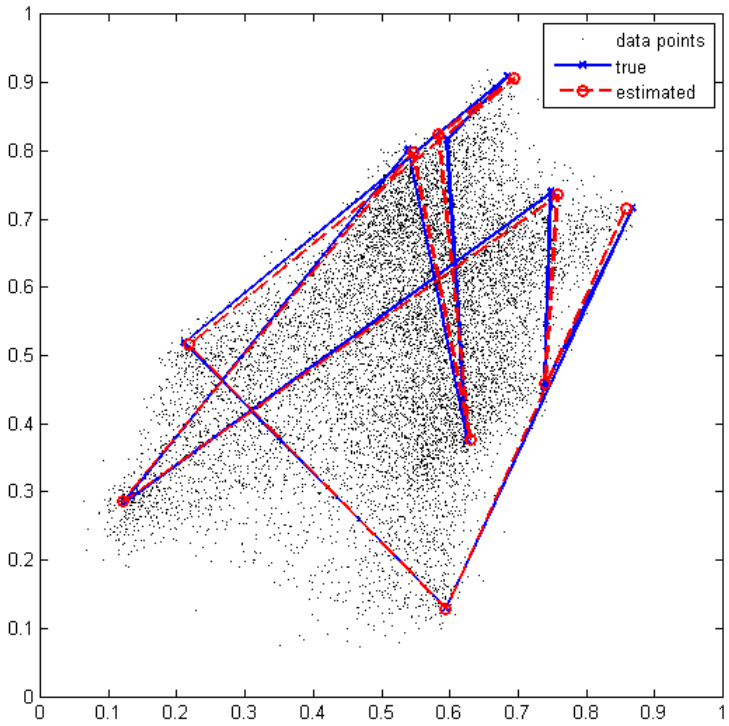


Figura 5.5. Símplex de los *Endmembers* generados y estimados.

En Tabla 5.2, se muestran los índices obtenidos, para cada uno de los métodos, el tiempo de ejecución y las medidas de error.

Tabla 5.2. Resultados obtenidos para una imagen 100x100x10.

demo 1, p=10, Lines = 100 y Columns = 100					
Ejecuciones	**Índices**	**Tiempo (ms)**	**SID**	**θ(°)**	**B(°)**
Original en Matlab	979 104 562 9499 3617 4813 5271 4153 1336 159	1880	0,0545	3,37	9,06467
***Embeddeb* C en Code::Bock**	979 104 562 9499 3617 4813 5271 4153 1336 159	474	0,0545	3,37	9,06467
***Fixed point* en Matlab**	979 4813 1450 9499 104 5271 3617 4153 1336 159	>6h	0,0545	3,37	9,06467
***Fixed point* Matlab en Code::Block**	979 4813 1450 9499 104 5271 3617 4153 1336 159	62431	0,0545	3,37	9,06467
***Fixed* de C++**	1055 104 1450 764 2242 6712 2779 3345 159 979 *tamaño parte fraccionaria 128	11108	0,0169	4,23	8, 16017
***Embeddeb* de parte entera**	4813 104 1336 9499 5271 3617 979 159 562 4153	76839	0,0545	3,37	9,06467

Vemos que los valores de índices obtenidos también guardan bastante similitud. En algunos casos no aparecen en el mismo orden, aunque esto no presenta importancia alguna. Para obtener valores lo más próximos posibles a los originales en la implementación desarrollada con la librería *fixed* de C++, se han tenido que realizar diferentes pruebas hasta obtener una combinación adecuada de bits asignados a la parte entera y decimal para evitar, en la medida de lo posible, que por un lado no se forme *overflow*, y por otro lado que no se pierda excesiva precisión en los resultados. La combinación con mejores resultados fue asignar 7 bits a la parte decimal y 57 bits a la parte entera, puesto que el tamaño máximo del dato es de 64 bits.

En la Figura 5.6 se representa en un diagrama de bloques el SID para cada una de las ejecuciones.

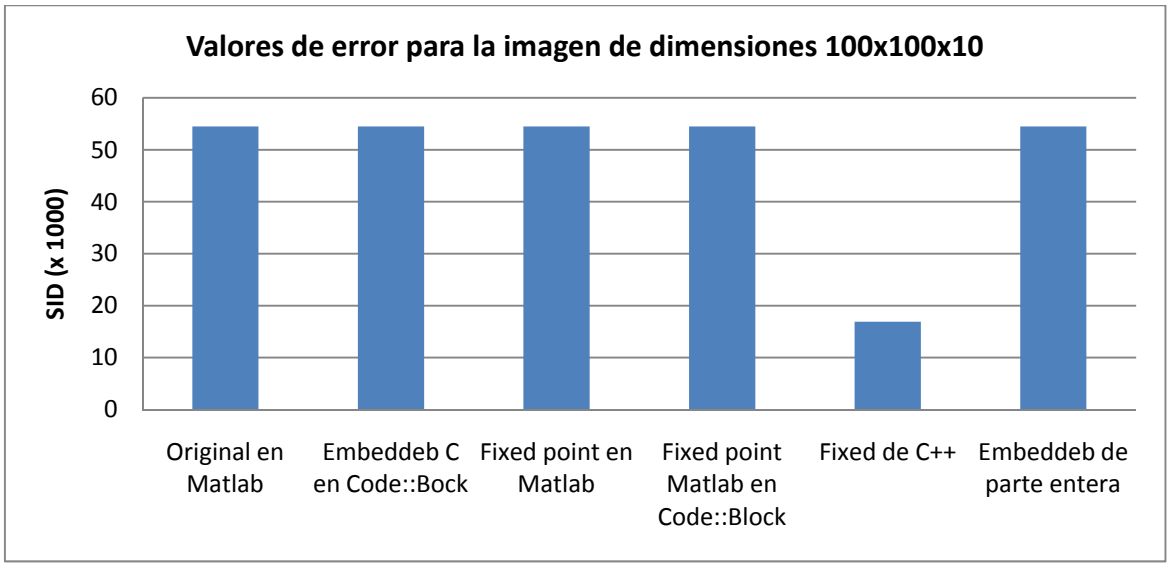


Figura 5.6. Representación gráfica del SID.

Vemos que los resultados de error son todos idénticos menos para el algoritmo desarrollado con la librería fixed de Matlab que se obtiene una mejora considerable debida a que ha aproximado mejor los cálculos de los volúmenes máximos.

La Figura 5.7. muestra una representación gráfica de los tiempos de ejecución del algoritmo en cada implementación:

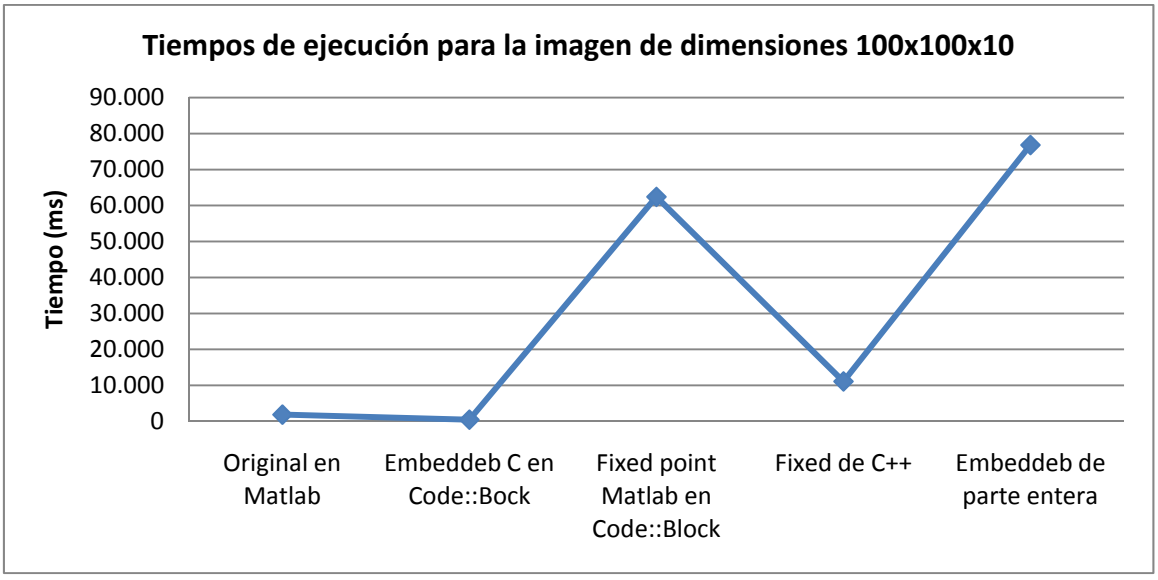


Figura 5.7. Representación gráfica de los tiempos de ejecución.

En este caso, vemos, que al igual que para el método anterior, el tiempo de ejecución del código en C++ generado con *Embeddeb* es mucho menor que ejecutándose en código Matlab. Respecto a los cálculos en *fixed*, se puede comprobar cómo con la clase *fixed* de C++ el tiempo de ejecución es mucho menor, que para el caso de *fixed point* de Matlab, que en sus dos versiones presenta un tiempo de ejecución considerablemente superior, al igual que para el de parte entera. Esto es lógico si tenemos en cuenta que en este último se ha desarrollado una parte del cálculo del determinante en aritmética de punto fijo.

5.3. Resultados para la imagen 100x100x15

A continuación se muestran los resultados para la imagen de dimensiones 100x100x15, es decir, reducida a quince bandas, y se han estimado quince *Endmembers*.

En primer lugar, igual que se ha hecho en los casos anteriores, se comprueba que el algoritmo N-FINDR original ha estimado correctamente los quince endmembers. Para ello se representa el símplex estimado, cuyos vértices se corresponden con dichos endmembers junto al original en la Figura 5.8.

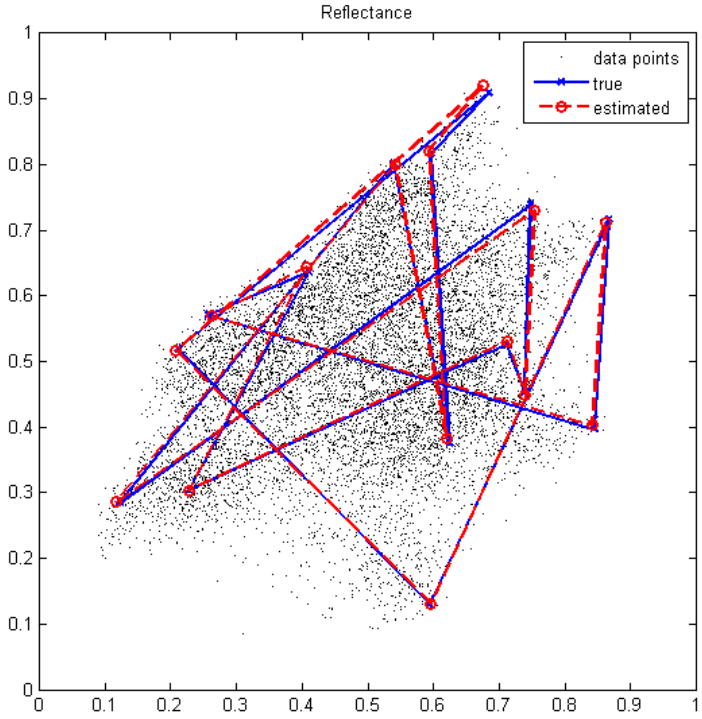


Figura 5.8. Símplex de los Endmembers generados y estimados.

Siguiendo el mismo esquema llevado hasta el momento, en la Tabla 5.3 se muestran los índices obtenidos para cada algoritmo, el tiempo de ejecución y los resultados de error obtenidos.

Tabla 5.3. Resultados obtenidos para una imagen de 100x100x15.

demo 1, p=15, Lines = 100 y Columns = 100					
Ejecuciones	**Índices**	**Tiempo (ms)**	**SID**	**θ(°)**	**B(°)**
Original en Matlab	7344 6550 2395 6312 1052 611 3047 205 6908 4536 9813 8586 8502 8180 1476	3726	0,0153	1,05	8,3
***Embeddeb* C en Code::Bock**	7344 6550 2395 6312 1052 611 3047 205 6908 4536 9813 8586 8502 8180 1476	38	0,0153	1,05	8,3
***Fixed point* en Matlab**	7344 8502 3047 1643 4411 2395 8180 6312 6908 6550 611 4536 8586 795 1476	>14 horas	0,0153	1,05	8,3
***Fixed point* Matlab en Code::Block**	7344 8502 3047 1643 4411 2395 8180 6312 6908 6550 611 4536 8586 795 1476	277282	0,0153	1,05	8,3
***Fixed* de C++**	666 1332 1998 13 856 12 4662 5328 4644 6660 7326 1349 7316 8403 4 *tamaño parte fraccionar 32	47749	0,018	1,5	8,9
***Embeddeb* de parte entera**	7344 8502 3047 1643 4411 2395 8180 6312 6908 6550 611 4536 8586 795 1476	333.449	0,0153	1,05	8,3

Podemos concluir también para este caso que los resultados de índices en todos los métodos tienen valores comparables con los obtenidos para el algoritmo original ejecutado en Matlab. Para obtener los valores lo más próximos posibles a los originales, en la implementación desarrollada con la librería *fixed* de C++, se han tenido que realizar diferentes pruebas hasta obtener una combinación adecuada de bits asignados a la parte entera y decimal, para evitar en la medida de lo posible, que por una lado no se forme *overflow*, y por otro lado, que no se pierda excesiva precisión en los resultados. La combinación con mejores resultados fue asignar 5 bits a la parte decimal y 59 bits a la parte entera, puesto que el tamaño máximo del dato es de 64 bits.

En la Figura 5.9 se representan los valores de parámetro SID para cada una de las ejecuciones.

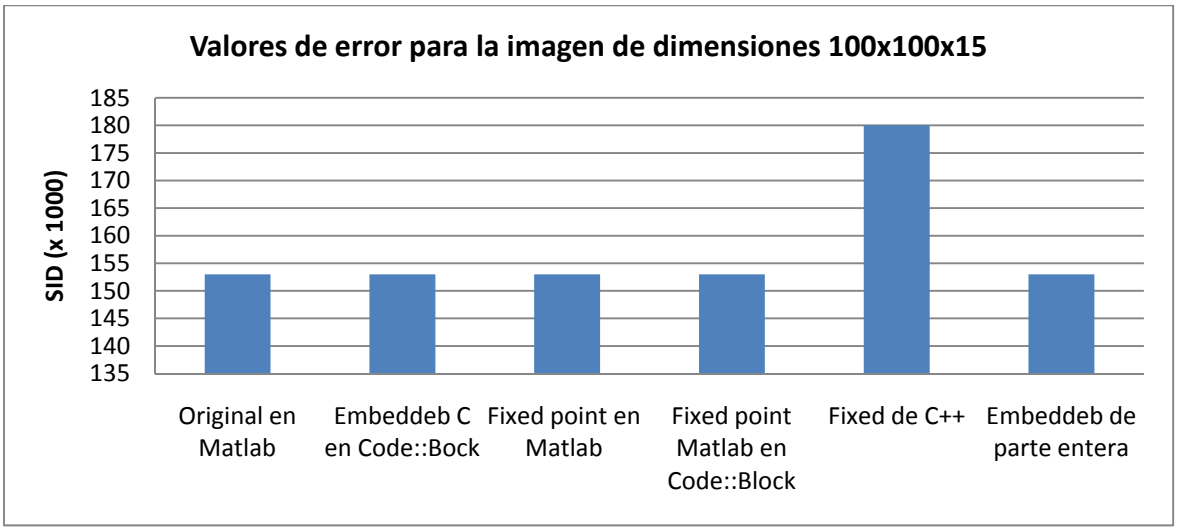


Figura 5.9. Representación gráfica del SID.

Podemos ver que los valores de error son idénticos a los obtenidos para la implementación original, menos en el caso del algoritmo en *fixed* point de C++, debido al *overflow* que se genera por la limitación del tamaño de bits.

La Figura 5.10 se muestra el diagrama de los tiempo de ejecución para esta imagen.

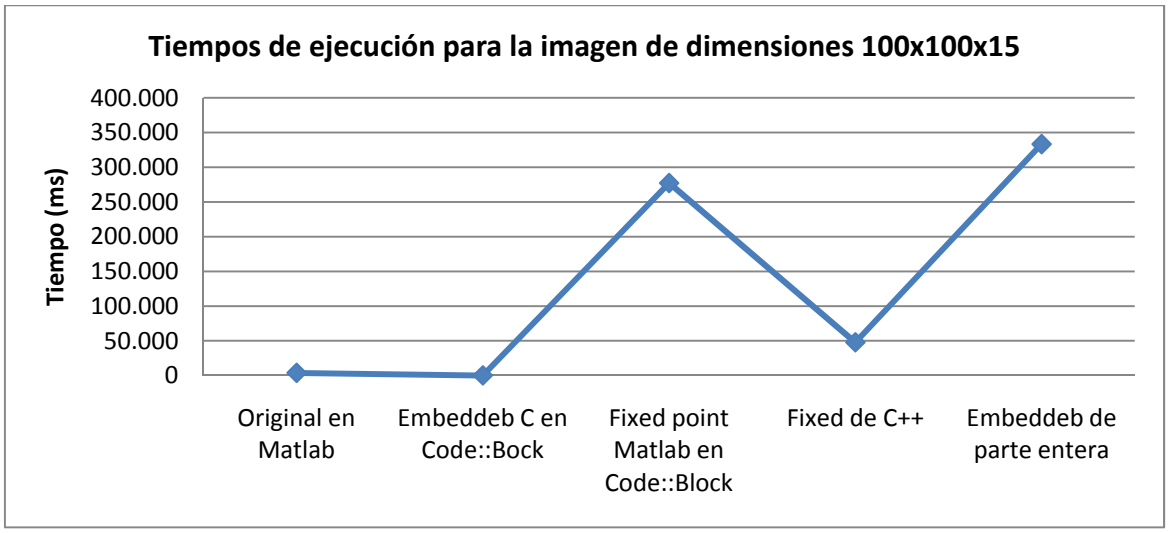


Figura 5.10. Representación gráfica de los tiempos de ejecución.

Se repite el comportamiento anterior respecto a los tiempo de ejecución. A medida que aumenta el tamaño de la imagen más clara es la diferencia en tiempo de ejecución que existe entre la *fixed* de C++, mucho más rápida, respecto a la implementación en *fixed point* de Matlab.

5.4. Resultados para la imagen Cuprite

A continuación se muestran los resultados para la imagen Cuprite de dimensiones 250x191x19, es decir, reducida a diecinueve bandas, y se han estimado diecinueve *Endmembers*.

En la Tabla 5.4 se indican los índices obtenidos, y el tiempo de ejecución. Para esta imagen, la demo no devuelve datos de error, dado que es una imagen real y no existe referencia de *Endmembers*.

Tabla 5.4. Resultados obtenidos para una imagen 250x191x19.

demo 2, p=19, Lines = 250 y Columns = 191		
Ejecuciones	**indices**	**tiempo (mg)**
Original en Matlab	2513 5026 7539 10052 12566 15079 17592 20105 22618 25132 27645 30158 32671 35184 37697 40211 42724 45237 47750	5486
***Embeddeb* C en Code::Bock**	2513 5026 7539 10052 12565 15078 17591 20104 22617 25130 27643 30156 32669 35182 37695 40208 42721 45234 47747	2721
***Fixed point* en Matlab**	2513 5026 7539 10052 12565 15078 17591 20104 22617 25130 27643 30156 32669 35182 37695 40208 42721 45234 47747	>24 horas
***Fixed point* Matlab en Code::Block**	2513 5026 7539 10052 12565 15078 17591 20104 22617 25130 27643 30156 32669 35182 37695 40208 42721 45234 47747	652264
***Fixed* de C++**	2513 5026 7539 10052 12565 15078 17591 20104 22617 25130 27643 30156 32669 35182 37695 40208 42721 45234 47747	63404
***Embeddeb* de parte entera**	2513 5026 7539 10052 12565 15078 17591 20104 22617 25130 27643 30156 32669 35182 37695 40208 42721 45234 47747	641694

Se puede comprobar cómo los valores de los índices obtenidos son prácticamente iguales en todos los casos. Lo cual es de destacar, dada la dimensionalidad de la imagen se cabía esperar que pudieran haber mayores diferencias.

En la Figura 5.11, se representa una gráfica con los tiempo de ejecución medidos para cada método.

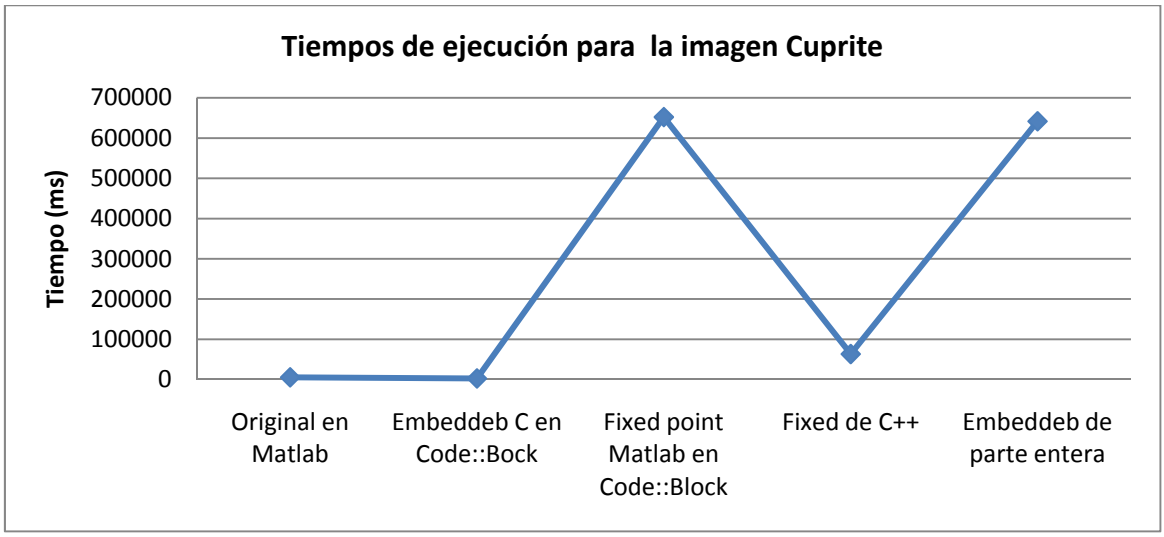


Figura 5.11. Representación gráfica de los tiempos de ejecución.

En este caso, se hace más patente la diferencia entre la ejecución del algoritmo con la clase *fixed* de C++, respecto a la ejecución *fixed point* de Matlab, lo que evidencia, en base a los resultados obtenidos, que la eficiencia del algoritmo con la librería de *fixed* de Matlab se ve reducida a medida que aumentan las dimensiones de la imagen. Por otro lado, con la ejecución de parte entera, en contra de los que se podía pensar a priori, no se han obtenidos buenos resultados en cuanto a tiempos de ejecución.

En la Figura 5.12 y 5.13 se muestran las firmas espectrales calculadas para los 19 *Endmembers* de la imagen Cuprite junto a las abundancias de cada uno de los *Endmembers* calculados.

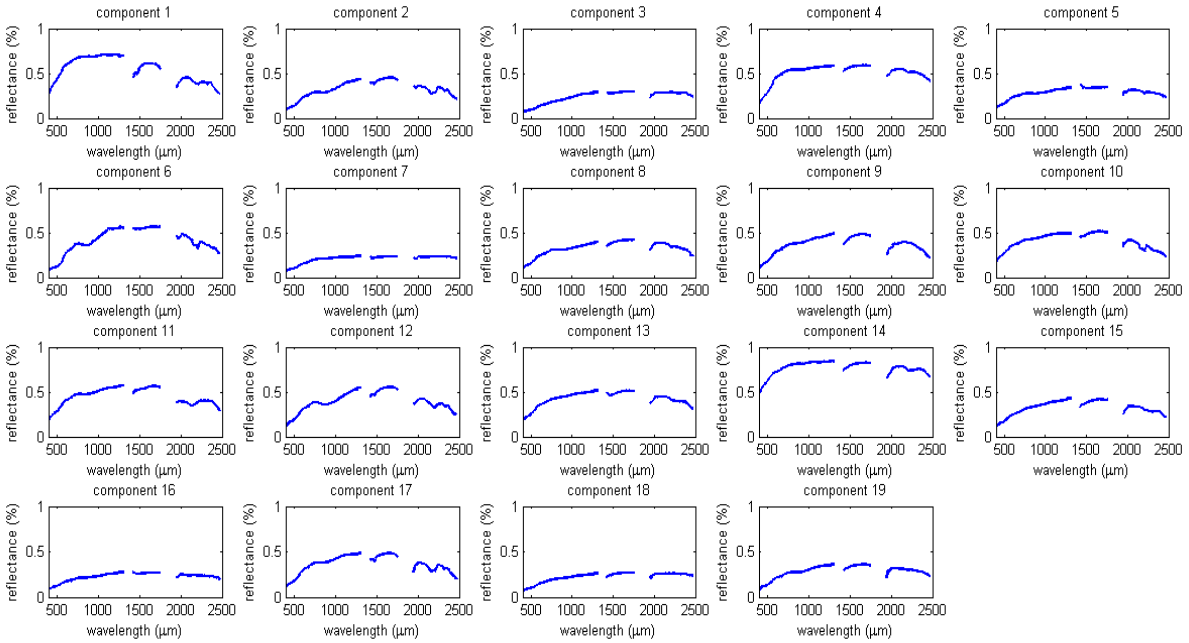


Figura 5.12. Firmas espectrales de los 19 *Endmembers* calculados de la imagen Cuprite

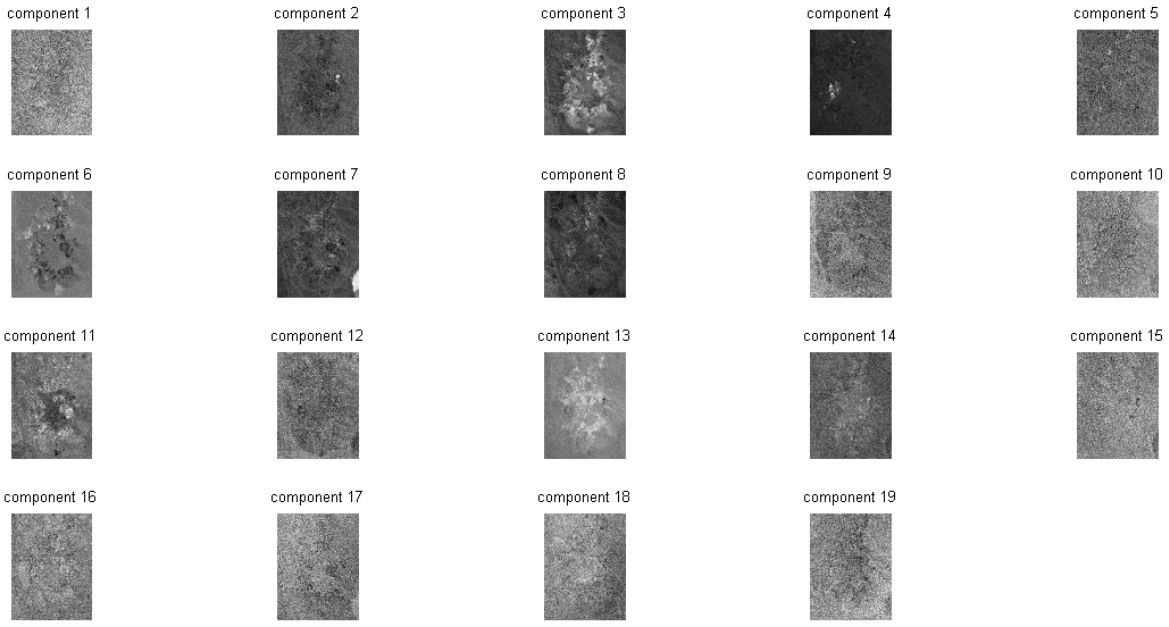


Figura 5.13. Abundancias de los 19 elementos o *Endmembers* calculados para la imagen Cuprite

5.5. Resultados de síntesis

Una vez expuestos los resultados obtenidos para la ejecución del código en sus diferentes versiones, en cuanto a errores y tiempo de ejecución, medidos en Matlab y en Code::Block, el siguiente paso es sintetizar en un lenguaje de descripción hardware los distintos códigos C++ generados. Para ello, como ya se ha adelantado en capítulos anteriores, se utiliza la herramienta Catapult C descrita en el capítulo 3.

Con el objetivo de facilitar la generación de un código con tamaños de variables manejables se han precompilado con *Embeddeb* Matlab, los algoritmos para una imagen de entrada de dimensiones 36x36x5. Se han comprobado que los resultados obtenidos con todas las metodologías guardan concordancia. A continuación se muestran en la Tabla 5.5 dichos resultados.

Tabla 5.5. Resultados obtenidos para una imagen 36x36x5.

Ejecuciones	Índices	Tiempo (ms)	SID
Original en Matlab	545 463 511 75 39	55	0,00192
***Embeddeb* C en Code::Bock**	545 511 75 463 39	15	0,00956
***Fixed point* en Matlab**	545 511 75 463 39	26849	0,00192
***Fixed point* Matlab en Code::Block**	545 511 75 463 39	189	0,00192
***Fixed* de C++**	324 511 75 463 39	49	0,00192
***Embeddeb* de parte entera**	224 511 529 811 400	231	0,0956

En la Figura 5.14 se puede ver que se repite el comportamiento en cuanto al error seguido hasta el momento para el resto de imágenes, ya que de nuevo, con el que se obtiene mayor error es con el de parte entera.

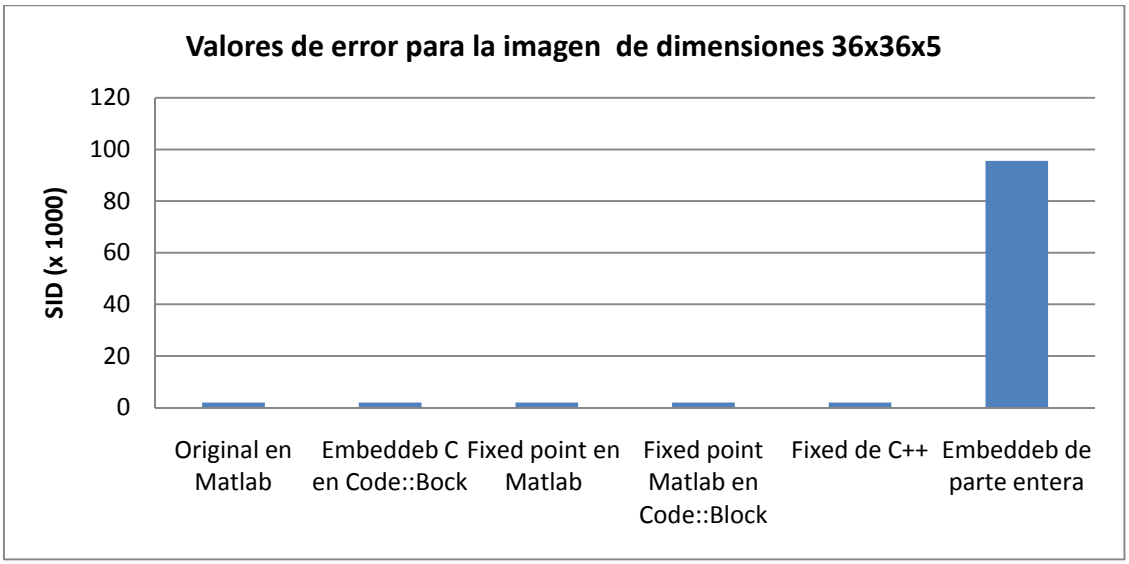


Figura 5.14. Representación gráfica del SID.

El siguiente paso ha sido introducir el codigo C++ en la herramienta Catapult C, para generar el archivo RTL. Se ha elegido como herramienta de síntesis *Precision* 2009a_up2 y como dispositivo una Altera Stratix III, hemos seleccionado todas las librerías compatibles, y como frecuencia de diseño hemos puesto 400MHz

Para la síntesis de los distintos códigos se han utilizado los mismos parámetros de diseño. El tiempo que ha necesitado Catapult C, para generar el archivo RTL, no ha sido igual en todos los casos. Destacamos por necesitar casi 72 horas, la versión de código C++ generado en parte entera, siendo el más rápido en sintetizar el código C++ de punto fijo utilizando la librería de C++ *fixed*, en aproximadamente 12 horas.

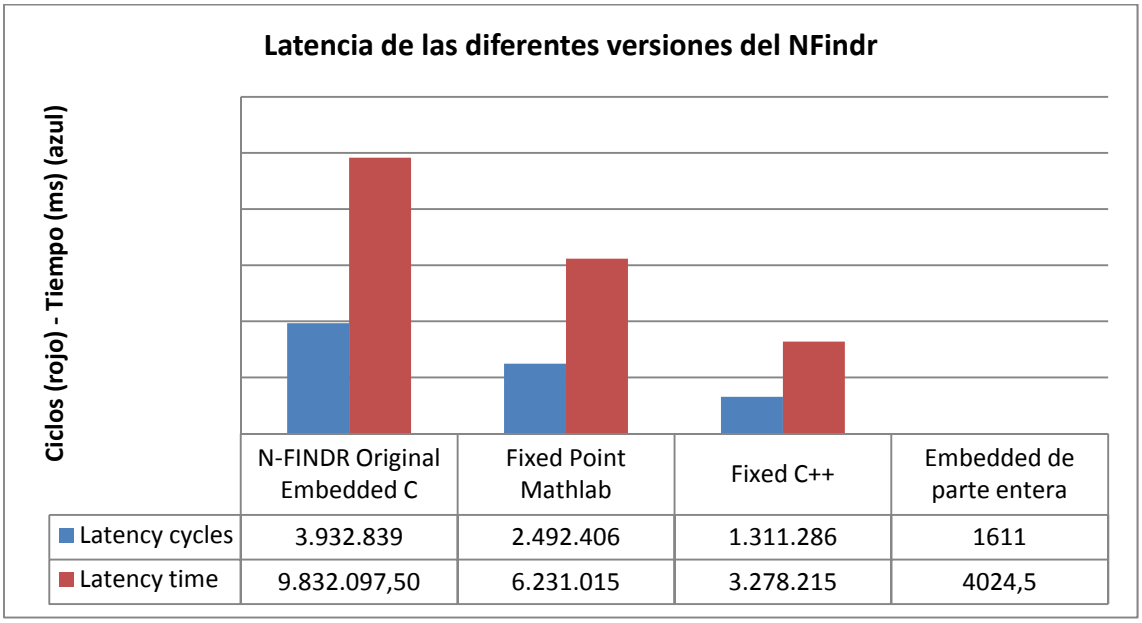
Los resultados obtenidos por Catapult para cada caso, se muestran en la Tabla 5.6:

Tabla 5.6. Resultados de la síntesis con Catapult C.

Salidas de Catapult C					
Algoritmo	Latency cycles	Latency time	Throughput cycles	Throughput time	Total area
N-FINDR Original *Embedded* C	3.932.839	9.832.097,50	3.932.849	9.832.122,5	47.188,09
***Fixed point* en Matlab**	2.492.406	6.231.015	2.492.417	6.231.042,55	102.156,1
***Fixed* de C++**	1.311.286	3.278.215	1.311.295	3.278.237,5	42.082,20
***Embedded* de parte entera**	1611	4024,5	1622	4055	148.027,61

Los datos mostrados en la Tabla 5.6 representan: la primera columna indica el número de ciclos de reloj necesarios para obtener un dato de salida, mientras que la segunda representa el tiempo necesario para obtener un dato de salida. Por otro lado, la tercera y cuarta columna hacen referencia a los resultados obtenidos por ciclos de reloj y por unidad de tiempo. Finalmente, la última columna muestra una estimación del área utilizada.

En la Figura 5.15 se muestran los resultados de latencia de las diferentes versiones del algoritmo, mientras que en la Figura 5.16 se muestran los throughput



	N-FINDR Original Embedded C	Fixed Point Mathlab	Fixed C++	Embedded de parte entera
Latency cycles	3.932.839	2.492.406	1.311.286	1611
Latency time	9.832.097,50	6.231.015	3.278.215	4024,5

Figura 5.15. Latencia en ciclos y en milisegundos para las diferentes versiones del algoritmo

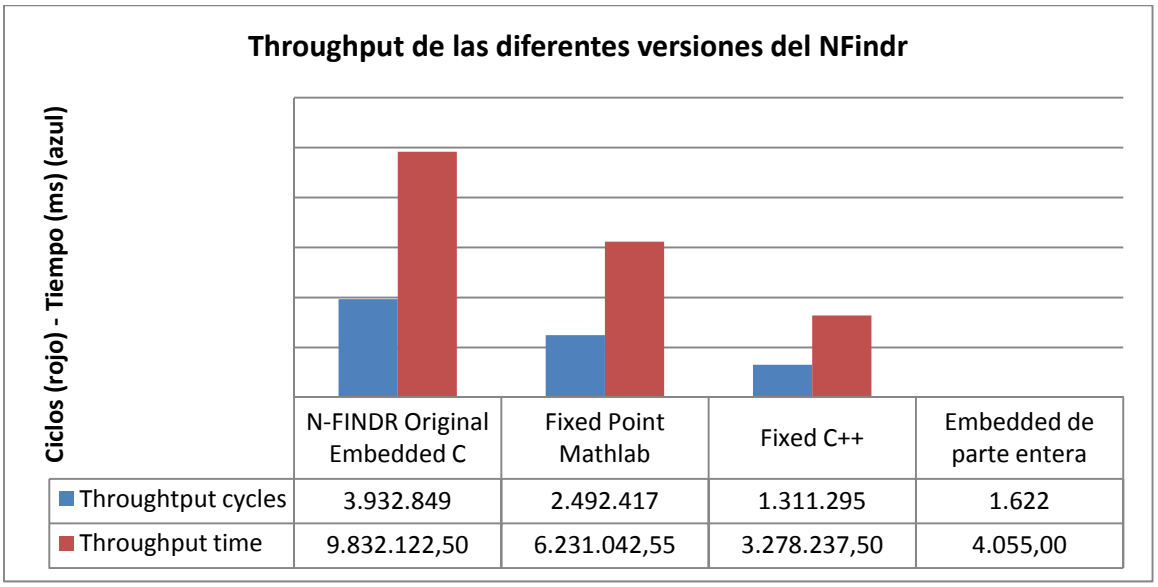


Figura 5.16. Throughtput de las diferentes versiones del algoritmo

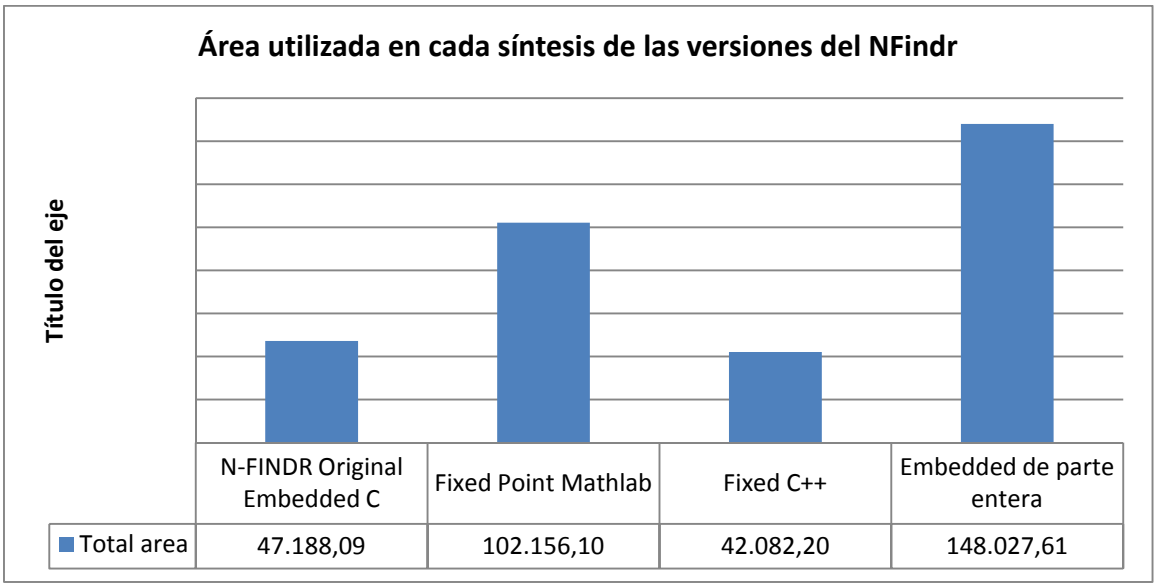


Figura 5.17. Áreas utilizadas de la FPGA para cada síntesis del algoritmo

En la Figura 5.17 se muestra el valor del área generada en la síntesis de las diferentes versiones del algoritmo N-FINDR.

Como se desprende de los datos de la tabla y las figuras, el algoritmo de aritmética entera, presenta los mejores resultados. Es el que menos ciclos de reloj necesita para obtener una salida de datos, de lo cual se deduce que es el más rápido. Podemos observar que es la versión que más área utiliza.

Los datos de Catapult suelen ser aproximados. Para obtener unos resultados más precisos de tiempo y recursos, se ha sintetizado el archivo RTL generado por Catapult con la herramienta *Precision*, integrado en la misma. Esta última nos permite obtener datos optimizados para el dispositivo con el que se ha sintetizado el código en Catapult.

Sin embargo, se han realizado dos pruebas en *Precision*, en una se ha sintetizado el archivo RTL para el mismo dispositivo utilizado en Catapult, en este caso la Altera, con resultados fallidos en dos de las síntesis. Esto nos llevó a probar la síntesis en Precisión con otro dispositivo distinto, la Xilinx - SPARTAN6, de ahí que a continuación se muestren los resultados para dos síntesis distintas en la Tabla 5.7 y 5.8.

El objetivo era obtener resultados comparables, de la síntesis de todas las metodologías, con un mismo dispositivo.

Tabla 5.7. Frecuencias y recursos para un Altera-Stratix III.

Salidas del *Precision* para una Altera - Stratix III			
Algoritmo	Frecuencias Altera (MHz)	Recursos	Totales
N-FINDR Original *Embedded* C	173,883	IOs	450
		LUTs	33763
		Registers	21718
		Memory Bits	13696
		DSP block 18-bit elems	8
***Fixed Point* en Matlab**	-sin datos	IOs	-sin datos
		LUTs	
		Registers	
		Memory Bits	
		DSP block 18-bit elems	
***Fixed* de C++**	113,675	IOs	450
		LUTs	29949
		Registers	15810
		Memory Bits	14848
		DSP block 18-bit elems	39
***Embeddeb* de parte entera**	-sin datos	IOs	-sin datos
		LUTs	
		Registers	
		Memory Bits	
		DSP block 18-bit elems	

Tabla 5.8. Frecuencias y recursos para una Xilinx - SPARTAN6.

Salidas de *Precision* para un Xilinx - SPARTAN6			
Algoritmo	Frecuencias Altera (MHz)	Recursos	Totales
N-FINDR Original *Embeddeb* C	37,164	IOs	449
		Global Buffers	1
		LUTs	29802
		CLB Slices	7451
		Dffs or Latches	21273
		Block RAMs	9
		DSP48A1s	6
***Fixed point* en Matlab**	48,783	IOs	449
		Global Buffers	1
		LUTs	72281
		CLB Slices	18071
		Dffs or Latches	46428
		Block RAMs	4
		DSP48A1s	6
***Fixed* de C++**	33,568	IOs	449
		Global Buffers	1
		LUTs	29625
		CLB Slices	7407
		Dffs or Latches	14075
		Block RAMs	10
		DSP48A1s	28
***Embeddeb* de parte entera**	32,624	IOs	449
		Global Buffers	1
		LUTs	100926
		CLB Slices	25232
		Dffs or Latches	61361
		Block RAMs	7
		DSP48A1s	27

Si comparamos los datos disponibles, para los distintos dispositivos (Altera y Xilinx), vemos como afecta que el código haya sido sintetizado en *Precision* para el mismo dispositivo Altera utilizado en Catapult, y que por lo tanto, el rango de frecuencias

obtenidas con *Precision* para este dispositivo son mucho más altas que para Xilinx. En resumen, que se ha optimizado mejor para el dispositivo de Altera.

Por otro lado, de los datos de frecuencia para la Xilinx, se podría aventurar que es más rápido el algoritmo sintetizado *fixed Point* de Matlab, sin embargo, estos datos de frecuencia, pueden llevar a engaños, ya que se puede estar refiriendo a la frecuencia que se realizan operaciones internas, que no tiene porque conllevar que se esté generando un dato de salida. Además de la frecuencia, es necesario conocer la latencia o throughput que la síntesis hardware es capaz de obtener.

En la Figura 5.18 se muestran las diferentes frecuencias máximas para las FPGAs de Xilinx de las cuales tenemos mayor información.

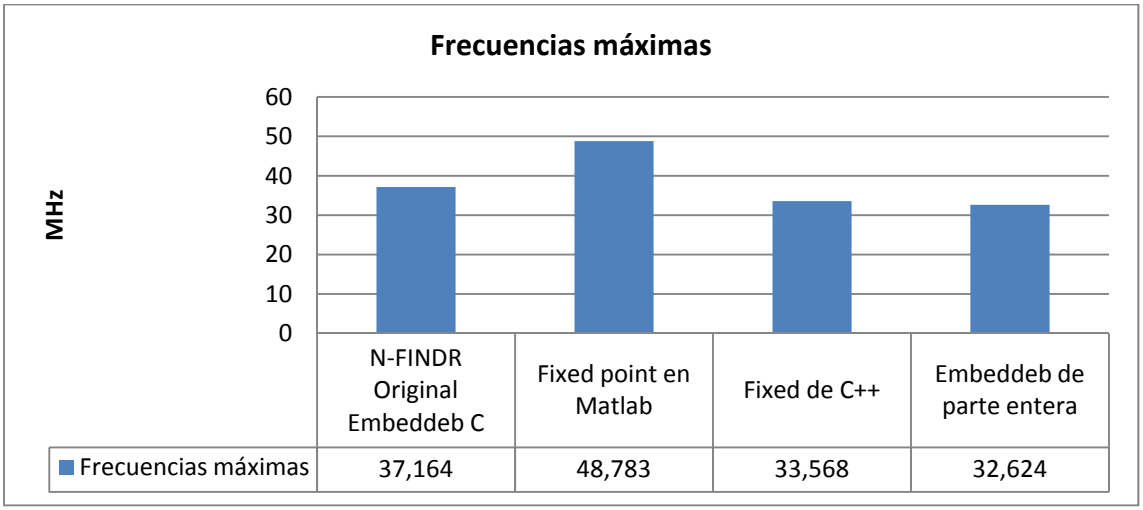


Figura 5.18. Frecuencias máximas de funcionamiento de las FPGAs de Xilinx Spartan-6

En cuanto a los recursos utilizados, sorprende la diferencia en el número de DSP utilizados para el código en *fixed* de C++ y en el de aritmética de parte entera. Esto puede indicar, que se realizan más operaciones en estas dos versiones.

Por otro lado, la versión de *fixed point* de Matlab, que está optimizada para sintetizarla en FPGAs es de las que menos recursos necesita.

Resulta extraño, bajo lo que se cabía esperar, que la síntesis para el algoritmo original, desarrollado en aritmética de coma flotante, sea de los que menos DSP necesita. A priori, se podría pensar, que necesita más DPSs para realizar las conversiones de coma flotante.

6.Conclusiones y líneas futuras

En este apartado se pretende presentar muy someramente los logros obtenidos con la realización de este trabajo fin de máster, así como las posibles líneas futuras de trabajo.

El objetivo principal de este trabajo ha sido evaluar las alternativas a la implementación directa en un lenguaje de descripción hardware como VHDL o Verilog, de un algoritmo de procesado de imágenes, en este caso se trataba de un algoritmo de extracción de *Endmembers*, llamado N-FINDR, del que se disponía definido en código Matlab.

La ventaja de desarrollar un algoritmo a un nivel más alto es la facilidad de diseñarlo, lo que ha llevado como consecuencia que haya un gran cantidad de algoritmos y funciones ya desarrollas para el tratamiento de imágenes. Así, el valor estaría en aprovechar estos condicionantes y desarrollar una metodología que permita aprovechar utilidades que ofrece Matlab, como la librería *Embeddeb*, que permite generar un código C++ que puede

a su vez ser introducido en herramientas como Catapult C, que a partir del código en C++ genera una descripción en un lenguaje orientado a la síntesis en hardware.

En función del código C++ generado y de los tipos de datos utilizados en las variables, los resultados de la síntesis hardware pueden ser mejores o peores. Así, se ha puesto todo el esfuerzo en implementar cuatro versiones de código Matlab del NFIDR.

Se ha implementado el código con aritmética de coma flotante, con aritmética de punto fijo, utilizando la librería *fixed point* de Matlab, con aritmética de parte entera y con aritmética de punto fijo utilizando una clase de C++ llamada *fixed*. Se han comparado a nivel software los resultados obtenidos para las cuatro versiones y los tiempos de ejecución en Matlab y la IDE de programación de C++ Code::Block. En cuanto a los tiempo de ejecución, es considerablemente más rápida, la implementación desarrollada con la clase *fixed point* de C++, siendo esta mejora directamente proporcional al tamaño de la imagen y es desproporcionadamente más lento, el algoritmo desarrollado en punto fijo, con la librería *fixed point* de Matlab. El código generado por *Embeddeb* para punto fijo se ha comprobado que es muy complejo y casi ilegible de interpretar. En general, se puede decir, que el código C generado por *Embeddeb* es bastante complejo, incluso para datos tipo double. Asimismo, programa muy a la defensiva, y está en todo momento comprobando los *overflow*, lo cual puede ser el causante de ralentizar la ejecución.

En cuanto a los errores de obtención de *Endmembers*, se puede decir que en general son muy similares, aunque con el que más errores se presentó, fue el desarrollado con la clase *fixed* de C++, debido a la limitación del tipo de datos a 64 bits, que era muy pequeño y se producía *overflow* en el cálculo del determinante.

Desde el punto de vista de los resultados obtenidos en la síntesis hardware de los algoritmos, los datos de frecuencia desconciertan en el sentido de que se obtienen valores más altos para el implementado en aritmética de coma flotante, cuando cabía esperar que fuera lo contrario. Lo cual puede tener su explicación, en que los datos de frecuencia obtenidos, hacen referencia a operaciones internas, que no tienen por qué estar devolviendo un dato de salida, si no referirse a posibles operaciones que se

desarrollan en la FPGA para convertir los datos en coma flotante a punto fijo. Por ello se ha considerado como dato más relevante la latencia.

Cuando se comparan los datos obtenidos de latencia, que representa los ciclos necesarios para obtener un dato de salida, se alcanzan valores considerablemente mejores para el caso del algoritmo en parte entera, comparado con el resto de las implementaciones.

A razón de los resultados obtenidos, se ha propuesto una metodología de trabajo en la que se tiene en cuenta, principalmente las características del algoritmo a implementar y los objetivos específicos para la síntesis hardware del mismo.

6.1. Líneas futuras de trabajo

Se presenta, como principal línea futura de trabajo, aplicar la metodología propuesta a otros algoritmos de tratamiento de imágenes, desarrollados en código Matlab, con el fin de evaluar su capacidad de estandarización.

Otra posible línea futura que se plantea es realizar una comparativa de las mejoras que se obtendrían optimizando la implementación del algoritmo desde el código Matlab, de cara a su conversión en código C++, ya que podría resultar tan beneficioso en los resultados como probar con diferentes tipos de datos utilizados.

Asimismo, otra posible línea futura de trabajo puede ser la prueba en diferentes dispositivos FPGAs, de la síntesis del código N-FINDR en Matlab, para identificar qué dispositivos están mejor diseñados para la síntesis hardware a partir de lenguajes de alto nivel como Matlab.

7. Bibliografía

[1] Hackwell JA, Warren DW, Bongiovi RP, Hansel SJ, Hayhurst TL, Mabry DJ, Sivjee MG, Skinner JW. *LWIR/MWIR imaging hyperspectral sensor for airborne and ground-based remote sensing.* Hackwell, JA (reprint author), AEROSP CORP, SPACE & ENVIRONM TECHNOL CTR, POB 92957, M2-251, LOS ANGELES, CA 90009 USA , 1996.

[2] A. Plaza, P. Martínez, J. Plaza, R. M. Pérez, P. L. Aguilar, M. C. Cantero. «Algoritmos de extracción de endmembers en imágenes hiperespectrales.» *Revista de Teledetección de la AET* 21 (2004).

[3] Chang CI, Wu CC, Tsai CT. «Random N-Finder (N-FINDR) Endmember Extraction Algorithms for Hyperspectral Imagery.» *IEEE TRANSACTIONS ON IMAGE PROCESSING* 20, nº 3 (2011).

[4] Plaza, Ángel. «Explicación geométrica de una medida de forma de símplices n-dimesional.» *Revista Internacional de Métodos Numéricos para Cálculo y Diseño en Ingeniería* 18, nº 4 (2001): 475-480.

[5] *Mathworks.* http://www.mathworks.com/products/matlab/ (último acceso: 16 de 06 de 2011).

[6] Pecheux F, Lallement C, Vachoux A. «VHDL-AMS and Verilog-AMS as alternative hardware description languages for efficient modeling of multidiscipline systems.» (IEEE-INST ELECTRICAL ELECTRONICS ENGINEERS INC, 445 HOES LANE, PISCATAWAY, NJ 08855 USA) 2005.

[7] *From MATLAB to Embedded C.* http://www.mathworks.com/company/newsletters/news_notes/2009/matlab-embedded-c.html (último acceso: 16 de 06 de 2011).

[8] *Fixed-Point Toolbox .* http://www.mathworks.com/products/fixed/ (último acceso: 16 de 06 de 2011).

[9] *Libfixmath.* http://code.google.com/p/libfixmath/ (último acceso: 16 de 06 de 2011).

[10] *Electronic System Level Design.* http://www.mentor.com/esl/catapult/overview (último acceso: 16 de 06 de 2011).

[11] F. Chaudhry, C Wu, W. Liu, C. Chang, A. Plaza. «Pirex purity index-dased algorithms for endmember extraction from hyperspectral imagery.» (Transworld Research Network) pp.30-62 (2006).

[12] M. Zortea, A. Plaza. «A quantitative and comparative analysis of different implementations of N-FINDR: A fast endmember extraction algorithm.» (IEEE Geoscience and Remote Sensing Letters), nº pp. 787-791 (2009).

[13] J. Nascimento, J. Bioucas. «Vertex component analysis: a fast algorithm to unmix hyperspectral data.» (IEEE Transactions on Geoscience and Remote Sensing), nº pp. 898-910 (2005).

[14] Winter, Michael E. «N-FINDR: an algorithm for fast autonomous spectral end-member determination in hyperspectral data .» (SPIE Digital Library) 3753, nº 266 (1999).

[15] Chein-I Chang, Fellow, IEEE, Chao-Cheng Wu, Member, IEEE, and Ching-Tsorng Tsai. «Random N-Finder (N-FINDR) Endmember Extraction Algorithms for Hyperspectral Imagery.» (IEEE Transactions on imagepProcessing) 30, nº 3 (2001).

[16] *Embedded Matlab Function Library Reference.* http://www.kxcad.net/cae_MATLAB/toolbox/eml/ug/bq1h2z7-9.html (último acceso: 2011 de 6 de 1).

[17] Erdelsky, Philip J. *A Fixed-Point Arithmetic Package.* http://www.efgh.com/software/fixed.htm (último acceso: 1 de 06 de 2011).

[18] *Catapult C Synthesis.* 1 de 06 de 2011. http://www.mentor.com/esl/catapult/overview.

[19] Doxaran D, Froidefond JM, Lavender S, Castaing P. *Spectral signature of highly turbid waters - Application with SPOT data to quantify suspended particulate matter concentrations.* Talence, France : ELSEVIER SCIENCE INC, 360 PARK AVE SOUTH, NEW YORK, NY 10010-1710 USA , 2002.

[20] Adams JB, Sabo de, Kapos V, Almeida R, Robert Da, Smith Mo, Gillespir Ar. Classification of Multispectral Images based on Fractions od Endmembers- Aplication to Land Cover Change in the Brazilian Amazon. Cambridge, England.+

[21] Nascimento JMP, Dias JMB. *Vertex component analysis: A fast algorithm to unmix hyperspectral data.* Lisbon, Portugal : IEEE-INST ELECTRICAL ELECTRONICS ENGINEERS INC, 445 HOES LANE, PISCATAWAY, NJ 08855 USA , 2005.

[22] Plaza, A., Chang, C. «Impact of Initialization on Design of Endmember Extraction Algorithms.» (IEEE Transactions on Geoscience and Remote Sensing), nº 3397-3407 (2006).

[23] Pablo Hostrans Andaluz, Roberto Sarmiento Rodríguez, Sebastián López Suárez. *Implementación hardware del algoritmo Vertex Component Analysis (VCA) para el procesamiento de imágenes hiperespectrales.* Las Palmas de G.C., 2011.