ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA Y SISTEMAS DE TELECOMUNICACIÓN

PROYECTO FIN DE GRADO

TÍTULO: Generación de una librería RVC – CAL para la etapa de determinación de *endmembers* en el proceso de análisis de imágenes hiperespectrales

AUTOR: Daniel Madroñal Quintín

TITULACIÓN: Grado en Ingeniería Electrónica de Comunicaciones

TUTOR (o Director en su caso): Eduardo Juárez Martínez

DEPARTAMENTO: Departamento de Ingeniería Telemática y Electrónica

VºBº

Miembros del Tribunal Calificador:

PRESIDENTE: Juana María Gutiérrez Arriola

VOCAL: César Sanz Álvaro

SECRETARIO: Eduardo Juárez Martínez

Fecha de lectura:

Calificación:

El Secretario,

Agradecimientos

Tras cuatro años de duro esfuerzo, al fin puedo decir que me siento orgulloso de mi mismo por haber conseguido una de las principales metas que me había propuesto en mi vida. Pero, por supuesto, esto no habría sido posible sin ciertas personas que me han estado apoyando y han confiado en mí incondicionalmente:

- En primer lugar, agradecer a mi familia por estar a mi lado siempre y aguantar mis malos humores pero, sobre todo, por sacarme una sonrisa nada más llegar a casa con saludos que ya se han grabado en mi mente como "Pero, ¿y éste quién es?" o "Ah, ¿Qué sigues viviendo en esta casa?".
- A continuación, quería agradecer a Raquel Lazcano, alguien que bien podría estar en el primer grupo ya que, al fin y al cabo, se ha convertido en una hermana para mí. Sin su apoyo, su dedicación, la mutua colaboración, sus broncas y, sobre todo, sin nuestras bromas, todo lo que he conseguido a día de hoy habría sido total y absolutamente imposible.
- Por supuesto, no puedo olvidarme de mis amigos, agradecerles por estar siempre ahí cuando les necesito y por devolverme a la realidad y hacerme desconectar como nadie más sabe hacerlo. Sin lugar a dudas, quiero destacar a dos personas que para mí, ya no son amigos, son familia. El primero de ellos es Agustín, ese muchacho que jamás me ha fallado y que, a pesar de mis múltiples negativas a la hora de quedar – todo por seguir haciendo cosas – nunca me puso una mala cara y nunca dejó de insistir. Y para finalizar, una persona que se ha convertido en alguien total y absolutamente imprescindible en mi vida, su apoyo incondicional y su compañía en todos y cada uno de los momentos más duros ha sido uno de los pilares fundamentales para que no me derrumbase y lo dejase todo a un lado, gracias por estar ahí siempre María
- Por último, quiero agradecer a Eduardo Juárez, aparte de darme la oportunidad de realizar este proyecto siendo el tutor del mismo, me ha dado a conocer un aspecto totalmente desconocido hasta entonces, la investigación de los campos de las telecomunicaciones y la medicina unidos en uno solo. Dentro del grupo de investigación, agradecer la colaboración de Alejo Iván Arias, no solo por su aportación al proyecto sino por los ratos que hemos pasado desde la apertura hasta el cierre del laboratorio día tras día. Agradecer a la UPM, en concreto, a la ETSIST la oportunidad de aprender con los mejores y que nos soporten también los mejores maestros de laboratorio que nos podríamos haber encontrado. Juan Carlos y César, un placer haber trabajado con vosotros.

A todos los aquí nombrados y a los que posiblemente se me haya olvidado mencionar, gracias por estar ahí. Espero que en la nueva etapa que comienzo a partir de ahora todo sea, como mínimo, como ha sido hasta ahora.

.

Resumen

El análisis de imágenes hiperespectrales permite obtener información con una gran resolución espectral: cientos de bandas repartidas desde el espectro infrarrojo hasta el ultravioleta. El uso de dichas imágenes está teniendo un gran impacto en el campo de la medicina y, en concreto, destaca su utilización en la detección de distintos tipos de cáncer. Dentro de este campo, uno de los principales problemas que existen actualmente es el análisis de dichas imágenes en tiempo real ya que, debido al gran volumen de datos que componen estas imágenes, la capacidad de cómputo requerida es muy elevada. Una de las principales líneas de investigación acerca de la reducción de dicho tiempo de procesado se basa en la idea de repartir su análisis en diversos núcleos trabajando en paralelo.

En relación a esta línea de investigación, en el presente trabajo se desarrolla una librería para el lenguaje RVC – CAL – lenguaje que está especialmente pensado para aplicaciones multimedia y que permite realizar la paralelización de una manera intuitiva – donde se recogen las funciones necesarias para implementar dos de las cuatro fases propias del procesado espectral: *reducción dimensional* y *extracción de* endmembers. Cabe mencionar que este trabajo se complementa con el realizado por Raquel Lazcano en su Proyecto Fin de Grado, donde se desarrollan las funciones necesarias para completar las otras dos fases necesarias en la cadena de desmezclado.

En concreto, este trabajo se encuentra dividido en varias partes. La primera de ellas expone razonadamente los motivos que han llevado a comenzar este Proyecto Fin de Grado y los objetivos que se pretenden conseguir con él. Tras esto, se hace un amplio estudio del estado del arte actual y, en él, se explican tanto las imágenes hiperespectrales como los medios y las plataformas que servirán para realizar la división en núcleos y detectar las distintas problemáticas con las que nos podamos encontrar al realizar dicha división. Una vez expuesta la base teórica, nos centraremos en la explicación del método seguido para componer la cadena de desmezclado y generar la librería; un punto importante en este apartado es la utilización de librerías especializadas en operaciones matriciales complejas, implementadas en C++. Tras explicar el método utilizado, se exponen los resultados obtenidos primero por etapas y, posteriormente, con la cadena de procesado completa, implementada en uno o varios núcleos. Por último, se aportan una serie de conclusiones obtenidas tras analizar los distintos algoritmos en cuanto a bondad de resultados, tiempos de procesado y consumo de recursos y se proponen una serie de posibles líneas de actuación futuras relacionadas con dichos resultados.

Abstract

Hyperspectral imaging allows us to collect high resolution spectral information: hundred of bands covering from infrared to ultraviolet spectrum. These images have had strong repercussions in the medical field; in particular, we must highlight its use in cancer detection. In this field, the main problem we have to deal with is the real time analysis, because these images have a great data volume and they require a high computational power. One of the main research lines that deals with this problem is related with the analysis of these images using several cores working at the same time.

According to this investigation line, this document describes the development of a RVC – CAL library – this language has been widely used for working with multimedia applications and allows an optimized system parallelization –, which joins all the functions needed to implement two of the four stages of the hyperspectral imaging processing chain: *dimensionality reduction* and *endmember extraction*. This research is complemented with the research conducted by Raquel Lazcano in her Diploma Project, where she studies the other two stages of the processing chain.

The document is divided in several chapters. The first of them introduces the motivation of the Diploma Project and the main objectives to achieve. After that, we study the state of the art of some technologies related with this work, like hyperspectral images and the software and hardware that we will use to parallelize the system and to analyze its performance. Once we have exposed the theoretical bases, we will explain the followed methodology to compose the processing chain and to generate the library; one of the most important issues in this chapter is the use of some C++ libraries specialized in complex matrix operations. At this point, we will expose the results obtained in the individual stage analysis and then, the results of the full processing chain implemented in one or several cores. Finally, we will extract some conclusions related with algorithm behavior, time processing and system performance. In the same way, we propose some future research lines according to the results obtained in this document.

Índice de contenidos

Índice de figuras

Índice de tablas

Lista de acrónimos

- **AMEE:** *Automatic Morphological Endmember Extraction*
- **CITSEM:** *Centro de Investigación en Tecnologías de Software y Sistemas Multimedia para la Sostenibilidad*
- **FET:** *Future & Emerging Tecnologies*
- **GDEM:** *Grupo de Diseño Electrónico y Microelectrónico*
- **HAnDy:** *Hyperspectral Algorithms Development*
- **HELICoiD:** *HypErspectraL Imaging Cancer Detection*
- **HSI:** *HyperSpectral Imaging*
- **HYSIME:** *Hyperspectral Signal Subspace Identification by Minimum Error*
- **IEA:** *Iterative Error Analysis*
- **ITK:** *Insight ToolKit* **(***Insight Segmentation and Registration Toolkit)*
- **LSU:** *Linear Spectral Unmixing*
- **MNF:** *Minimum Noise Fraction*
- **N-FINDR:** *N – finder algorithm*
- **ORCC:** *Open RVC – CAL Compiler*
- **OSP:** *Orthogonal Subspace Projection*
- **OTB:** *ORFEO ToolBox*
- **PAPI:** *Performance Application Programming Interface*
- **PCA:** *Principal Component Analysis*
- **RVC-CAL:** *Reconfigurable Video Coding – CAL Actor Language*
- **SPP:** *Spatial Pre - Processing*
- **SSEE:** *Spatial – Spectral Endmember Extraction*
- **SVD:** *Singular Values Decomposition*
- **UPM:** *Universidad Politécnica de Madrid*
- **VCA:** *Vertex Component Analysis*
- **VGL:** *Vision Geometric Library*
- **VNL:** *Vision Numeric Library*
- **VUL:** *Vision Utilities Library*
- **VXL:** *Vision something Library*

CAPÍTULO 1. INTRODUCCIÓN

1.1. Motivaciones

El trabajo desarrollado en este Proyecto Fin de Grado se centra en el desarrollo de un nuevo método de implementación de cadenas de procesado de imágenes hiperespectrales. Este proyecto se enmarca dentro de una de las líneas de investigación del Grupo de Diseño Electrónico y Microelectrónico (GDEM) perteneciente al Centro de Investigación en Tecnologías de Software y Sistemas Multimedia para la Sostenibilidad (CITSEM) de la Universidad Politécnica de Madrid (UPM).

Esta línea de investigación se inscribe dentro del proyecto europeo HELICoiD[1] (de sus siglas en inglés, *HypErspectraL Imaging Cancer Detection*). Dicho proyecto comenzó en enero de 2014 y está enmarcado en el Séptimo Programa Marco de la Unión Europea, concretamente en el programa de Tecnologías Emergentes y Futuras (FET – *Future & Emerging Technologies)*. En este proyecto de colaboración participan miembros de diversos países, tales como Francia, Reino Unido, Holanda y España y, a su vez, reúne dos hospitales, tres empresas y cuatro universidades, entre las que se incluye la UPM.

El objetivo del proyecto HELICoiD es estudiar la viabilidad de utilizar imágenes hiperespectrales para realizar una diferenciación entre tejido sano y tejido tumoroso en tiempo real durante una cirugía. El motivo por el que se utilizan dichas imágenes es la capacidad de extraer la firma espectral de cada tipo de tejido y, de esta manera, realizar la diferenciación.

En concreto, dentro de HELICoiD, el grupo GDEM lidera el paquete de trabajo relacionado con la selección de la plataforma hardware que se utilizará y la forma de implementación de los algoritmos de la cadena de procesado; además, ayudará en el desarrollo y análisis de funcionamiento de dichos algoritmos. En este análisis, el principal trabajo del grupo será la detección de los posibles cuellos de botella que ralenticen el sistema y, por lo tanto, no permitan la consecución del análisis en tiempo real. En este campo el GDEM tiene una amplia experiencia, debido a su línea de investigación relacionada con la optimización del consumo de energía mediante la monitorización de sistemas.

Por tanto, el presente Proyecto Fin de Grado se centrará en el desarrollo de un método que facilite la elaboración de distintos algoritmos pertenecientes a la cadena de procesado de imágenes hiperespectrales. En concreto, se pretende facilitar la monitorización de los distintos algoritmos y, a su vez, reducir la dificultad que supone la paralelización de los algoritmos desarrollados – ya que, presumiblemente, la plataforma hardware seleccionada será una plataforma multinúcleo.

En este sentido, cabe destacar que, para facilitar la realización de dicho paralelismo, se utilizará un lenguaje de flujo de datos denominado RVC – CAL. La principal característica de un lenguaje de este tipo es su estructura: bloques funcionales que intercambian datos entre sí.

A su vez, para realizar la monitorización y la consecuente localización de cuellos de botella, se utilizará la herramienta de análisis hardware denominada PAPI, cuya integración ha sido optimizada por Alejo Iván Arias dentro del propio GDEM.

[1] http://helicoid.eu/

1.2. Objetivos

El principal objetivo del presente trabajo es el desarrollo de un nuevo método que facilite la elaboración y posterior implementación de algoritmos propios de una cadena de procesado de imágenes hiperespectrales. Para ello, se han estudiado los distintos algoritmos que componen cada una de las fases de una cadena de procesado y, posteriormente, se ha optado por elaborar una librería que incluya las funciones necesarias para implementar algoritmos de procesado espectral en RVC – CAL. Para alcanzar este objetivo global, se han fijado los siguientes objetivos específicos:

- ✓ Estudiar en detalle las imágenes hiperespectrales, su proceso de análisis y los métodos utilizados actualmente para reducir el tiempo de procesado y aproximarse, en la medida de lo posible, al tiempo real.

- ✓ Implementar un algoritmo de cada etapa de procesado con el fin de ampliar la librería y, por consiguiente, demostrar la reducción de la dificultad y el tiempo de desarrollo que conlleva la implementación de cualquiera de los algoritmos.

- ✓ Desarrollar interfaces de comunicación entre etapas y de entrada y salida del sistema, con el fin de ser capaces de realizar paralelismos y establecer la comunicación entre procesadores.

- ✓ Importar la librería desarrollada al entorno de programación RVC – CAL. Esto nos permite establecer de manera óptima el paralelismo existente en la cadena de procesamiento y, a su vez, nos aporta la facilidad de programar en un lenguaje tradicional, como C.

- ✓ Realizar un estudio exhaustivo de la bondad de los algoritmos desarrollados en cuanto a lo que exactitud de resultados y tiempo de ejecución se refiere. Este estudio sirvió también para verificar el correcto funcionamiento de la librería desarrollada.

- ✓ Analizar el consumo de recursos de los distintos algoritmos desarrollados y de las distintas distribuciones en procesadores planteadas mediante la utilización de la herramienta PAPI. Este análisis nos permite localizar los posibles cuellos de botella existentes en la cadena de procesado y mejorar la distribución en procesadores realizada.

Cabe destacar que este Proyecto Fin de Grado se ha desarrollado en paralelo con el de Raquel Lazcano López [1]. Por esta razón, ambos comparten objetivos y, en consecuencia, su trabajo será referenciado con asiduidad.

CAPÍTULO 2. ANTECEDENTES

A lo largo de este capítulo se llevará a cabo un estudio del estado del arte de los conceptos y tecnologías relacionadas con el Proyecto Fin de Grado que se desarrolla en este documento. En dicho estudio, se hará una descripción detallada y, posteriormente, se resaltarán los aspectos más significativos de cada uno de ellos.

En primer lugar, se realizará un estudio del estado del arte de las imágenes hiperespectrales, realizando una definición formal y resaltando sus características más importantes. A continuación, se introducirá el método de procesamiento utilizado a la hora de trabajar con estas imágenes y, tras este estudio, se destacarán los aspectos más relevantes para el desarrollo del Proyecto Fin de Grado.

Una vez definidas las imágenes hiperespectrales y su proceso de análisis, se realizará un estudio de las aplicaciones en las que se utiliza esta tecnología, centrándonos especialmente en las que se encuentran dentro del campo de la medicina.

Posteriormente, se realizará un análisis de la velocidad de procesamiento de las imágenes hiperespectrales. Este aspecto es especialmente crítico debido a que uno de los objetivos principales es, en la medida de lo posible, alcanzar el tiempo real a la hora de analizar cada imagen.

A continuación, basándonos en los puntos fuertes y débiles de la utilización de imágenes hiperespectrales, se analizará la necesidad de la utilización de plataformas multinúcleo dentro del proyecto HELICoiD. Tras este análisis, se realizará un estudio del estado del arte de las plataformas multinúcleo presentes en el mercado. Para ello, se estudiarán las prestaciones de cada una de ellas y, posteriormente, se realizará una comparativa de especificaciones, señalando, para finalizar, los puntos fuertes y débiles que presentan unas respecto a otras.

Tras analizar la necesidad de utilizar plataformas multinúcleo, nos centraremos en analizar un lenguaje creado con el fin de facilitar la implementación de un programa en distintos núcleos. RVC – CAL será el lenguaje analizado y, en consecuencia, el utilizado a lo largo del desarrollo del Proyecto Fin de Grado. Durante este estudio, se explicarán sus principales características, se aportará un análisis de cada uno de los componentes que lo integran y, por último, se realizará un resumen de las principales ventajas y desventajas de su utilización, haciendo especial hincapié en los que afectan de manera directa a esta investigación.

Por último, se realizará un estudio del estado del arte de la herramienta software PAPI (*Performance Application Programming Interface* o Interfaz de Programación para el Rendimiento de Aplicaciones). Esta herramienta se ha utilizado para analizar el consumo de recursos que tiene el procesador donde se ejecuta la aplicación. El objetivo de la inclusión de esta herramienta es determinar los cuellos de botella presentes en la cadena de procesado para, en el caso de que fuera posible, reducir al mínimo estos puntos críticos.

Cabe destacar que este estudio se ha realizado en paralelo con Raquel Lazcano López. Por lo tanto, no todos los apartados están recogidos con el mismo nivel de detalle dentro de este documento y, en consecuencia, se harán referencias reiteradas a su Proyecto Fin de Grado [1].

2.1. Imágenes hiperespectrales: definición y cadena de procesado

Durante el desarrollo de este Proyecto Fin de Grado, se pretende elaborar una cadena de procesado de una imagen hiperespectral. Por ello, a lo largo de este apartado, se definirá el concepto de imagen hiperespectral y, posteriormente, se explicarán las distintas etapas que componen la cadena de procesamiento.

Una imagen hiperespectral se define como una imagen compuesta tanto de resolución espacial como de resolución espectral. Esto significa que, para un mismo pixel, la imagen no sólo consta de tres valores de reflectancia (equivalentes a las bandas de color rojo, verde y azul) tal y como sucede en las imágenes conocidas comúnmente como RGB (*Red-Green-Blue*). En el caso de las imágenes hiperespectrales, cada pixel puede llegar a tener cientos de valores de reflectancia asociados a diferentes bandas repartidas por todo el espectro electromagnético; estas bandas pueden pertenecer al espectro visible, al infrarrojo o al ultravioleta – los dos últimos invisibles para el ojo humano. Esta idea queda plasmada gráficamente en la figura 2.1.

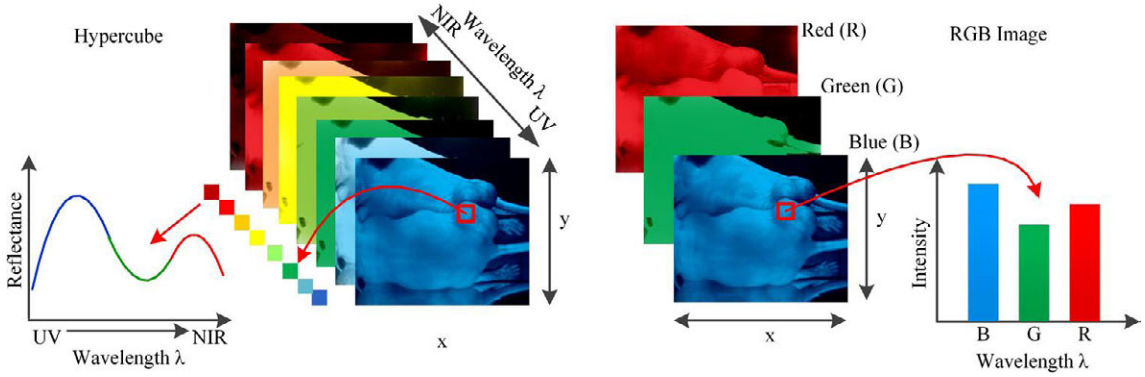


Fig. 2.1. Comparativa de imagen hiperespectral (izquierda) con imagen RGB (derecha) [2]

Una de las mayores ventajas de la utilización de imágenes hiperespectrales es que aportan información perteneciente al espectro electromagnético invisible para el ojo humano. Sin embargo, el volumen de datos es muy elevado y, por lo tanto, su procesado requiere un gran consumo de recursos.

A continuación, para hablar del procesamiento de imágenes hiperespectrales, cabe destacar que, en las implementaciones habitualmente utilizadas, esta tarea se divide en cuatro fases.

La primera de ellas, denominada *estimación del número de endmembers,* suele considerarse opcional, ya que su objetivo es obtener un número aproximado de los distintos elementos – denominados habitualmente *endmembers* – que componen dicha imagen.

En segundo lugar, existe una fase también opcional – pero imprescindible cuando se quiere alcanzar el procesado en tiempo real – denominada *reducción dimensional*. Esta etapa tiene por objetivo reducir el volumen de datos de la imagen original sin perder información relevante.

A continuación, se lleva a cabo lo que se conoce como *extracción de endmembers,* donde se obtienen las firmas espectrales de cada elemento, es decir, los valores de reflectancia asociados a un *endmember* de la imagen hiperespectral.

Para finalizar, se realiza la última fase: *estimación de abundancias.* En esta fase se estima el grado de coincidencia de cada *endmember* y su distribución espacial en la imagen captada.

Cabe destacar que tanto el concepto de imagen hiperespectral como las distintas etapas que componen la cadena de procesado están definidos detalladamente en [1].

2.2. Aplicaciones médicas

Tras explicar el concepto de imagen hiperespectral y su cadena de procesado, se resume a continuación el estudio realizado por Raquel Lazcano López en su Proyecto Fin de Grado [1] sobre los ámbitos de utilización de estas imágenes.

En primer lugar, el campo en el que más tiempo lleva aplicándose esta tecnología es en el estudio y análisis de la superficie terrestre. Las aplicaciones más extendidas dentro de este ámbito son la identificación de cultivos y la localización de yacimientos mineros y petrolíferos. Una de las aplicaciones más innovadoras ha sido el estudio de la superficie de Marte donde, gracias al descubrimiento de materiales que necesitan agua durante su formación mediante la detección de su firma espectral, se ha demostrado la existencia de agua líquida en algún momento de la historia de este planeta.

Un caso destacable de la aplicación de imágenes hiperespectrales fue durante el hundimiento del petrolero denominado *Prestige* frente a la costa gallega. En este caso, la utilización de imágenes hiperespectrales permitió localizar vertidos en la superficie oceánica menores de 5 metros cuadrados.

Durante los últimos años han aparecido dos campos de aplicación realmente novedosos: la ciencia forense y la verificación de documentos. En el campo de la ciencia forense su utilización se centra en la detección de huellas latentes y en la detección de residuos. Por su parte, en el campo de verificación de documentos su carácter no invasivo ha hecho que se utilice para no provocar deterioro y conservar los distintos documentos históricos intactos.

Por último, uno de los sectores que se han beneficiado más recientemente – y también el que más interesa de cara al desarrollo de este Proyecto Fin de Grado – es el de la medicina, en concreto, la detección de tumores cancerígenos.

Como se explica en profundidad en [1], se han realizado diversos estudios acerca de la viabilidad de detectar y localizar tumores cancerígenos, primero en animales y, posteriormente, en humanos, con resultados satisfactorios para ambos casos.

En principio, un ejemplo clásico de aplicación de esta técnica es detectar con éxito un tumor humano de próstata provocado en un ratón de laboratorio [3]. Posteriormente, se utilizaron las imágenes hiperespectrales de alta resolución para detectar – también en ratones – tumores residuales y, de esta manera, retirar los posibles restos con el objetivo de impedir que el tumor se reproduzca de nuevo [4]. Por último, en humanos, se han realizado estudios para detectar tumores gástricos *ex vivo* – tras realizar una biopsia – [5] y tumores en la lengua *in vivo* – sin realizar extirpación – [6]. El resultado de este último caso se observa en la figura 2.2.

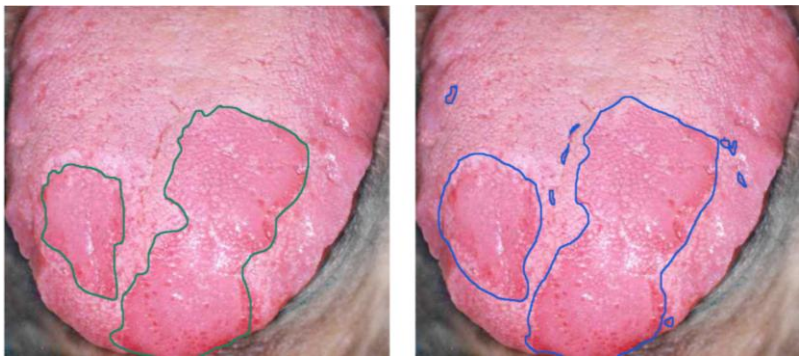


Fig. 2.2. Tumor delimitado por experto (izquierda) y tumor delimitado mediante análisis espectral (derecha) [6]

2.3. Velocidad de análisis

Debido a que, como se mencionó en el capítulo 1 de este mismo proyecto, uno de los objetivos principales es la consecución – en la medida de lo posible – del procesamiento en tiempo real, uno de los puntos más importantes a tener en cuenta es el tiempo que se tarda en analizar una imagen hiperespectral y la posibilidad de reducir este tiempo lo máximo posible.

Como concluye Raquel Lazcano López en su Proyecto Fin de Grado [1], tras realizar un análisis en profundidad de esta característica, la literatura de imágenes hiperespectrales normalmente no aporta análisis de tiempos de las aplicaciones concretas que utilizan, o bien no mencionan el algoritmo – o incluso el entorno/plataforma – concreto que han utilizado.

Por otro lado, sí se ha encontrado un análisis exhaustivo realizado por Sergio Sánchez [7] sobre la mejora de rendimiento que implica la paralelización de una cadena de procesado. Para ello utiliza un sistema mononúcleo para el procesado secuencial y dos GPUs distintas de la empresa NVIDIA para el procesado en paralelo. Los resultados obtenidos se muestran en la tabla 2.1 para el caso secuencial más rápido y en la tabla 2.2 para el más lento.

En dichas tablas observamos que, gracias a la paralelización, se obtiene una mejora máxima de más de 13 veces para el caso más rápido y de más de 38 para el caso más lento; en porcentaje, sería reducir el tiempo de procesado un 92.5% y un 97.4%, respectivamente.

Por otro lado, como dato adicional, podemos observar que otro método para mejorar el tiempo de procesado de la cadena implementada es el compilador utilizado, ya que, como se observa en ambos casos, se reduce el tiempo de procesado – tanto secuencial como paralelizado – al utilizar el compilador icc.

	LOAD	VD	PCA	N-FINDR	UCLS	SAVE	TOTAL	SPEEDUP
Serie gcc	0.186	5.555	5.401	0.883	0.313	0.008	12.346	
Tesla gcc	0.202	0.411	0.147	0.112	0.054	0.008	0.934	13.219
GTX gcc	0.204	0.423	0.137	0.107	0.045	0.010	0.926	13.331
Serie icc	0.170	3.940	2.253	0.847	0.251	0.008	7.470	
Tesla icc	0.183	0.420	0.149	0.115	0.054	0.012	0.933	8.007
GTX icc	0.194	0.441	0.132	0.094	0.041	0.012	0.914	8.174

Tabla 2.1. Tiempo de procesamiento - caso más rápido [7]

	LOAD	HYSIME	SPCA	N-FINDR	NCLS	SAVE	TOTAL	SPEEDUP
Serie gcc	0.179	28.282	5.428	0.600	77.685	0.007	112.181	
Tesla gcc	0.197	1.853	0.169	0.083	1.075	0.007	3.383	33.161
GTX gcc	0.201	1.655	0.145	0.069	0.859	0.007	2.936	38.207
Serie icc	0.164	20.100	2.222	0.526	61.435	0.007	84.453	
Tesla icc	0.183	1.842	0.171	0.085	1.075	0.007	3.363	25.116
GTX icc	0.183	1.664	0.143	0.069	0.859	0.007	2.926	28.867

Tabla 2.2. Tiempo de procesamiento - caso más lento [7]

2.4. Plataformas multinúcleo

Como se ha explicado en apartados anteriores de este mismo capítulo, la cantidad de datos generada al utilizar imágenes hiperespectrales es enorme y la capacidad para procesar dichos datos es un problema serio a tratar ya que, como dice Sergio Sánchez en su tesis doctoral [7]: "*a pesar de que diferentes instituciones como NASA o la Agencia Europea del Espacio (ESA) obtienen varios Terabytes de datos hiperespectrales cada día, se estima que una parte significativa de dichos datos no son nunca utilizados o procesados, sino meramente almacenados en una base de datos y muchas veces reservados para futuro uso que, en ocasiones, no se materializa debido a los enormes requerimientos computacionales para almacenar, procesar y distribuir dichas imágenes de forma eficiente.*"

Instintivamente, nos surge la solución de desarrollar procesadores más potentes que puedan soportar el análisis de una cantidad tan abrumadora de datos. Esta idea nos lleva a analizar la capacidad de cómputo que puede soportar un procesador, lo que nos conduce directamente a lo que se conoce como ley de Moore.

En esta ley de Moore – desarrollada por Gordon Moore, fundador de Intel, en los años sesenta – se predijo que la densidad de los transistores presentes en un procesador se duplicaría cada 24 meses. Este aumento de la densidad se debe a la reducción del tamaño de los transistores conforme mejora la tecnología disponible en el mercado. Un aumento del número de transistores presente en un único procesador le otorgaría la capacidad de incrementar la frecuencia a la que trabaja pero, a su vez, produciría un gran aumento de la energía necesaria para poner en marcha dicho procesador, aumentando su nivel de tensión de alimentación y, en consecuencia, el calor generado mientras está en funcionamiento.

Tal y como se recoge en la tesis doctoral de Marko Bertogna [8], Intel se percató del problema que surgía al aumentar la capacidad de procesamiento de los procesadores cuando, en mayo de 2004, empezaron a desarrollar el que se conocería como procesador Pentium Tejas. Como se observa en la figura 2.3, el aumento de la densidad de transistores en un mismo procesador conlleva al aumento de la potencia generada por centímetro cuadrado, lo que a su vez implica un aumento en la temperatura de dicho procesador, pudiendo llegar a un punto en el que se haría completamente insostenible.

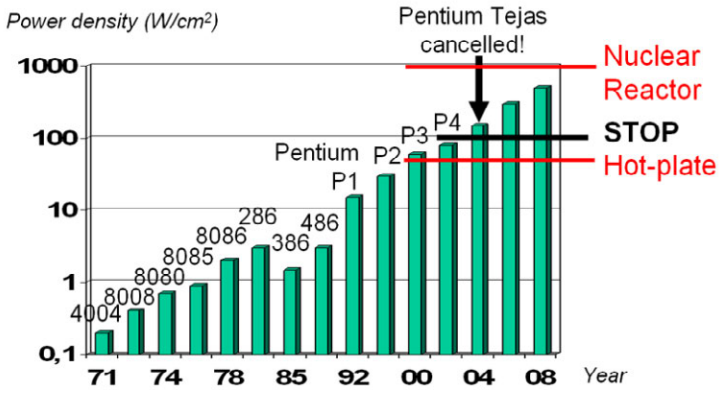


Fig. 2.3. Aumento cronológico de la densidad de potencia en un procesador [8]

Tras observar este posible suceso, los fabricantes tanto de ordenadores como de plataformas hardware dedicaron su trabajo a realizar sistemas basados en varios núcleos, manteniendo, como es de esperar, una alta capacidad de cómputo en cada uno de los procesadores por separado. Esta solución permite tener procesadores de grandes prestaciones trabajando en paralelo y, de esta forma, mantener estable la temperatura a la que trabajan los distintos núcleos – así como su frecuencia de funcionamiento y la tensión de alimentación requerida –, aumentando la capacidad de cómputo total de la plataforma desarrollada.

Por tanto, debido a la ingente cantidad de datos a procesar en este proyecto y a la necesidad de realizar dicho procesamiento rápidamente - ya que, como hemos mencionado anteriormente, uno de los principales objetivos de este proyecto es alcanzar el tiempo real -, se deduce que es estrictamente necesaria la utilización de plataformas multinúcleo. Estas plataformas, correctamente configuradas, nos facilitarán el manejo y análisis de gran cantidad de datos y, como se ha mencionado en el apartado anterior, reducirá sustancialmente el tiempo necesario para su procesamiento.

La utilización de estas plataformas conlleva, sin embargo, la aparición de nuevos problemas como, por ejemplo, la repartición, sincronización y programación de las tareas que se ejecutarán en cada uno de los procesadores. Para ello, se han realizado diversos estudios de paralelización de código, sincronización de procesadores, etc, tales como [8], [9] y [10].

A su vez, para facilitar la elaboración de aplicaciones que basen su funcionamiento en el uso de varios procesadores, se han desarrollado nuevos lenguajes de programación basados en flujo de datos – divididos en bloques que constan de una serie de entradas, unas funcionalidades internas y una serie de salidas –. Uno de ellos, RVC – CAL, se explica detalladamente en la siguiente sección del presente capítulo.

A continuación, se realiza un estudio de las distintas plataformas que ofertan los fabricantes en el mercado actual. Para ello, nos centraremos en las características que se muestran en la tabla 2.3. Posteriormente, se analizan las prestaciones de plataformas punteras de cada uno de estos distribuidores y, para finalizar, se aporta una tabla comparativa – y su respectivo análisis – en la que se observan las diferencias existentes entre ellas. Cabe destacar que Eduardo Juárez y Rubén Salvador han realizado este estudio en profundidad dentro del grupo GDEM con el fin de seleccionar la plataforma que se utilizará dentro del proyecto HELICoiD [11].

Características de la arquitectura	Herramientas de desarrollo	Plataforma como sistema	Soporte
Tipo de arquitectura	Modelo de computación	Único chip	Entrenamiento del personal
Número de núcleos	Lenguaje	Plataforma de desarrollo	Aceptación y uso actual
Capacidad de procesamiento	Programador de tareas	Interfaces	
Memoria		Uso remoto	

Tabla 2.3. Características a analizar de cada plataforma.

Sabiendo que nuestro principal trabajo se basa en el análisis y procesado de imágenes, es indispensable barajar la posibilidad de trabajar con GPUs (Unidad de Procesamiento Gráfico – de sus siglas en inglés, *Graphics Processing Unit*) [12].

Una GPU es un circuito electrónico diseñado para procesar imágenes de manera rápida y eficiente, añadiendo las imágenes procesadas a un *buffer*, con el objetivo principal de mostrarlas a través de un *display*. Estas unidades de procesado se utilizan en dispositivos móviles, sistemas empotrados, ordenadores personales, estaciones de trabajo y consolas de videojuegos, y sus aplicaciones van desde el análisis estándar de imágenes en un dispositivo [13], hasta la generación de hologramas [14] o, incluso, la implementación de redes neuronales [15].

A su vez, se han desarrollado aplicaciones – basadas en la utilización de GPUs – enfocadas a acelerar el procesamiento de datos – principalmente, imágenes y vídeos-. Uno de estos ámbitos de utilización ha sido el análisis de imágenes hiperespectrales y, en este caso, los resultados obtenidos han sido satisfactorios [16].

Analizando los distribuidores principales de GPUs del mercado actual, observamos que, para nuestro ámbito concreto – procesado de gran cantidad de datos y rendimiento computacional elevado –, destacan dos fabricantes: NVIDIA[2] y AMD/ATI[3].

NVIDIA GPUs

De las plataformas disponibles en esta empresa, analizaremos la última lanzada al mercado, la cual, a su vez, es la más potente de las desarrolladas por NVIDIA. Esta plataforma se ha denominado *Tesla K40* y se muestra en la figura 2.4.

Fig. 2.4. Tesla K40.

Analizando esta placa a nivel de arquitectura observamos que consta de 15 multiprocesadores en cadena (o SMX) con 192 núcleos CUDA cada uno. Esta GPU aúna más de 2800 núcleos de procesamiento trabajando a 4.29 *TeraFlops* utilizando precisión simple – 32 bits – o 1.43 a precisión doble – 64 bits.

En cuanto a consumo, este dispositivo trabaja a una potencia máxima de 235W. Esto implica un rendimiento de 18.25 *GigaFlops/W* – trabajando con una distribución de 32 bits – y de 6.08 *GigaFlops/W* – utilizando una distribución de 64 bits-.

Para concluir con el análisis a nivel estructural, queda mencionar que la máxima memoria proporcionada para esta plataforma es de 12*GB* y que, en cuanto al ancho de banda en la comunicación con el procesador, se llegan a obtener velocidades de hasta 288 *Gbytes/s*.

Respecto a las herramientas de desarrollo disponibles para este tipo de tarjetas, NVIDIA presenta un amplio abanico de herramientas de análisis de rendimiento; por ejemplo, para análisis software, *NVIDIA Visual Profiler* y, para análisis hardware¸ *PAPI CUDA*.

Existen varios lenguajes utilizados en la implementación de aplicaciones que utilizan GPUs de NVIDIA, entre los cuales destacan CUDA C/C++ y OpenCL. A su vez, el compilador utilizado es *NVIDIA's CUDA Compiler* (NVCC). Una desventaja que presenta el uso de estas plataformas es que el lenguaje utilizado es demasiado específico. Este problema conllevaría la necesidad de tener expertos en el uso de CUDA para desarrollar las aplicaciones y solventar los errores que pudieran surgir durante su funcionamiento normal.

[2] http://www.nvidia.es
[3] http://www.amd.com

AMD/ATI GPUs

De cara a analizar las plataformas desarrolladas por esta empresa, tenemos que distinguir entre plataforma dedicada a funcionar como servidor (de manera independiente) – la última lanzada al mercado ha sido la *FirePro W9100*, la cual se observa en la figura 2.5 (izquierda) – y, por otro lado, las desarrolladas como estaciones de trabajo (utilizadas por un procesador externo) – por ejemplo, la *FirePro S10000 dual-GPU* que se observa en la figura 2.5 (derecha).

Fig. 2.5. FirePro W9100 (izquierda), FirePro S10000 (derecha).

A nivel de arquitectura, la plataforma *FirePro W9100* consta de 44 unidades de computación – lo que reúne un total de 2816 procesadores en cadena –, alcanzando los 5.24 *TeraFlops* con precisión simple y los 2.62 *TeraFlops* trabajando con precisión doble. Por otro lado, la capacidad de memoria del sistema es de 16*GB* con un ancho de banda de 320*GB/s*.

En cuanto a la arquitectura de la plataforma *FirePro S10000*, conviene señalar que está formada por dos GPUs trabajando en paralelo y compuestas, cada una de ellas, de 1792 procesadores en cadena, alcanzando en conjunto una velocidad de procesamiento de 5.91 *TeraFlops* en precisión simple y 1.48 *TeraFlops* en precisión doble. A su vez, esta plataforma incluye una memoria GDDR5 de 6*GB* soportando una comunicación entre las GPUs a una velocidad de 480*GB/s*.

El consumo máximo de ambas plataformas es de 275*W*, lo que se traduce en un rendimiento de, aproximadamente, 19 *GigaFlops/W* – trabajando a precisión simple – y de 9.5 *GigaFlops/W* – trabajando a precisión doble.

AMD oferta como entorno de desarrollo para sus plataformas una amplia gama de herramientas software entre las que destacan *AMD OpenCL™ APP SDK* para el desarrollo de aplicaciones con ejecución en paralelo y *CodeXL* para el análisis y corrección de errores.

El lenguaje de programación utilizado por AMD es *OpenCL* que, a su vez, consta de librerías específicas de análisis matemático y permite el acceso a los registros PAPI – explicados más adelante en este mismo capítulo – lo que nos daría una gran soltura a la hora de desarrollar algoritmos y analizar el rendimiento de los mismos.

Una vez analizada la opción de utilizar GPUs, se pasa a barajar las opciones alternativas que ofrecen el resto de fabricantes para el procesado de gran volumen de datos.

Intel Xeon Phi Coprocessors

La solución desarrollada por Intel[4] es la utilización de un coprocesador basado en la arquitectura MIC (de sus siglas en inglés, *Many Integrated Core Architecture*) controlado por un sistema basado en el procesador *Intel Xeon*. Una plataforma que integra este conjunto presentaría el diseño que se observa en la figura 2.6.

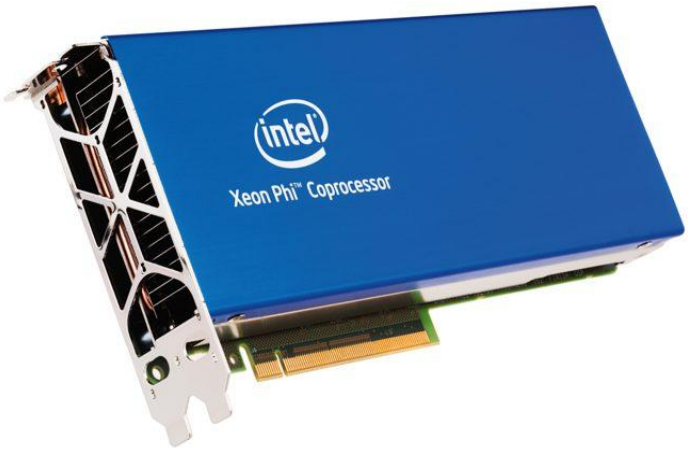


Fig. 2.6. Intel Phi coprocessor.

Esta plataforma, a nivel de arquitectura, se presenta en tres versiones distintas. La primera de ellas, con 57 núcleos, alcanza una velocidad de 1003 *GigaFlops;* la segunda alcanza 1011 *GigaFlops* utilizando 60 núcleos; por último, la más potente de ellas trabaja a una velocidad de 1208 *GigaFlops* y está compuesta de 61 núcleos – todas ellas analizadas para precisión doble-.

En cuanto al consumo, éste varía entre los 225*W* y los 300*W,* dependiendo de la versión utilizada y alcanzando un rendimiento de 4 *GigaFlops/W* y 4.5 *GigaFlops/W* respectivamente. A su vez, la memoria disponible puede variar desde los 6*GB* hasta los 16*GB,* con un ancho de banda útil de entre 240 y 352 *GB/s*.

La utilización de esta arquitectura permite generar las aplicaciones en lenguaje C/C++ o Fortran ya que, dentro de las herramientas de desarrollo disponibles, existe un compilador que soporta estos lenguajes. A su vez, se aporta una librería de análisis matemático denominada MKL (del inglés, *Math Kernel Library*) importante dentro de nuestro proyecto.

Kalray Manycore Processors

Por otra parte, una solución aportada por el fabricante Kalray[5] es el uso de su serie MPPA (*Multi Purpose Processor Array),* que tiene como objetivo principal tener un alto rendimiento computacional.

Actualmente, existe en el mercado una plataforma denominada *MPPA 256* – la cual se puede observar en la figura 2.7. Esta plataforma consta de 256 procesadores divididos en 16 *clusters*. Cada procesador funciona individualmente a una frecuencia de 400MHz, alcanzando así un total de 230 *GigaFlops*.

[4] http://www.intel.es
[5] http://www.kalray.eu/

Fig. 2.7. MPPA 256.

El punto fuerte de esta plataforma se centra en el apartado del rendimiento ya que, al consumir únicamente 5*W*, se alcanzan niveles de 50 *GigaFlops/W*. En cuanto a la comunicación, cabe destacar que ésta únicamente es posible entre *clusters* adyacentes y que alcanza velocidades superiores a 3.2*GB/s*.

Kalray aporta un entorno de desarrollo completo que incluye herramientas de análisis, detección de errores, desarrollo, compiladores, etc. Esta plataforma, a su vez, contempla la utilización de aplicaciones implementadas tanto en lenguaje C/C++ como en OpenCL.

Una característica destacable dentro de esta arquitectura es la forma en la que se expresa el paralelismo. Para ello, se genera una serie de *agents* que llevan a cabo la transmisión de datos entre *clusters,* lo que implica la eliminación de la posibilidad de existencia de variables globales del sistema o compartidas entre varios *clusters*.

Hyper X Processors

Para finalizar, es necesario destacar la aportación de Coherent Logix[6]. Esta empresa ha lanzado un procesador llamado *HyperX hx3100* como solución al procesado de datos de alto rendimiento y bajo consumo. Una plataforma perteneciente a esta familia se observa en la figura 2.8.

Fig. 2.8. Plataforma HyperX.

Arquitecturalmente, este procesador está conformado por 100 núcleos de procesado trabajando a una frecuencia de 500MHz, lo que hace que, utilizando una arquitectura de 32 bits, se alcance una velocidad máxima de 25 *GigaFlops*.

[6] http://www.coherentlogix.com/

A su vez, este procesador está alimentado a 2.5*W* por lo que, en cuanto a rendimiento se refiere, nos proporciona una tasa de 10 *GigaFlops/W*.

Una característica propia de este procesador es que tiene una estructura de memoria programable a nivel de software. Esta distribución permite mapear a tiempo real el algoritmo y sincronizar las tareas que se ejecuten en cada procesador – realizar paralelismo – de manera instantánea, estableciendo comunicaciones entre ellos de tiempo real. Esto daría como resultado que las variables utilizadas puedan considerarse globales para todo el procesador.

Esta plataforma se puede configurar utilizando lenguaje C y, como dato adicional, cabe destacar que incluye una librería específica para análisis de imágenes hiperespectrales.

Comparativa

En la tabla 2.4 se recoge, a modo de resumen, las características de las distintas plataformas analizadas. A continuación, analizando la tabla resumen y teniendo en cuenta todo lo dicho en el análisis individual, realizamos una comparativa en la que resaltamos las ventajas e inconvenientes más importantes de cada una de ellas.

Las ventajas de la plataforma *Tesla K40* son su alta velocidad de procesado con un rendimiento bastante elevado. A su vez, tiene detrás una gran empresa en el mundo de las GPUs como es NVIDIA. Sin embargo, el lenguaje de programación es propio de la empresa – CUDA C/C++ – por lo que el desarrollo de aplicaciones sería complejo.

Por otro lado, las plataformas de la serie *FirePro* tienen la mayor velocidad de procesamiento de las opciones estudiadas pero, a su vez, en cuanto a memoria se refiere, disponen de tan sólo 6GB de RAM.

Con respecto a la plataforma *Intel Xeon* cabe destacar que, teniendo el menor número de procesadores entre las plataformas analizadas, es capaz de llegar al *TeraFlop* de velocidad de procesamiento. Sin embargo, presenta un consumo muy elevado y, por ello, el rendimiento de esta plataforma es el menor de todas.

De las plataformas estudiadas, la que nos proporcionaría un mayor rendimiento es la *MPPA 256*; en contraposición, la velocidad máxima alcanzada es de 230 *GigaFlops* y, además, no existe la posibilidad de utilizar variables globales del sistema. Esta limitación tiene como resultado que, si se desarrollan aplicaciones con un alto grado de comunicación entre procesadores, esta plataforma consumiría muchos recursos y tiempo a la hora de realizar las comunicaciones pertinentes.

Por último, en la plataforma *HyperX hy3100* de Coherent Logix, destaca la capacidad de transmitir datos entre procesadores en tiempo real; por el contrario, únicamente alcanza una velocidad de procesamiento de 25 *GigaFlops*.

Plataforma	Empresa	Procesadores	Velocidad	Consumo	Rendimiento	Memoria	Lenguaje
Tesla K40	NVIDIA	>2800	· 4.29 TF – 32 bits · 1.43 TF – 64 bits	235 W	· 18.25 GF/W – 32 bits · 6.08 GF/W – 64 bits	12 GB – 288 GB/s	CUDA C/C++ OpenCL
FirePro W9100	AMD/ATI	2816	· 5.24 TF – 32 bits · 2.62 TF – 64 bits	275 W	· 19.05 GF/W – 32 bits · 9.53 GF/W – 64 bits	6 GB – 480 GB/s	OpenCL
FirePro S10000	AMD/ATI	2x1792	· 5.91 TF – 32 bits · 1.48 TF – 64 bits	275 W	· 21.5 GF/W – 32 bits · 5.38 GF/W – 64 bits	6 GB – 480 GB/s	OpenCL
Intel Xeon	Intel	58-60-61	1003-1011-1208 GF – 64 bits	225-300 W	4-4.5 GF/W	6-16 GB – 240-352 GB/s	C/C++ Fortran
MPPA 256	Kalray	256	230 GF	5 W	50 GF/W	-	C/C++ OpenCL
HyperX hx3100	CoherentLogix	100	25 GF – 32 bits	2.5 W	10 GF/W	-	C

Tabla 2.4. Resumen de prestaciones de plataformas multinúcleo

2.5. RVC – CAL

Una vez establecido que la plataforma utilizada será una plataforma multinúcleo, pasamos a explicar el lenguaje de programación que se utilizará a lo largo de este Proyecto Fin de Grado: RVC – CAL [17]. Este lenguaje está basado en el lenguaje conocido como CAL (*CAL Actor Language* o Lenguaje de Actores CAL) [18]. Debido a la gran similitud que guardan ambos lenguajes de programación, es necesario explicar la estructura que utiliza CAL para entender el funcionamiento de RVC – CAL.

CAL es un lenguaje de programación de alto nivel basado en flujo de datos. Este lenguaje fue desarrollado en 2001 en la Universidad de Berkeley con el objetivo de facilitar la implementación de aplicaciones divididas en bloques de funcionalidad y, a su vez, establecer una comunicación entre dichos bloques. Esta idea de codificación por bloques nos posibilita la realización de un reparto de los mismos entre procesadores y, de este modo, conseguir paralelizar la ejecución de la aplicación desarrollada.

El concepto de paralelización de procesadores resulta complejo de implementar cuando se utilizan lenguajes de programación secuencial tales como C, C++, Java, etc. En consecuencia, la utilización de CAL nos permite alcanzar un nivel de abstracción en el que la paralelización es transparente para el usuario.

Para realizar esta división en bloques, CAL utiliza una estructura básica compuesta de actores, acciones y *networks* – o redes –. A continuación, se analizan en profundidad estos conceptos, con el objetivo de obtener una idea clara de los elementos principales que componen la estructura de CAL:

- El actor es la unidad más básica y característica de CAL. Este actor es lo que, con anterioridad, se ha denominado bloque y, tal y como se observa en la figura 2.9, se trata de una entidad compuesta de una serie de entradas, una o varias funcionalidades internas y un conjunto de salidas. Dichas entradas y salidas nos permiten ver claramente que cada actor puede establecer una comunicación con otro actor. El proceso de funcionamiento de un actor sería, en resumen, utilizar los datos recibidos – e incorporar nuevos de un fichero almacenado en disco si es necesario – para realizar una serie de operaciones cuyo resultado es enviado a otro actor y, de esta manera, continuar la ejecución del programa. Los bloques de datos que se transmiten entre actores se denominan *tokens* dentro de este lenguaje.

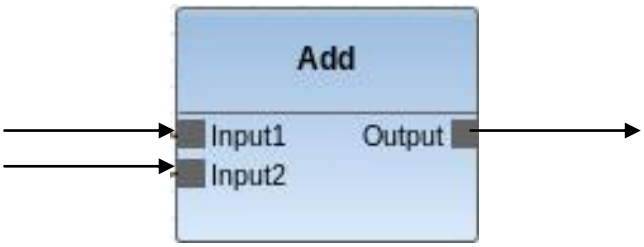


Fig. 2.9. Gráfico de un actor en lenguaje CAL

- Una vez definido el concepto de actor como bloque funcional, es necesario definir el siguiente concepto característico presente en este lenguaje: la acción. Se entiende por acción cada una de las funcionalidades internas existentes dentro de un actor.

Cada acción, a su vez, puede tener definidas sus propias entradas y salidas (que deben estar contempladas en la declaración del actor) y, en consecuencia, realizar una funcionalidad distinta al resto. Esta característica permite al usuario organizar el código de tal manera que se divida, dentro de un mismo actor, en diversas funcionalidades que dependan – o no – las unas de las otras.

Otra de las características principales de las acciones es la capacidad que nos ofrece CAL para condicionar su ejecución ya que, aprovechando esta cualidad del lenguaje, se puede implementar la secuenciación de acciones que más se adapte a nuestras necesidades de manera rápida y eficiente.

- Tras definir la estructura interna de un actor, únicamente nos queda por definir el concepto de *network*. Entendemos por *network* el conjunto de varios bloques funcionales – en su mayoría actores – conectados entre sí. Cabe destacar que, en la configuración estándar de una *network*, existen, aparte de los actores explicados anteriormente, unas instancias específicas para implementar la interfaz de entrada y salida del sistema. Estas instancias se denominan dentro de este lenguaje *input and output ports* – puertos de entrada y salida –. Debido a la complejidad variable que puede tener la entrada o salida de un sistema, existen situaciones en las que la utilización de estos puertos no es viable; por ejemplo, cuando el número de entradas o salidas es muy numeroso o cuando la entrada/salida del sistema es compleja. En estos casos, se utilizan actores denominados *source* – fuente – y *display* – visor – desarrollados para esta funcionalidad específica. Un ejemplo clásico en el que se da esta situación es cuando la entrada del sistema es un fichero almacenado en disco.

Una vez explicada la estructura básica de este tipo de lenguajes, somos capaces de dar una definición formal y explicar en profundidad RVC – CAL.

RVC – CAL es un lenguaje de flujo de datos – estructura heredada de CAL –. Su principal característica es, tal y como indica su nombre, su orientación al desarrollo de codificadores y decodificadores de vídeo reconfigurables. Este lenguaje está siendo desarrollado por MPEG (*Moving Pictures Experts Group* o Grupo de Expertos en Imágenes en Movimiento) y, con él, están desarrollando un nuevo estándar de codificadores y decodificadores de vídeo [19]. Este nuevo estándar se basa en desarrollar los codificadores, no como un todo, si no como bloques funcionales que, unidos de una forma concreta, conformen un decodificador completo. Este desarrollo por bloques permite que, si se quiere modificar una fase del decodificador – o actor –, ya sea porque no funciona como se espera o se quieren variar sus prestaciones, únicamente habría que modificar un actor concreto o sustituir dicho actor por otro completamente distinto.

Para analizar este lenguaje en profundidad se dividirá su estudio en dos fases:

- En primer lugar, se estudiará el compilador ORCC (*Open RVC – CAL Compiler* o Compilador de RVC – CAL abierto), atendiendo a la estructura de una *network* y a la manera de establecer la comunicación entre actores.
- En segundo lugar, se estudiará la propia sintaxis del lenguaje, tocando así todos los aspectos necesarios para ser capaces de desarrollar nuestras propias aplicaciones.

Con el fin de facilitar la implementación de aplicaciones en lenguaje RVC – CAL se ha desarrollado un *plugin* de Eclipse denominado ORCC con el que se pueden observar claramente los elementos característicos de CAL explicados anteriormente.

Desde ORCC se puede observar gráficamente el concepto de *network* como una entidad compuesta por una serie de instancias conectadas entre sí a través de puertos de entrada y de salida. En la figura 2.10 se muestra una *network* sencilla implementada en ORCC. En ella únicamente se utiliza un actor básico – denominado *Add* – compuesto por dos entradas y una salida.

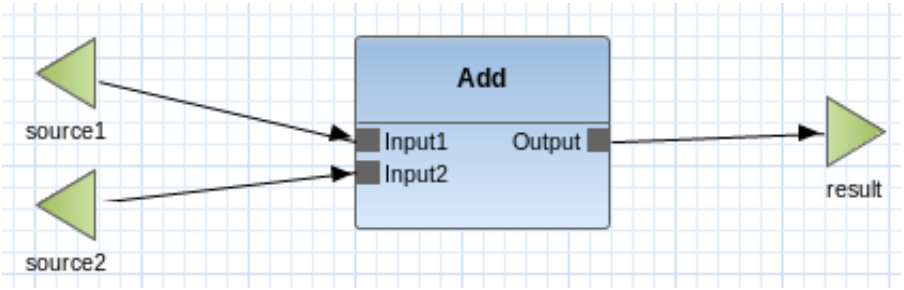


Fig. 2.10. Network básica en ORCC

Como se ha mencionado con anterioridad, los puertos de entrada y salida del sistema pueden llegar a ser muy complejos y sustituirse por entidades denominadas *source* y *display*. Estas entidades tendrían como característica principal o bien no tener entradas – en el caso del *source* –, o no tener salidas – en el caso del *display* –. Esto se debe a que estas entidades estarían leyendo o escribiendo los datos utilizados, por ejemplo, de disco. Una *network* en la que ocurre esto tendría una distribución similar a la que aparece en la figura 2.11. En este caso existen dos *source* distintos denominados *Actor1* y *Actor2* y un *display* denominado *Printer*.

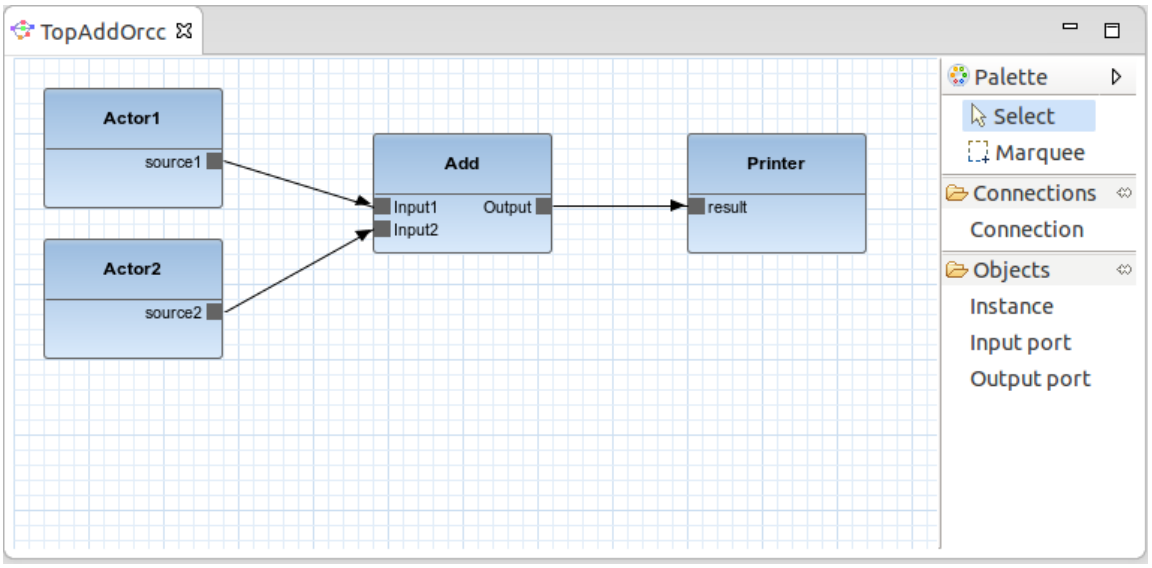


Fig. 2.11. Network con entrada y salida implementado con actores
(Fuente: http://orcc.sourceforge.net/tutorials/a-very-simple-network/)

En la figura 2.12 se puede observar un ejemplo de *network* compleja. En este caso se puede observar la capacidad de realizar un sistema en el que exista un alto grado de comunicación entre actores (e incluso, implementar realimentación). Al mismo tiempo, si tenemos en cuenta que cada instancia se puede ejecutar desde un procesador distinto, se puede comprobar de manera casi intuitiva que la paralelización de un sistema es una tarea relativamente sencilla.

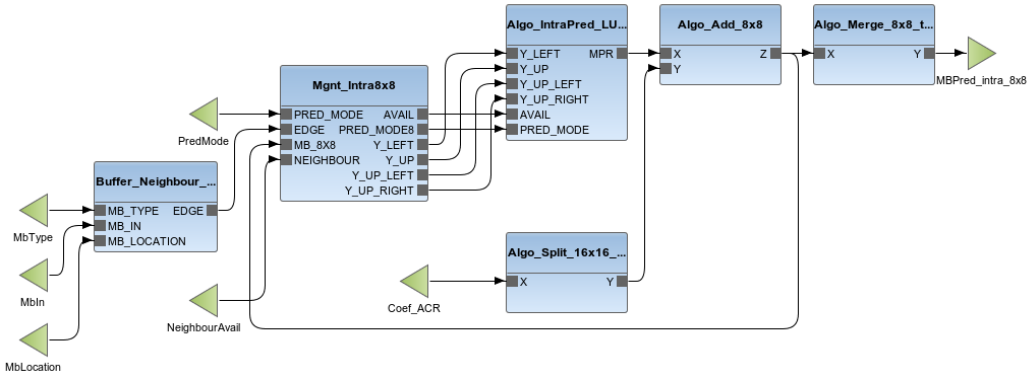


Fig. 2.12. Network con paralelismo complejo en ORCC
(Fuente: http://orcc.sourceforge.net/)

Cabe destacar que, durante la explicación de ORCC, se ha utilizado en algunas ocasiones la palabra entidad en lugar de actor para denominar a los distintos bloques que componen una misma *network*. Esto se debe a que una entidad puede ser, a su vez, otra *network* más pequeña –o con una funcionalidad más específica –. Esto se observa claramente en la figura 2.13, en la que existe una instancia constituida por un decodificador completo denominado *HevcDecoder*. En ella se observa que, para diferenciar estos tipos de entidades, se utilizan diferentes colores: amarillo para las *networks* y azul para los actores.

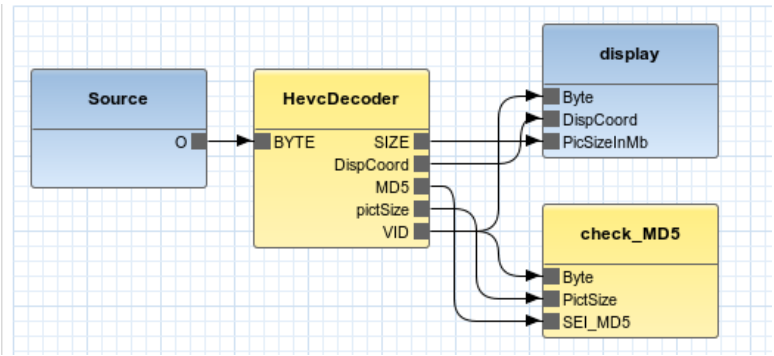


Fig. 2.13. Network incluida en otra más compleja
(Fuente: http://orcc.sourceforge.net/tutorials/make-an-hevc-decoder/)

Para finalizar la explicación de ORCC, queda puntualizar su capacidad para generar código en distintos lenguajes. Como se explica en [20], [21] y [22] se puede generar código en lenguajes tales como C, C++, Java, LLVM, etc. A modo de ejemplo, en la tabla 2.5 se recogen los distintos lenguajes generables para las diferentes tecnologías desarrolladas con RVC – CAL. En ella se observan los códigos para los que están disponibles (OK), los que se encuentran en desarrollo (NOK) y los que aún no se han empezado a implementar (N/A).

	MPEG-4 Part 2 SP	MPEG-4 Part 10	MPEG-H Part 2	JPEG
C	OK	OK	OK	OK
HLS	OK	NOK	NOK	N/A
Jade	OK	OK	OK	N/A
LLVM	OK	OK	OK	N/A
Promela	OK	N/A	N/A	N/A
Simulator	OK	OK	NOK	OK
TTA	OK	NOK	OK	N/A
Xronos	OK	NOK	N/A	OK
DAL	N/A	N/A	N/A	N/A

Tabla 2.5. Lenguajes disponibles en ORCC

A continuación, tras explicar el funcionamiento del *plugin* ORCC y su capacidad para construir una *network* y establecer las comunicaciones apropiadas, citando, a su vez, algunos de los lenguajes de programación que permite generar en su compilación, queda centrarse en la sintaxis propia del lenguaje RVC – CAL.

En primer lugar, para instanciar un actor se sigue la estructura expuesta en la figura 2.14. Como podemos observar en la imagen, se indica únicamente el nombre del actor y el tipo de dato de los distintos *tokens* de entrada y salida – *int, double, boolean,* etc-.

```
actor Actor() int Input1, double Input2, bool config ==> float Output:
```

Fig. 2.14. Sintaxis para la declaración de un actor

Estos actores pueden agruparse en paquetes y, a su vez, desde un actor, se puede importar el contenido de un paquete externo con el fin de incorporar las variables o las funciones implementadas en el mismo. Estas sentencias se realizan utilizando la sintaxis mostrada en la figura 2.15. Para indicar el paquete al que pertenece el actor que estamos codificando se utiliza el término *package*, mientras que para importar variables o funciones incluidas en un actor que forma parte de un paquete externo se utiliza la sentencia *import*.

```
package example;

import other_packages.*;
```

Fig. 2.15. Ejemplo de utilización de paquetes

Por otro lado, la declaración de una acción se realiza de la manera que se indica en la figura 2.16. En ella observamos dos entradas, que se corresponden con las entradas *Input1* e *Input2* declaradas en el actor ejemplo, y una salida, que se corresponde con la salida *Output* de ese mismo actor. Cabe destacar que en la declaración de la acción es donde, si procede, se indica el tamaño del *array* asociado al *token* de entrada o de salida. Este tamaño se indica utilizando el comando *repeat.*

```
tarea1: action Input1:[array] repeat 256, Input2:[extra_value] ==> Output:[result]
```

Fig. 2.16. Declaración de una acción

Por tanto, la estructura completa de un actor sería la que se observa en la figura 2.17. En ella se muestra el actor utilizado anteriormente, el paquete al que pertenece, la utilización de otros paquetes y, por último, las diversas acciones en las que se divide dicho actor.

La acción denominada *tarea2* es un ejemplo de que, al igual que los actores, no es necesario que todas las tareas consten de entradas y salidas.

```
package example;

import other_packages.*;

actor Actor() int Input1, double Input2, bool config ==> float Output:

    tarea1: action Input1:[array] repeat 256, Input2:[extra_value] ==> Output:[result]
    end

    tarea2: action Input2:[value], config:[option] ==>
    end

end
```

Fig. 2.17. Estructura de un actor completo

Una vez explicada la estructura general de un actor en RVC – CAL, pasamos a hacer un repaso de los principales recursos presentes en este lenguaje:

– En primer lugar, tenemos que distinguir entre constantes y variables y, dentro de cada uno de estos grupos, entre las que son globales (utilizables durante todo el actor) y las que son propias de una tarea (utilizables únicamente dentro de dicha tarea). La única diferencia existente a la hora de declarar una constante o una variable reside en el elemento con el que se realiza la asignación. Concretamente, las constantes se asignan mediante el elemento "=" – signo de igualdad – y las variables con "**:=**" – dos puntos seguidos del signo de igualdad–. Por otro lado, para distinguir entre variable global y variable propia de una tarea, la diferencia reside en el lugar donde se realiza la declaración. La instanciación global se realiza entre la cabecera del actor y la primera de las tareas pero, al realizar una instanciación propia de una tarea, hay que introducir el término *var*. Todo esto se observa claramente en la figura 2.18.

```
actor Actor() int Input1, double Input2, bool config ==> float Output:

    int(size = 32) constante_global = 5;
    int(size = 32) variable_global := 5;

    tarea: action Input2:[value], config:[option] ==>
    var
        int(size = 32) constante_propia = 5,
        int(size = 32) variable_propia := 5
    do

    end

end
```

Fig. 2.18. Instanciación de constantes y variables

– A continuación, tal y como se observa en la figura 2.19 (Arriba) existen, como en la mayoría de lenguajes habituales, unos elementos de condición. La principal diferencia con respecto a lenguajes como C es que esta comparación – sentencias *if-elsif-else* – se realizan con el símbolo "=" – signo de igualdad – y no con "= =" – dos signos de igualdad seguidos – como es habitual. Como caso característico de este lenguaje, en la figura 2.19 (Abajo) observamos un ejemplo en el que la condición está impuesta al inicio de una acción, esto es, si la condición dentro del conjunto *guard-end* no se cumple, la acción a la que está asignada no se ejecutará.

```
if(variable = 5)then
    //Condición 1
elsif (variable =7) then
    //Condición 2
else
    //Resto
end
```

```
tarea: action Input2:[value], config:[option] ==>
guard
    variable = 5 and variable2 = 7
do
//Cuerpo de la tarea
end
```

Fig. 2.19. Condición estándar (Arriba). Condición de ejecución de tarea (Abajo)

– Otro de los elementos clásicos presente en la mayoría de los lenguajes es el bucle. En este lenguaje existen dos formas de utilizarlo.

  En primer lugar está el bucle *while* común, el cual se ejecuta todas las veces necesarias hasta que se deja de cumplir la condición de inicio. Un ejemplo de uso es el que se observa en la figura 2.20.

```
while i < MAX do
    //Código a ejecutar mientras se cumple la acción
end
```

Fig. 2.20. Bucle while

  En segundo lugar, encontramos el también clásico bucle *for* que, en nuestro caso, recibe el nombre de *foreach*. Dicho bucle se repite tantas veces como se indique en la declaración. La sintaxis necesaria para declarar un bucle de este tipo la podemos observar en la figura 2.21.

```
foreach int(size=32) i in 0 .. n do
    //Código a ejecutar n veces
end
```

Fig. 2.21. Bucle foreach

– Por último, los principales operadores están divididos en dos grupos [18]. El primero de ellos está recogido en la tabla 2.6 y se refiere a los operadores unitarios.

Operator	Operand type	Meaning
not	Boolean	logical negation
#	Collection[T]	number of elements
	Map[K, V]	number of mappings
dom	Map[K, V]	domain of a map
rng	Map[K, V]	range of a map
-	Number	arithmetic negation

Tabla 2.6. Tabla de operadores unitarios [18]

  El segundo de ellos está recogido en la tabla 2.7 y se refiere a los operadores binarios.

P	Operator	Operand 1	Operand 2	Meaning
1	and	Boolean	Boolean	logical conjunction
	or	Boolean	Boolean	logical disjunction
2	=	Object	Object	equality
	!=	Object	Object	inequality
	<	Number	Number	less than
		Set[T]	Set[T]	
		String	String	
		Character	Character	
	<=	analogous to <		less then or equal
	>	analogous to <		greater than
	>=	analogous to <		greater than or equal
3	in	T	Collection[T]	membership
4	+	Number	Number	addition
		Set[T]	Set[T]	union
		List[T]	List[T]	concatenation
		Map[K, V]	Map[K, V]	map union
	-	Number	Number	difference
		Set[T]	Set[T]	set difference
5	div	Number	Number	integral division
	mod	Number	Number	modulo
	*	Number	Number	multiplication
		Set[T]	Set	intersection
	/	Number	Number	division
6	^	Number	Number	exponentiation

Tabla 2.7. Tabla de operadores binarios [18]

Para finalizar la explicación de la sintaxis propia de RVC – CAL, únicamente queda por analizar la forma de programar la ejecución de las acciones que componen un mismo actor.

Esta organización se puede realizar de tres formas distintas:

- La primera de ellas es utilizar las sentencias *guard* explicadas en este mismo documento. Un ejemplo de ello puede verse en la figura 2.22 (A).
- La segunda es utilizar un elemento denominado *priority* que fija un orden de ejecución concreto, asignando a cada tarea un nivel de prioridad mayor que el que tienen las que se ejecutan tras ella. Esto se observa en la figura 2.22 (B).
- La última forma de realizar esta asignación es utilizar otro elemento propio del lenguaje RVC – CAL: el *scheduler* – o planificador de tareas. Este organizador funciona como un autómata, indicando, acción por acción, la tarea que se llevará a cabo tras ella. Un ejemplo de esta organización también puede observarse en la figura 2.22 (C).

A
```
tarea1: action Input2:[value], config:[option] ==>
guard
    tarea1_done = false
do
    //Código de la tarea 1
    tarea1_done := true;
end
tarea2: action Input1:[value] ==> Output:[result]
guard
    tarea1_done = true and tarea2_done = false
do
    //Código de la tarea 2
    tarea2_done := true;
end
```

B
```
priority
    tarea1 > tarea2;
end
```

C
```
schedule fsm s_first:
    s_first (tarea1) --> s_second;
    s_second (tarea2) --> s_first;
end
```

Fig. 2.22. Formas de organizar las tareas: mediante el uso del guard (A), mediante la utilización del comando priority (B) y mediante el uso del scheduler (C).

Tras analizar la sintaxis completa de este lenguaje, en la figura 2.23 se aporta un ejemplo de actor completo (obtenido de la página oficial de ORCC) con la mayoría de los elementos utilizados a lo largo de este estudio.

```
package org.mpeg4.part10.cbp.selectMacroblock;

import org.mpeg4.part10.cbp.MacroBlockInfo.BLOCK_TYPE_INTRA_MAX;

actor SelectMb()
    uint(size=6) MbType,
    uint(size=8) MbFromIntra,
    uint(size=8) MbFromInter
        ==>
    uint(size=8)  MbOut
        :
    uint(size=6) mbType;

    getMbType: action MbType:[valMbType] ==>
    do
        mbType := valMbType;
    end

    forwardMb.intra: action MbFromIntra:[x] repeat 64 ==>
        MbOut:[x] repeat 64
    guard
        mbType <= BLOCK_TYPE_INTRA_MAX
    end

    forwardMb.inter: action MbFromInter:[x] repeat 64 ==>
        MbOut:[x] repeat 64
    guard
        mbType > BLOCK_TYPE_INTRA_MAX
    end

    schedule fsm GetMbType:
        GetMbType (getMbType  ) --> ForwardMb;
        ForwardMb (forwardMb  ) --> GetMbType;
    end
end
```

Fig. 2.23. Ejemplo de actor completo
(Fuente: http://orcc.sourceforge.net/)

Una vez realizado el análisis de este lenguaje, se recogen las principales ventajas e inconvenientes de la utilización del mismo durante el desarrollo de este Proyecto Fin de Grado.

Ventajas

- Implementación de código por actores.
- Comunicación sencilla entre actores.
- Posibilidad de realizar paralelización (mapeo) directamente desde ORCC.
- Generación del código realizado en RVC – CAL en distintos lenguajes, mejorando la interoperabilidad entre plataformas.
- Si cada etapa del análisis hiperespectral se implementa como un actor independiente, este lenguaje aporta facilidad para variar el algoritmo escogido en cada etapa, sin afectar al resto de la cadena.
- Capacidad de importación de librerías externas en un lenguaje compatible con el generado.

Inconvenientes

- Escasez de librerías propias.
- Lenguaje en desarrollo (abundancia de nuevas versiones).
- Escasez de recursos propios del lenguaje: las distintas operaciones admiten únicamente tipos de datos muy concretos.
- No genera ejecutables directamente si no que, a partir de una descripción implementada en RVC – CAL, genera código en otros lenguajes tales como C, C++, Java, etc.

2.6. PAPI

Para concluir el capítulo del estudio del estado del arte, se realizará el análisis del último de los términos indicados al principio del mismo: la herramienta software de análisis PAPI[7].

La principal aplicación de esta herramienta medir el rendimiento de una aplicación dentro de una arquitectura concreta. Para ello, PAPI permite al usuario acceder a una serie de contadores hardware presentes en la mayoría de los procesadores modernos. Utilizando estos contadores PAPI forma sets de registros en los que se recogen lo que se denomina como *Events* – activación de una o varias señales específicas del procesador que indican la ocurrencia de sucesos importantes dentro del mismo.

Gracias al estudio de la activación de estas señales y la consiguiente monitorización de los registros, se puede establecer una relación entre la eficiencia del mapeo del programa – reparto del mismo dentro de la arquitectura o plataforma utilizada – y la estructura del programa en sí. Una vez establecida esta relación, la información recopilada puede analizarse desde distintos puntos de vista, los más comunes son: optimización de la compilación, testeo de la aplicación, bancos de pruebas, monitorización y modelado del rendimiento – en la figura 2.24 se observa un ejemplo de monitorización en el que la herramienta PAPI es capaz de localizar un cuello de botella en cuanto a accesos a memoria caché fallidos se refiere.

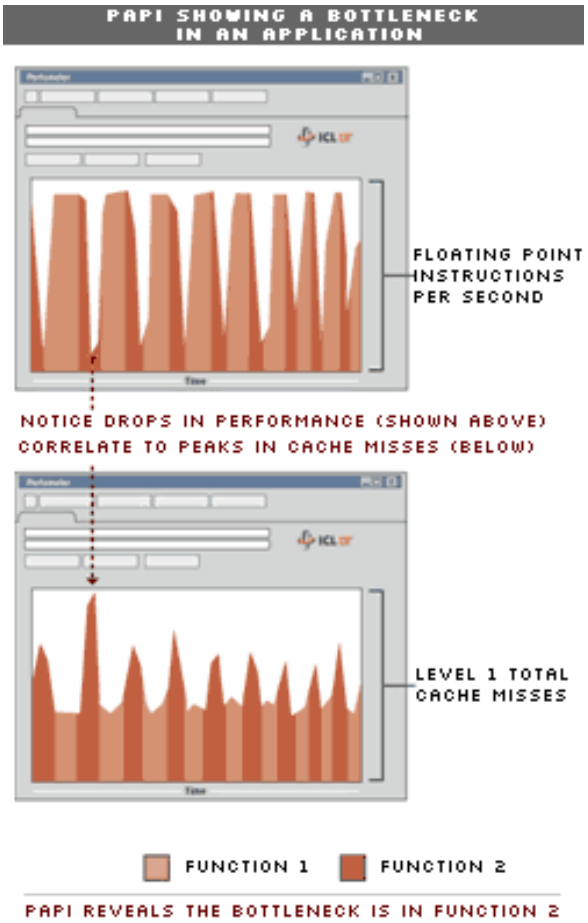


Fig. 2 24. Detección de un cuello de botella usando PAPI (Fuente: http://icl.cs.utk.edu/papi/overview/index.html)

[7] Se puede consultar el manual de usuario de PAPI – *PAPI user guide* – en la siguiente web: http://icl.cs.utk.edu/projects/papi/files/documentation/

Otra de los resultados de la utilización de PAPI es el desarrollo de un nuevo tipo de compilación. El objetivo de esta nueva compilación es, utilizando la información recogida en los registros PAPI de un procesador, detectar y eliminar – o reducir en la medida de lo posible – los cuellos de botella presentes en las aplicaciones de alto rendimiento. La necesidad de evitar estos puntos críticos se hace más acuciante conforme se desarrollan plataformas más potentes – por ejemplo, las plataformas multinúcleo explicadas con anterioridad en este mismo documento.

En cuanto a la organización de PAPI, cabe destacar que existen dos niveles de interfaz diferenciados, una de alto nivel y otra de bajo nivel.

La primera de ellas, la interfaz de alto nivel, es la más simple de las dos y está compuesta únicamente por funcionalidades básicas: iniciar PAPI, leer el set de registros estándar, detener la monitorización, obtener las prestaciones de la plataforma, detener PAPI, etc. De esta manera, si el usuario desea realizar una monitorización del set de eventos predefinido por PAPI, puede llevarla a cabo de utilizando únicamente este nivel.

La segunda, es una interfaz de bajo nivel completamente programable. En este nivel el usuario se genera lo que se conoce como un *EventSet* – o set de eventos – compuesto por los registros que el usuario decida. De esta manera, el usuario puede escoger de entre todos los registros incorporados dentro de la librería PAPI y realizar un estudio personalizado del comportamiento de su aplicación. En la tabla 2.8 se recogen algunos de los registros más importantes junto a la descripción oficial de los mismos.

El conjunto completo de registros está recogido en [23] donde, a su vez, están organizados según su funcionalidad. Del mismo modo, un pequeño manual de usuario de la librería PAPI se encuentra en [24] y, en él, se definen sus funciones diferenciando entre las que pertenecen a la interfaz de alto nivel y las que conforman la interfaz de bajo nivel.

PRESET NAME	DESCRIPTION	PRESET NAME	DESCRIPTION
PAPI_L1_DCM	Level 1 data cache misses	**PAPI_BR_NTK**	Conditional branch instructions not taken
PAPI_L1_ICM	Level 1 instruction cache misses	**PAPI_BR_MSP**	Conditional branch instructions mispredicted
PAPI_L1_TCM	Level 1 total cache misses	**PAPI_BR_PRC**	Conditional branch instructions correctly predicted
PAPI_CA_SNP	Requests for a Snoop	**PAPI_FMA_INS**	FMA instructions completed
PAPI_CA_SHR	Requests for access to shared cache line (SMP)	**PAPI_TOT_IIS**	Total instructions issued
PAPI_CA_CLN	Requests for access to clean cache line (SMP)	**PAPI_TOT_INS**	Total instructions executed
PAPI_CA_INV	Cache Line Invalidation (SMP)	**PAPI_INT_INS**	Integer instructions executed
PAPI_CA_ITV	Cache Line Intervention (SMP)	**PAPI_FP_INS**	Floating point instructions executed
PAPI_L3_LDM	Level 3 load misses	**PAPI_LD_INS**	Load instructions executed

PAPI_L3_STM	Level 3 store misses	**PAPI_SR_INS**	Store instructions executed
PAPI_BRU_IDL	Cycles branch units are idle	**PAPI_BR_INS**	Total branch instructions executed
PAPI_FXU_IDL	Cycles integer units are idle	**PAPI_VEC_INS**	Vector/SIMD instructions executed
PAPI_FPU_IDL	Cycles floating point units are idle	**PAPI_FLOPS**	Floating Point Instructions executed per second
PAPI_LSU_IDL	Cycles load/store units are idle	**PAPI_RES_STL**	Cycles processor is stalled on resource
PAPI_TLB_DM	Data translation lookaside buffer misses	**PAPI_FP_STAL**	Cycles any FP units are stalled
PAPI_TLB_IM	Instruction translation lookaside buffer misses	**PAPI_TOT_CYC**	Total cycles
PAPI_TLB_TL	Total translation lookaside buffer misses	**PAPI_IPS**	Instructions executed per second
PAPI_L1_LDM	Level 1 load misses	**PAPI_LST_INS**	Total load/store instructions executed
PAPI_L1_STM	Level 1 store misses	**PAPI_SYC_INS**	Synchronization instructions executed
PAPI_BTAC_M	Branch target address cache (BTAC) misses	**PAPI_L1_DCH**	L1 data cache hits
PAPI_PRF_DM	Pre-fetch data instruction caused a miss	**PAPI_L1_DCA**	L1 data cache accesses
PAPI_TLB_SD	Translation lookaside buffer shootdowns (SMP)	**PAPI_L1_DCR**	L1 data cache reads
PAPI_CSR_FAL	Failed store conditional instructions	**PAPI_L1_DCW**	L1 data cache writes
PAPI_CSR_SUC	Successful store conditional instructions	**PAPI_L1_ICH**	L1 instruction cache hits
PAPI_CSR_TOT	Total store conditional instructions	**PAPI_L1_ICA**	L1 instruction cache accesses
PAPI_MEM_SCY	Cycles Stalled Waiting for Memory Access	**PAPI_L1_ICR**	L1 instruction cache reads
PAPI_MEM_RCY	Cycles Stalled Waiting for Memory Read	**PAPI_L1_ICW**	L1 instruction cache writes
PAPI_MEM_WCY	Cycles Stalled Waiting for Memory Write	**PAPI_L1_TCH**	L1 total cache hits
PAPI_STL_ICY	Cycles with No Instruction Issue	**PAPI_L1_TCA**	L1 total cache accesses
PAPI_FUL_ICY	Cycles with Maximum Instruction Issue	**PAPI_L1_TCR**	L1 total cache reads
PAPI_STL_CCY	Cycles with No Instruction Completion	**PAPI_L1_TCW**	L1 total cache writes
PAPI_FUL_CCY	Cycles with Maximum Instruction Completion	**PAPI_FML_INS**	Floating Multiply instructions
PAPI_HW_INT	Hardware interrupts	**PAPI_FAD_INS**	Floating Add instructions
PAPI_BR_UCN	Unconditional branch instructions executed	**PAPI_FDV_INS**	Floating Divide instructions
PAPI_BR_CN	Conditional branch instructions executed	**PAPI_FSQ_INS**	Floating Square Root instructions
PAPI_BR_TKN	Conditional branch instructions taken	**PAPI_FNV_INS**	Floating Inverse instructions

Tabla 2.8. Tabla de registros PAPI resumida

Una de las características principales de PAPI, es su interoperabilidad [25-27]. Esta capacidad se debe, principalmente, a que las funciones están desarrolladas de tal manera que utilizan los mismos argumentos independientemente de la plataforma para la que se haya desarrollado.

También cabe destacar que, si utilizamos el set de eventos predeterminado por PAPI, toda la información necesaria está incluida en la cabecera de configuración. Esto implica que, independientemente de la versión de PAPI y de la plataforma utilizada, se podrá acceder a dichos registros sin necesidad de realizar ninguna configuración extra. Además, si se da el caso de que, en el entorno utilizado, no existe uno de estos registros predeterminados, PAPI intenta realizar una estimación del valor que estaría asociado a dicho registro utilizando los demás registros disponibles.

Por el contrario, si el usuario necesita hacer un seguimiento de un set de eventos específico, la interfaz de bajo nivel proporcionada por PAPI otorga al usuario un amplio grado de libertad a la hora de configurar el set de eventos y, si se da el caso de que un evento no existe en la plataforma utilizada, PAPI devuelve un error indicando el motivo del fallo.

Otra forma de dividir PAPI en dos niveles es tener en cuenta la independencia de las distintas funciones respecto a la plataforma utilizada. Podríamos decir que consta de un nivel superior, independiente de la plataforma, que, únicamente, obtiene el valor de los distintos registros indicados a través del nivel inferior. Este nivel inferior, accede al substrato del soporte – ya sea a través del sistema operativo, del kernel o de funciones específicas del procesador – y constituye las relaciones pertinentes entre las señales del procesador y los registros establecidos. Estas relaciones no siempre son posibles de establecer por lo que, dependiendo de la cantidad de registros que se pueda definir, el nivel superior estará más o menos completo. Si una señal no se puede relacionar directamente con un registro, tal y como se ha explicado anteriormente, si es posible, se estimará su valor.

Asimismo, cabe destacar dos funcionalidades que incorpora la herramienta PAPI: la capacidad de multiplexar la monitorización de los distintos registros del procesador y la capacidad de ejecutar de PAPI en diversos hilos [24].

La primera de ellas, se desarrolló debido a la limitación existente en los procesadores modernos a la hora de acceder a sus registros de manera simultánea. Esto tenía como consecuencia que para monitorizar un conjunto de eventos había que lanzar la aplicación varias veces y, por lo tanto, este análisis podía extenderse durante horas. Para solventar este problema, los desarrolladores de PAPI decidieron dividir el uso de los distintos registros en porciones de tiempo, utilizando dichos registros para monitorizar a la vez varios eventos distintos. Esto resuelve el problema de monitorizar un mayor número de registros de una sola vez pero, a su vez, añade un problema de precisión al realizar la medida – la ocurrencia de un evento solo se detecta cuando tiene un registro asociado.

Por otro lado, la funcionalidad de ejecutar PAPI en varios hilos se desarrolló debido al auge de las aplicaciones que utilizan varios hilos – y varios procesadores – durante su ejecución. Estos programas suelen basarse en la utilización de plataformas con varios procesadores o, simplemente, en la necesidad de paralelizar la ejecución de tareas en un mismo núcleo. Gracias a esta característica de PAPI, un usuario es capaz de monitorizar cada hilo por separado y detectar de manera más fiable los distintos cuellos de botella debidos a la paralelización de una aplicación. Cabe destacar que, si cada hilo corresponde a las tareas asociadas a cada procesador, utilizando este tipo de monitorización seríamos capaces de analizar el consumo de recursos que tiene cada procesador por separado.

En sus últimas versiones, PAPI ha introducido una nueva funcionalidad: activar una señal cada vez que se produce un *overflow* – o desbordamiento – de uno de los registros PAPI. Esta función trabaja a nivel asíncrono – respecto al reloj del sistema – ya que, cada registro PAPI, puede aumentar su valor debido a la ocurrencia de señales independientes que pueden activarse en un mismo ciclo de reloj.

Como ya hemos venido adelantando a lo largo de la explicación, la principal aplicación de PAPI es la medida de consumo de recursos, energía y potencia – en resumen, del rendimiento – que tiene una plataforma cuando se ejecuta sobre ella una aplicación concreta [28-29]. En [29] se realiza la monitorización del procesador de Intel *SandyBridge EP* y, en él, se mide en el tiempo el número total de instrucciones – en rojo –, la energía consumida por el *chip0* y la memoria *DRAM* asociada al *chip0* – en verde – y la energía consumida por el *chip1* – en azul – arrojando el resultado que se observa en la figura 2.25.

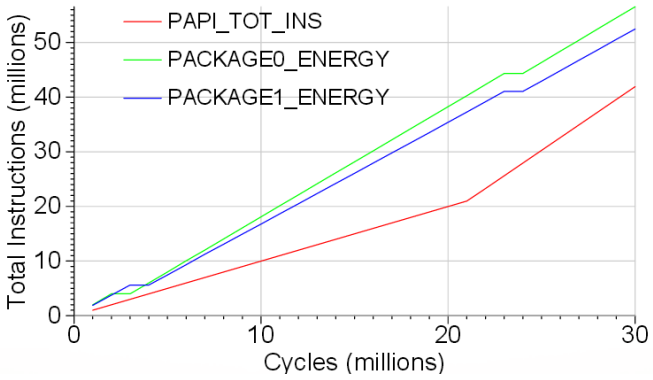


Fig. 2.25. Resultados de PAPI monitorizando el procesador SandyBridge EP

CAPÍTULO 3. DESCRIPCIÓN DE LA SOLUCIÓN DESARROLLADA

A lo largo del presente capítulo se realizará una explicación detallada del desarrollo de una cadena de análisis de imágenes hiperespectrales implementada en RVC – CAL. Este capítulo se encuentra dividido en tres apartados: Justificación de la solución elegida, contenido de la librería y procedimiento de desarrollo.

3.1. Justificación de la solución elegida

Como se mencionó en el capítulo 1 de este mismo documento, el objetivo principal de este Proyecto Fin de Grado es implementar un método de análisis de imágenes hiperespectrales capaz de alcanzar, en la medida de lo posible, el tiempo real. Para ello, tal y como reveló el estudio del estado del arte realizado en el capítulo 2, es necesario analizar una gran cantidad de datos en poco tiempo, o lo que es lo mismo, alcanzar una elevada velocidad de procesamiento. En definitiva, la conclusión que se alcanzó tras realizar dicho estudio fue la necesidad de desarrollar una cadena de procesado basada en plataformas multinúcleo.

Una vez se tomó la decisión de implementar una cadena de procesado que se apoyase en varios núcleos, se optó por utilizar el lenguaje de programación conocido como RVC – CAL. Las principales ventajas de la utilización de dicho lenguaje se resumen a continuación:

- *Implementación de código por actores:* debido a que RVC – CAL es un lenguaje basado en flujo de datos, se puede variar el contenido de un actor sin tener que modificar toda la red. Esto facilita la variación del algoritmo seleccionado para una etapa de análisis concreta sin necesidad de modificar el resto.
- *Facilidad para establecer la comunicación entre actores:* esto nos proporciona la capacidad de utilizar varios núcleos al mismo tiempo. Gracias a ello, únicamente quedaría planificar la granularidad con la que se realiza el reparto de actores.
- *Capacidad de generar automáticamente código en diversos lenguajes:* esta funcionalidad nos aporta una gran interoperabilidad entre plataformas. Cabe destacar que uno de los lenguajes disponibles es C – lenguaje seleccionado para implementar la librería.
- *Posibilidad de incorporar librerías externas:* esto permite añadir librerías (en un lenguaje compatible con el lenguaje generado), facilitando de esta manera el desarrollo de código dentro del propio RVC – CAL.

Por el contrario, la utilización de este lenguaje implica lidiar con una serie de desventajas presentes en el mismo. Aunque ya se han señalado en el capítulo 2, a modo de resumen, se mencionan a continuación:

- *Librerías propias del lenguaje limitadas:* ya que este lenguaje está especialmente orientado a desarrollar decodificadores de vídeo y audio, las librerías que incorpora están directamente relacionadas con estas aplicaciones. Por ello, únicamente aporta librerías de lectura/escritura de imágenes, análisis y transmisión de fotogramas, etc.
- *Escasez de recursos propios del lenguaje:* el lenguaje RVC – CAL está pensado para realizar operaciones a nivel de dato único, entendiendo como tal cada uno de los elementos que componen un *array*, una matriz o, simplemente, un número entero. Esto reduce drásticamente la capacidad de realizar operaciones con imágenes completas ya que éstas, normalmente, se almacenan en forma de matriz.

En nuestro caso, la escasez de medios para trabajar con matrices supone una gran limitación, ya que las imágenes hiperespectrales están almacenadas formando cubos de datos (las bandas de la imagen se organizan como planos contiguos formando un cubo), los cuales, en definitiva, son matrices de tres dimensiones – plano espacial y profundidad espectral –. A su vez, en la literatura analizada sobre algoritmos de análisis de imágenes hiperespectrales, de cara a facilitar el trabajo con estos cubos de datos, se reduce el número de dimensiones de la imagen a dos, manteniendo una de ellas como la propia dimensión espectral del cubo de datos y combinando las dos dimensiones espaciales.

Teniendo en cuenta este problema y con el objetivo de aportar la capacidad de trabajar con matrices utilizando RVC – CAL, se decidió implementar una librería externa que recogiera todas las funciones necesarias para desarrollar los algoritmos pertenecientes a las diferentes fases del análisis de una imagen hiperespectral.

3.2. Contenido de la librería

Una vez resaltada la complejidad de desarrollar algoritmos de procesado de imágenes hiperespectrales en RVC – CAL y, por tanto, la importancia de implementar una librería que facilite el desarrollo de dichos algoritmos, es necesario definir el tipo de funciones que se incluirán en ella.

Para definir el contenido de la librería, se realizará un estudio de los algoritmos existentes en cada etapa de procesado, estudiando su complejidad, su velocidad de procesamiento, los tipos de operaciones de las que se componen y analizando las diferencias entre ellos. Tras esto, se seleccionará uno de cada fase y se codificarán las funciones que lo componen. A su vez, el algoritmo escogido se implementará utilizando las funciones desarrolladas y, de esta manera, se utilizará al mismo tiempo como soporte de desarrollo y como test de fiabilidad. Un ejemplo gráfico de la cadena de procesado completa se puede observar en la figura 3.1.

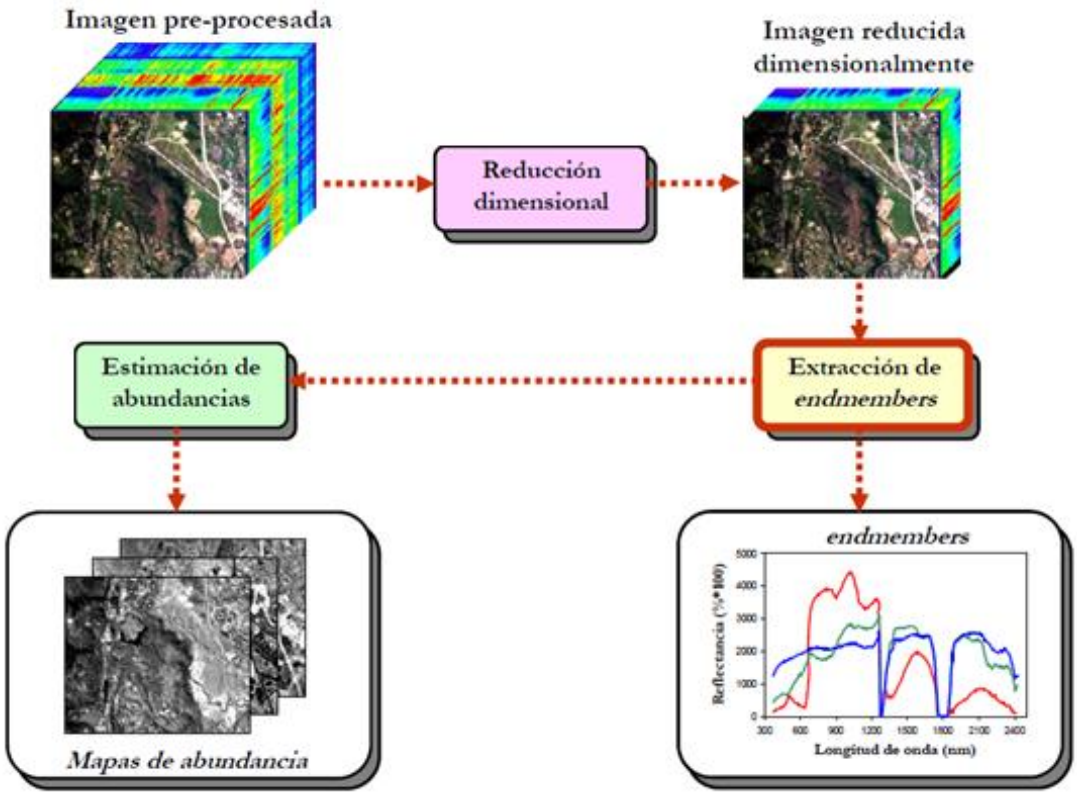


Fig. 3.1. Cadena de procesado de imágenes hiperespectrales (Fuente: http://www.umbc.edu/)

A su vez, para probar la utilidad de implementar dicha librería, se ha decidido dividir su desarrollo en dos partes independientes. Por ello, en este documento únicamente se analizan las etapas de *reducción dimensional* y *extracción de* endmembers; las otras dos fases – *estimación del número de* endmembers y *estimación de abundancias* – serán analizadas y desarrolladas por Raquel Lazcano en [1]. El objetivo de esta división es realizar una comparativa de las funciones obtenidas en ambos casos y comprobar el grado de similitud existente entre ellas. Si ambas partes tienen funciones similares, concluiremos que la librería es lo suficientemente genérica como para desarrollar algoritmos de análisis hiperespectral y que no se basa en funciones específicas propias de un algoritmo – o etapa – concreto.

Reducción dimensional

Durante el análisis de esta fase se realizará una comparación de los algoritmos PCA (de sus siglas en inglés, *Principal Component analysis*), MNF (del inglés, *Minimum Noise Fraction*) y SPP (de sus siglas, *Spatial Pre-Processing*), ya que son los más comunes y más utilizados dentro de la literatura existente actualmente.

En primer lugar, el algoritmo PCA se basa en reducir la imagen de tal manera que se obtenga como resultado la información relevante presente en la imagen. Para ello, según se explica en [30] y en [7], se realiza una descomposición en valores singulares o SVD (del inglés *Singular Values Decomposition),* obteniendo de esta forma los autovectores de la imagen original. Estos autovectores están ordenados en orden decreciente de cantidad de información espectral útil, esto es, el primer autovector contiene la mayor parte de la información espectral de la imagen – eliminando ruido e información redundante – y el último, por ejemplo, contiene únicamente ruido. En el caso ideal, los autovectores obtenidos son ortogonales entre sí y, si esto se cumple, la información contenida en las primeras bandas de la imagen de salida es información útil en su totalidad. Cabe mencionar que este algoritmo únicamente realiza una reducción dimensional de la componente espectral de la imagen original.

Un ejemplo del funcionamiento del algoritmo PCA se observa en la figura 3.2. En este caso se ha realizado una reducción PCA a veinte bandas – en la figura 3.2, a modo ilustrativo, únicamente se muestras las cinco primeras y las cinco últimas –. Podemos observar que las cinco primeras bandas contienen información nítida, por el contrario, observamos que las cinco últimas no aportan información y están compuestas casi en su totalidad por ruido.

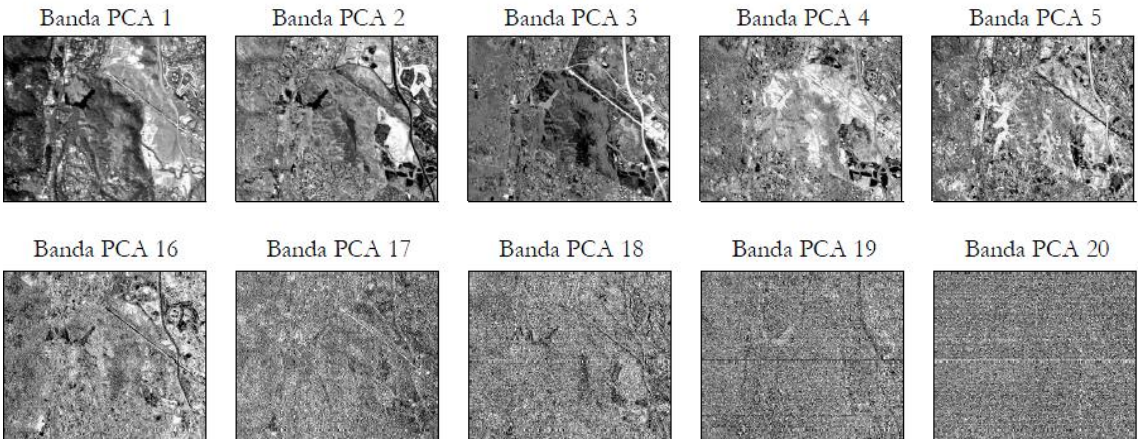


Fig. 3.2. Aplicación del algoritmo PCA sobre una imagen hiperespectral [31]

En segundo lugar analizaremos el algoritmo MNF. Este algoritmo es una versión mejorada del PCA – por lo que también se aplica una reducción de la componente espectral de la imagen – y se basa en los siguientes tres pasos:

- Para empezar, se aplica a la imagen original el algoritmo PCA.
- A continuación, se estiman tanto la matriz de covarianza de la señal – la parte que contiene información de la imagen original – como la matriz de covarianza del ruido.
- Por último, se obtienen una serie de bandas, las cuales estarán ordenadas de mayor a menor relación señal-ruido con respecto a la imagen original.

Por tanto, como observamos en la figura 3.3, en la cual se muestra una imagen hiperespectral a la que se le ha aplicado el algoritmo MNF para veinte bandas – aunque en la imagen únicamente se aporten las cinco primeras y las cinco últimas –, este algoritmo nos aportará imágenes totalmente nítidas en las primeras bandas e imágenes en las que únicamente se observa ruido en las últimas. Esta nitidez se corresponde con información útil obtenida de la imagen original.

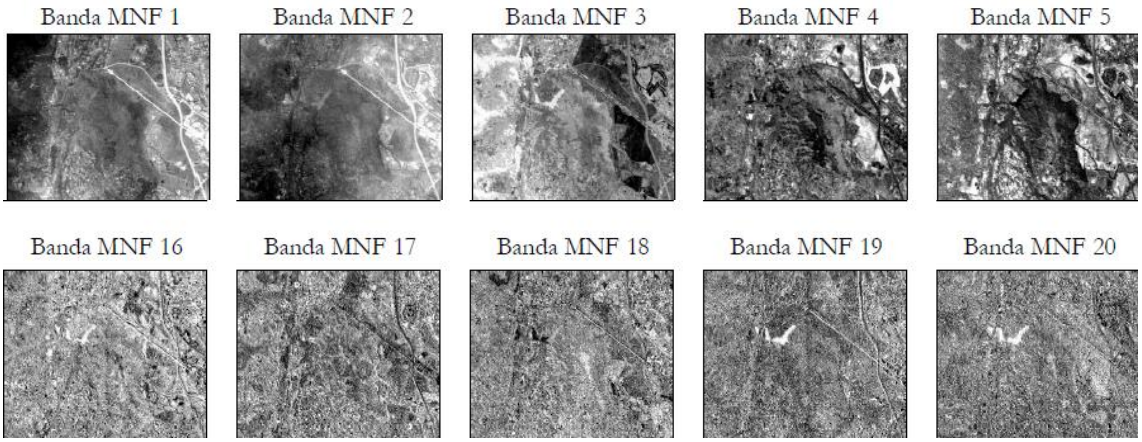


Fig. 3.3. Aplicación del algoritmo MNF sobre una imagen hiperespectral [31]

Por último, el algoritmo SPP tiene un funcionamiento completamente distinto a los dos explicados anteriormente. Al contrario que el PCA y el MNF, el algoritmo SPP trata la imagen de manera espacial. Toma dicha imagen, píxel a píxel, y la transforma, basándose en el grado de similitud que tiene la firma espectral de un pixel con respecto a los que tiene alrededor. En concreto, el algoritmo SPP consta de los siguientes pasos:

- En primer lugar, se define el centroide de la imagen. Este punto es la media de las bandas espectrales de todos los píxeles de la imagen.
- Posteriormente, se calcula para cada píxel un número escalar que relaciona el grado de similitud de la firma espectral de dicho píxel con todos los de su alrededor.
- Por último, se desplazan los píxeles, acercando al centroide de la imagen los que tienen un grado de similitud menor con sus vecinos, y manteniendo alejados los conjuntos de píxeles que comparten firma espectral.

De esta manera, el cálculo de *endmembers* posterior será más sencillo, debido a que los píxeles más alejados del centroide serán aquellos que tengan una firma espectral similar y, en consecuencia, formen un elemento de la imagen concreto.

Debido a que uno de los principales objetivos nombrados en el capítulo 1 de este mismo documento es la consecución del tiempo real, uno de los aspectos fundamentales dentro de este Proyecto Fin de Grado es la velocidad de procesamiento. Teniendo en cuenta lo expuesto anteriormente, observamos que el algoritmo SPP es bastante lento comparado con los otros dos. Esto se debe a que hace un análisis píxel a píxel de la imagen y, en consecuencia, tiene en cuenta tanto componente espacial como componente espectral, ralentizando de este modo el procesado de la imagen. A su vez, el algoritmo MNF realiza una reducción dimensional PCA durante su ejecución, por lo que, obviamente, será más lento que el propio PCA.

Por otro lado, la complejidad y el tipo de operación que componen estos algoritmos difieren bastante de unos a otros. El SPP, por su parte, se basa en realizar una media de la firma espectral de todos los píxeles de la imagen para, posteriormente, comparar de nuevo cada firma con la media de dichos píxeles – a la que denominan centroide –. Por su parte, tanto PCA como MNF realizan una descomposición SVD e incluyen numerosas operaciones con matrices, como por ejemplo, calcular autovectores.

En conclusión, el algoritmo SPP es descartado por ser lento y no compartir funciones con los otros dos – cabe recordar en este punto el objetivo principal de este Proyecto Fin de Grado: implementar una librería que recoja el máximo de funciones comunes a todos los algoritmos de procesado espectral –. A su vez, entre el algoritmo PCA y el MNF, nos decantamos por desarrollar el PCA por ser más rápido, ya que otro de los objetivos del proyecto es implementar una cadena de procesado espectral en tiempo real y, como vimos en el capítulo 2 de este mismo documento, el algoritmo PCA puede llegar a ejecutarse en menos de 200ms. A modo de recordatorio, se aporta la tabla 3.1 – esta tabla se incluyó en el análisis de tiempos realizado en el capítulo 2 –. En ella se observa que el algoritmo PCA puede ejecutarse si se realiza una paralelización óptima para, por ejemplo, una GPU de la serie GTX, en un tiempo de 0.132s.

	LOAD	VD	PCA	N-FINDR	UCLS	SAVE	TOTAL	SPEEDUP
Serie gcc	0.186	5.555	5.401	0.883	0.313	0.008	12.346	
Tesla gcc	0.202	0.411	0.147	0.112	0.054	0.008	0.934	13.219
GTX gcc	0.204	0.423	0.137	0.107	0.045	0.010	0.926	13.331
Serie icc	0.170	3.940	2.253	0.847	0.251	0.008	7.470	
Tesla icc	0.183	0.420	0.149	0.115	0.054	0.012	0.933	8.007
GTX icc	0.194	0.441	0.132	0.094	0.041	0.012	0.914	8.174

Tabla 3.1. Tiempo de procesamiento – caso más rápido [7]

Extracción de *endmembers*

A continuación, para realizar el análisis de esta fase, se estudiarán los siguientes algoritmos: IEA (*Iterative Error Analysis)*, N-FINDR (*N-Finder algorithm*), OSP (*Orthogonal Subspace Projection*), VCA (*Vertex Component Analysis*), AMEE (*Automatic Morphological Endmember Extraction*) y SSEE (*Spatial-Spectral Endmember Extraction*). Cabe destacar que el análisis de estos algoritmos se ha realizado en profundidad en el Proyecto Fin de Máster de Luis Ignacio Jiménez [32]. Por ello, en este documento únicamente se recogerán las definiciones de dichos algoritmos y un resumen de la comparativa de los mismos realizada en [32]. De esta manera, se seguirá el mismo esquema que en el análisis de la fase de *reducción dimensional*.

En primer lugar, analizaremos el algoritmo IEA. Un punto fuerte de este algoritmo es que integra tanto la fase de *extracción de* endmembers como la de *estimación de abundancias*. Esto se debe al funcionamiento característico de este algoritmo:

- El primer paso es calcular lo que se conoce dentro de este algoritmo como *endmember inicial*. Esto se realiza obteniendo la media espectral de todos los píxeles presentes en la imagen original.
- Tras esto, se calcula la abundancia del *endmember* calculado para cada píxel de la imagen.
- A continuación, se calcula la raíz del error cuadrático medio (RMSE), componente a componente, entre la imagen original y la abundancia calculada.
- Posteriormente, se toma el píxel con mayor RMSE de los calculados y se añade al conjunto de los *endmembers* calculados.
- En este momento, se inicia un proceso iterativo que recoge los tres pasos anteriores; esto es, para el conjunto de *endmembers* calculado en cada iteración, se calcula la abundancia de la imagen y su consiguiente RMSE, añadiendo al conjunto de *endmembers* el píxel que mayor RMSE tenga de todos.
- Este proceso iterativo se realiza tantas veces como *endmembers* se quieran calcular y, tras finalizar las iteraciones, se obtiene el conjunto de *endmembers* y sus respectivos mapas de abundancias (correspondientes a la fase de *estimación de abundancias*).

Por otro lado, nos encontramos con el algoritmo N-FINDR. Este algoritmo trabaja con una imagen reducida espectralmente a tantas bandas como *endmembers* se pretende calcular – esto implica haber utilizado anteriormente el PCA o el MNF en la etapa de *reducción dimensional* –. El objetivo de dicho algoritmo es construir el mayor volumen posible – denominado comúnmente *simplex* (entendiendo como tal aquel que contenga el mayor número de píxeles de la imagen) –, utilizando por vértices los *endmembers* que se pretendan calcular. Para ello se siguen los siguientes pasos:

- En primer lugar, como ya se ha dicho anteriormente, se realiza una reducción dimensional a tantas bandas como *endmembers* se quiera calcular.
- Tras esto, se selecciona un conjunto aleatorio de píxeles que se tomarán como *endmembers iniciales*. Esta selección se irá refinando de manera iterativa a lo largo de la ejecución del algoritmo.
- A continuación, se selecciona cada píxel de la imagen y se va intercambiando por cada uno de los *endmembers* seleccionados. Con este nuevo conjunto, se calcula de nuevo el volumen resultante.
- Si el volumen es mayor que el que había anteriormente al cambio, se utiliza ese nuevo punto como *endmember*; si no, se deshace el cambio realizado.
- Este proceso se realiza iterativamente hasta que se han probado todos los píxeles de la imagen. Una vez probados todos, los píxeles que se hayan establecido como vértices del volumen resultante, se consideran *endmembers*.

Este algoritmo presupone que un aumento en el volumen calculado implica unos *endmembers* más exactos y de mayor precisión pero, como se observa en la figura 3.4, esto no siempre se cumple. En esta figura se observa que, para un volumen mayor, se encuentran menos puntos dentro del mismo que para un volumen menor.

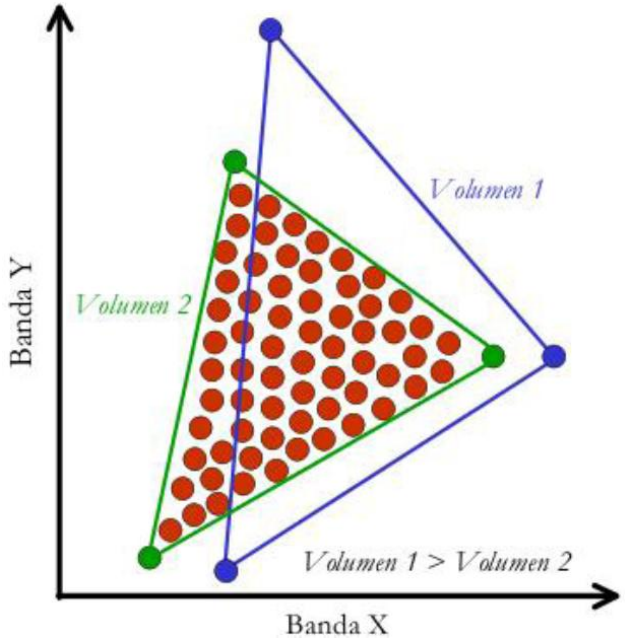


Fig. 3.4. Comparativa de dos volúmenes propios del algoritmo N-FINDR [32]

El siguiente algoritmo a analizar es el OSP. Este algoritmo iterativo se basa en la idea de las proyecciones ortogonales para encontrar las distintas firmas espectrales – o *endmembers* –. Los pasos que sigue este algoritmo son los siguientes:

- En primer lugar, busca el primer *endmember* como aquel píxel en el que se cumple que su producto vectorial por el traspuesto de dicho pixel es máximo.
- Tras esto, aplica un operador de proyección ortogonal $P_U = I - U(U^T U)^{-1}U^T$ y, en él sustituimos I por la matriz identidad y U por el conjunto de *endmembers* que tengamos actualmente.
- A continuación, el algoritmo busca el siguiente *endmember* como aquél que presenta una mayor ortogonalidad con respecto a la proyección del conjunto de *endmembers* que tenemos actualmente calculado.
- Añadimos el nuevo *endmember* al conjunto final.
- Se repiten los pasos 2-4 tantas veces como *endmembers* queramos que formen el conjunto resultante.

A continuación, el algoritmo a analizar es el VCA. Este algoritmo se basa en dos ideas fundamentales:

- Igual que en el algoritmo N-FINDR explicado anteriormente, los *endmembers* calculados serán considerados los vértices de un *simplex* con tantos extremos como *endmembers* se quieran calcular.
- La transformación afín de un *simplex* tiene como resultado otro *simplex*.

Las bases de este algoritmo son una combinación de los algoritmos N-FINDR y OSP. Del primero comparte la idea de que existen píxeles puros en la imagen, mientras que del OSP comparte la idea de hacer proyecciones ortogonales iterativas hacia el subespacio que componen los *endmembers* calculados hasta ese momento, siendo el nuevo *endmember* el extremo de dicha proyección.

Una vez estudiados los algoritmos que únicamente tienen en cuenta la información espectral de la imagen original, pasamos a definir los algoritmos AMEE y SSEE, los cuales analizan la imagen tanto espectral como espacialmente.

El algoritmo AMEE considera la información espectral y espacial de manera simultánea. En este método se utilizan dos operaciones morfológicas extendidas: la dilatación y la erosión. Durante la dilatación se expanden las zonas espectralmente puras – atendiendo a la forma y al tamaño de dicha zona – y, por el contrario, durante la erosión se reducen las zonas impuras, atendiendo a los mismos parámetros de tamaño y forma. El algoritmo AMEE se divide en cuatro pasos:

- En primer lugar, se analiza la imagen y se aplican las operaciones morfológicas extendidas, explicadas anteriormente. La dilatación se aplica para la zona más pura, mientras que, para la zona menos pura, se aplica la erosión. En este algoritmo se presupone que los píxeles están formados por más de un elemento, por lo que un píxel se considera puro si contiene un elevado porcentaje de un único elemento.
- A continuación se localizan píxeles puros, utilizando la información obtenida en el apartado anterior.
- Existen dos opciones en este paso: o bien se aplica un proceso de crecimiento de regiones para obtener una lista de *endmembers* concreta, o bien se aplica el algoritmo OSP con la misma finalidad.
- En el paso final, si se ha utilizado la primera opción del apartado anterior, este paso tiene por objetivo eliminar *endmembers* redundantes.

Por último, el único algoritmo que queda por analizar es el SSEE. Este algoritmo, al contrario que el AMEE, considera la información espectral y espacial por separado. Su ejecución sigue una secuencia de cuatro pasos:

- En primer lugar, se realiza una descomposición SVD de la imagen con el objetivo de obtener una serie de autovectores. Esta descomposición se realiza mediante submatrices cuadradas que, en conjunto, tienen que englobar toda la imagen – dichas submatrices no pueden solaparse unas con otras –. Si la imagen es cuadrada, se puede realizar la descomposición SVD directamente.
- A continuación, se proyectan todos los píxeles sobre cada uno de los autovectores obtenidos en el paso anterior. El máximo y el mínimo de cada proyección pasa a ser un posible *endmember* del conjunto final.
- Posteriormente, se aumenta el número de píxeles candidatos. Este aumento se realiza analizando los píxeles vecinos a los candidatos ya existentes y escogiendo de entre ellos los que sean espectralmente similares. Una vez aumentado el conjunto de píxeles, se examinan aquellos que sean cercanos espacialmente, se realiza una media de su firma espectral y se asigna dicha firma al que más se aproxime al valor obtenido. Este proceso se repite iterativamente, de tal forma que los píxeles que sean espectralmente similares y estén cerca los unos de los otros convergerán a la media de todos ellos.
- Por último, exactamente como en el algoritmo anterior, existen dos caminos a seguir para obtener los *endmembers*. Por un lado, existe la posibilidad de aplicar el algoritmo OSP y, por el otro, analizar los píxeles candidatos, ordenarlos por la distancia existente entre ellos y, de esta manera, agruparlos por clases y extraer los *endmembers*.

Una vez hemos descrito el funcionamiento básico de cada uno de los algoritmos de la etapa de *extracción de* endmembers, a continuación se analizan sus características, con la finalidad de decantarnos por desarrollar uno de ellos. Cabe recordar que este análisis se ha realizado en profundidad en [32] y, en consecuencia, aquí se recoge un breve resumen de las conclusiones más relevantes de dicho análisis.

En primer lugar, los resultados en cuanto a precisión para una misma imagen se encuentran resumidos en la tabla 3.2. En ella, observamos que los algoritmos que nos aportan unos *endmembers* más exactos son OSP y VCA, mientras que, por el contrario, AMEE y N-FINDR son los menos precisos de los algoritmos analizados. Los valores cercanos a cero indican coincidencia y los más alejados indican mayor discrepancia con respecto al resultado esperado.

	1	2	3	4	5	6	7	8	9	SAD
IEA	0,002	0,002	0,007	0,000	0,288	0,001	0,002	0,009	0,000	0,034
NFINDR	0,083	0,013	0,151	0,082	0,160	0,004	0,002	0,093	0,028	0,068
OSP	0,002	0,002	0,005	0,000	0,007	0,001	0,002	0,009	0,000	0,003
VCA	0,002	0,002	0,010	0,000	0,007	0,001	0,002	0,007	0,000	0,003
AMEE	0,104	0,184	0,010	0,001	0,005	0,354	0,002	0,223	0,090	0,108
SSEE	0,002	0,002	0,005	0,000	0,010	0,003	0,002	0,009	0,000	0,004

Tabla 3.2. Comparativa de precisión [32]

Por otro lado, dado que uno de los objetivos principales de este Proyecto Fin de Grado es alcanzar – en la medida de lo posible – el tiempo real, se recoge en la tabla 3.3 un análisis orientativo del tiempo de ejecución de los distintos algoritmos.

	Tiempo de ejecución
OSP	< 10 segundos
IEA	< 10 segundos
VCA	< 10 segundos
AMEE	Aprox. 1 minutos
PCA + N-FINDR	Aprox. 2 minutos
SSEE	> 10 minutos

Tabla 3.3. Comparativa de tiempo de ejecución [32]

Observando estos dos aspectos, se han preseleccionado los algoritmos OSP y VCA, ya que son precisos y rápidos en ejecución. Por ello, se ha analizado en detalle la complejidad de programación de ambos y, en conclusión, se ha decidido desarrollar el algoritmo VCA con el fin de completar al máximo posible la librería, ya que, como se ha mencionado en su descripción, comparte características de los algoritmos OSP y N-FINDR.

El algoritmo VCA consta de funciones matemáticas en su mayoría, entre las cuales se incluyen funciones matemáticas básicas, tales como realizar una potencia u obtener el logaritmo de un número, y operaciones con matrices – característica que comparte con el algoritmo escogido para la fase anterior.

3.3. Procedimiento de desarrollo

Una vez fijado el contenido de la librería, se va a explicar el método seguido para completarla y las funciones resultantes del análisis de dos de las cuatro fases del procesado espectral. Además, en este documento se aporta, por un lado, el procedimiento seguido para incorporar las librerías especializadas al lenguaje de programación C, y por otro, una explicación de cómo se ha importado la librería al lenguaje RVC – CAL como una librería externa de funciones nativas desarrolladas en C. Tras esto, se explica cómo se ha procedido para incorporar la herramienta de análisis PAPI – explicada en el capítulo 2 de este mismo documento –, apoyándonos en una aplicación desarrollada por Alejo Iván Arias – miembro del grupo GDEM.

Cabe destacar que en la memoria del Proyecto Fin de Grado de Raquel Lazcano [1] se han explicado varios aspectos importantes para el desarrollo de este proyecto. En concreto, se recoge la unificación de la librería resultante tras implementar cada uno dos etapas de la cadena de procesado, la comunicación entre los distintos actores dentro del lenguaje RVC – CAL, la interfaz de entrada-salida del sistema y, por último, las posibles divisiones de la cadena en un sistema multinúcleo.

Funciones desarrolladas

Tras desarrollar los algoritmos seleccionados previamente, las funciones obtenidas han sido principalmente de carácter matemático, de las cuales una gran mayoría se relaciona con operaciones matriciales. A continuación, se va a detallar la lista de funciones que se ha obtenido tras implementar los algoritmos PCA y VCA.

En un primer momento se desarrolló el algoritmo PCA, obteniendo las funciones descritas a continuación:

- *vector_minus_value*: resta el mismo valor a todos los elementos de un vector.
- *mean_vector*: calcula la media de un vector.
- *matrix_get_column*: devuelve el vector columna que se desee de una matriz.
- *matrix_set_column*: sustituye una columna de una matriz por el vector que se desee.
- *matrix_reduce*: trunca las dimensiones de una matriz.
- *transpose*: devuelve la traspuesta de la matriz de entrada.
- *matrix_mult*: multiplica dos matrices.
- *get_eigenvectMatrix*: devuelve los autovectores de una matriz cuadrada.

Como observamos por las funciones obtenidas, se cumple lo que habíamos mencionado durante el análisis de este algoritmo: el algoritmo PCA es sencillo de implementar y predominan las funciones básicas de manejo de matrices. Tras analizar más en profundidad las funciones resultantes, observamos que hemos obtenido operaciones con vectores sencillas, funciones básicas para el tratamiento de matrices, operaciones sencillas con matrices y una operación matricial más compleja – obtener los autovectores de una matriz –. Esta última función se apoya en una descomposición SVD y, por esta razón, es la que tiene una mayor complejidad de implementación. Para resolver este problema se ha recurrido a utilizar una serie de librerías especializadas, implementadas en C++. El proceso de adaptación y uso de dos lenguajes simultáneamente se explicará más adelante en este mismo capítulo.

Tras desarrollar el PCA, se implementó el algoritmo seleccionado durante el estudio de la fase de e*xtracción de* endmembers: el algoritmo VCA.

Durante su desarrollo se reutilizaron las siguientes funciones, obtenidas tras implementar el PCA: *mean_vector, vector_minus_value, transpose, matrix_mult, matrix_reduce* y *get_eigenvectMatrix*. Esta reutilización de funciones nos demuestra que la implementación de una librería es una idea óptima ya que, de una fase a otra, se ha reutilizado el 75% de las funciones desarrolladas durante la etapa de *reducción dimensional*.

Por otro lado, las funciones que no coinciden con la etapa anterior son las siguientes:

- *random_vector*: rellena un vector con valores aleatorios entre 0 y 1.
- *vector_minus_vector*: devuelve el vector resultante de la resta de dos vectores.
- *getIndexMax*: devuelve la posición del elemento de mayor valor en valor absoluto de un vector.
- *vector_scale*: divide todos los elementos de un vector por un mismo número.
- *matrix_mult_vector*: multiplica una matriz por un vector.
- *matrix_get_row*: devuelve el vector fila que se desee de una matriz.
- *matrix_set_row*: sustituye una fila de una matriz por el vector que se desee.
- *matrix_extend*: aumenta las dimensiones de una matriz rellenando con ceros.
- *matrix_fill_column*: rellena una columna de una matriz con un valor concreto.
- *matrix_scale*: divide todos los elementos de una matriz por un mismo número.
- *maxSqrtSum*: suma todos los elementos al cuadrado de cada fila de una matriz por separado y, posteriormente, devuelve la raíz cuadrada del máximo de las sumas anteriores.
- *get_pinverse*: devuelve la matriz pseudo-inversa de la matriz de entrada.
- *log*: realiza el logaritmo en base 10 de un número.
- *elev*: eleva un número a la potencia deseada.
- *sq_root*: realiza la raíz cuadrada de un número.
- *rem*: devuelve el resto de la división de dos números.

Como se puede observar, la mayoría de las funciones desarrolladas, al igual que en las funciones de la fase anterior, son operaciones con vectores, tratamiento de matrices u operaciones con matrices**.** Como caso particular, en esta etapa se han desarrollado funciones matemáticas básicas tales como un logaritmo, una potencia, una raíz cuadrada o el resto de una división. Cabe destacar que, en la misma situación que en el desarrollo del PCA, existe una función – *get_pinverse* – que necesita de una descomposición SVD para realizarse satisfactoriamente.

En conclusión, tras el desarrollo de las etapas de *reducción dimensional* y *estimación de* endmembers, se han añadido a la librería un total de 24 funciones, de las cuales seis son comunes a ambas fases. Esto se resume en que un 25% de la librería total es compartido por dos de las cuatro etapas que componen la cadena de procesado completa. En su mayoría, estas funciones son operaciones matemáticas básicas, operaciones vectoriales y operaciones con matrices, siendo las más complejas las asociadas con la descomposición SVD.

Integración de C y C++

Como se ha mencionado en el apartado anterior, para realizar la descomposición SVD se han utilizado una serie de librerías especializadas implementadas en C++. Las librerías utilizadas han sido VXL (del inglés, *Vision something Library*), ITK (*Insight ToolKit – Insight Segmentation and Registration Toolkit*) y OTB (*ORFEO ToolBox*), cuya instalación y configuración están explicadas en el Anexo 1.

Antes de entrar en detalle de cómo se han utilizado estas librerías – implementadas en C++ – desde nuestra librería – desarrollada en C – explicaremos brevemente el contenido de las mismas.

En primer lugar se va a explicar el conjunto de librerías VXL[8]. Esta colección de librerías desarrollada en C++ está enfocada a la investigación e implementación de visión computacional. Para ello, tal y como se explica en su página web (referenciada en el pie de página), aúna una serie de librerías especializadas en distintos campos dentro de la visión computacional. Algunos ejemplos de ello son: VNL (*vision numeric library*), que contiene algoritmos y estructuras para manejar vectores, matrices, etc; VGL (*vision geometric library*), útil para manejar elementos de hasta tres dimensiones; y VUL (*vision utilities library*), que aporta funcionalidades básicas propias de esta librería. Gracias a la amplia gama de estructuras propias de manejo de matrices y vectores, esta librería nos es esencial cuando queremos realizar operaciones complejas con matrices.

A continuación, acerca de la librería ITK[9] cabe destacar que está codificada como *open-source* (código abierto) y se basa, principalmente, en el análisis, registro y segmentación de imágenes médicas. El funcionamiento estándar de dicha librería es el de identificar y clasificar una serie de datos recogidos en una muestra digital – por ejemplo, una imagen tomada con un escáner ultrasónico – o, en su defecto, encontrar correspondencias entre los datos de una o varias imágenes.

Por último, la librería OTB[10] desarrollada por CNES (*French Space Agency*) está compuesta por una serie de algoritmos de procesado de imágenes remotas de gran escala. Al igual que ITK, esta librería es *open-source* y se basa en la propia librería ITK para realizar el procesado de imágenes. En concreto, esta librería se ha desarrollado con el propósito particular de analizar imágenes captadas a distancia y, más concretamente, imágenes de alta resolución espacial.

La utilización conjunta de ITK y OTB nos permita utilizar de manera transparente la librería VXL. Su uso nos proporciona la capacidad de realizar descomposiciones SVD con una única instrucción y, posteriormente, obtener las matrices de autovectores, autovalores, inversas, pseudo-inversas, etc, de manera totalmente automática.

Una vez comentadas las librerías, es importante mencionar que la librería desarrollada a lo largo de este Proyecto Fin de Grado se decidió implementar en lenguaje C por su alta interoperabilidad entre plataformas. Por lo tanto, a continuación, se va a explicar el método para integrar C y C++. Para hacer más fácil el seguimiento de dicha explicación se utilizará un ejemplo real durante todo el tiempo - en este caso, la implementación de la función *get_eigenvectMatrix*.

[8] http://vxl.sourceforge.net/
[9] http://www.itk.org/
[10] http://orfeo-toolbox.org/otb/

La integración de estos dos lenguajes se ha llevado a cabo mediante la utilización de un *wrapper*. Este método se basa en que el compilador pueda leer un archivo tanto en lenguaje C como en lenguaje C++ y, de esta manera, conseguir realizar un puente entre ambos lenguajes. Para ello, en este método están implicados cinco archivos, explicados a continuación.

En primer lugar, dentro del archivo que contiene la librería codificada en C – en nuestro caso, este archivo se denomina *hsi_analysis.c* – se realizan dos pasos. El primero de ellos es importar el *wrapper* desarrollado; como se observa en la figura 3.5, hemos incluido una línea de importación en la cabecera, mediante la cual se añade el archivo *cppWrapper.h,* donde se incluyen los prototipos de las *funciones puente* necesarias. Estas *funciones puente* son aquellas que están incluidas dentro del *wrapper* y que, como característica principal, pueden ser leídas como métodos de una clase en C++ y como funciones de una estructura en C.

```
#include "cppWrapper.h"
```

Fig. 3.5. Importación del wrapper *en la librería implementada en C*

El segundo paso es implementar la función que utilizará lenguaje C++. El código necesario para implementar la función *get_eigenvectMatrix* se muestra en la figura 3.6. En ella observamos que, en primer lugar, creamos una estructura del tipo *HsiCppFunctions* – éste es el nombre con el que hemos denominado a nuestra clase en C++ –; posteriormente, utilizamos la *función puente* asociada a la obtención de los autovectores de una matriz y, por último, eliminamos la estructura creada para liberar memoria.

```
void get_eigenvectMatrix(double *matrixIn, double *matrixEigenvectors, int rows, int columns){

        struct HsiCppFunctions* c = newHsiCppFunctions();
        HsiCppFunctions_get_eigen(c, matrixIn, matrixEigenvectors, rows, columns);
        deleteHsiCppFunctions(c);
}
```

Fig. 3.6. Código de la función get_eigenvectMatrix

El segundo archivo implicado es el que hemos mencionado anteriormente: *cppWrapper.h*. Este archivo tiene la característica de poder ser leído en lenguaje C y en lenguaje C++ y, dentro del mismo, se definen los prototipos de las *funciones puente*. A su vez, contiene la definición de la estructura *HsiCppFunctions* (que puede ser leída desde C) y el constructor de la clase *HsiCppFunctions* (utilizado desde el lenguaje C++). Todo esto se puede observar en la figura 3.7. Un detalle destacable de los prototipos de las *funciones puente* es que tienen como parámetro la propia estructura. Esto se debe a que dicho parámetro es un puntero que tiene almacenado la dirección de memora de la estructura en C y de la clase en C++ – la cual, gracias a la utilización del *wrapper,* es la misma para ambos.

```
#ifndef __CPPWRAPPER_H
#define __CPPWRAPPER_H

#ifdef __cplusplus
extern "C" {
#endif

typedef struct HsiCppFunctions HsiCppFunctions;

HsiCppFunctions* newHsiCppFunctions();

void HsiCppFunctions_get_eigen(HsiCppFunctions* v,double* matrixIn, double* matrixU, int rows
void deleteHsiCppFunctions(HsiCppFunctions* v);

#ifdef __cplusplus
}
#endif
#endif
```

Fig. 3.7. Código del archivo cppWapper.h *para la función* get_eigenvectMatrix

En tercer lugar, el archivo a implementar es el que nosotros hemos denominado *cppWrapper.cpp*. Este fichero, codificado en C++, tiene codificadas las *funciones puente* y, además, incluye tanto el fichero con la estructura de *HsiCppFunctions* (*cppWrapper.*h) como el archivo – explicado más adelante – que contiene la clase codificada en C++ (*HsiCppFunctions.h*). Estas *funciones puente* tienen el formato de una llamada a un método de una clase implementada en C++ pero, a su vez, se realizan desde una función codificada en C – esta configuración es la que nos permite incorporar funciones desarrolladas en C++ a la librería original implementada en C-. Todas estas explicaciones, junto a la codificación del constructor y el destructor de la estructura *HsiCppFunctions,* se observan en la figura 3.8.

```
#include "HsiCppFunctions.h"
#include "cppWrapper.h"

extern "C" {
    HsiCppFunctions* newHsiCppFunctions() {
        return new HsiCppFunctions();
    }
    void HsiCppFunctions_get_eigen(HsiCppFunctions* v, double* matrixIn, double* matrixU, int rows
        v->get_eigen(matrixIn, matrixU, rows, columns);
    }
    void deleteHsiCppFunctions(HsiCppFunctions* v) {
        delete v;
    }
}
```

Fig. 3.8. Código del archivo cppWapper.cpp *para la función* get_eigenvectMatrix

Una vez codificado el *wrapper,* empezamos a codificar la clase en C++. Para ello, tenemos que utilizar dos archivos: el primero contiene la importación de las librerías especializadas, las definiciones de los tipos de datos que usaremos y las cabeceras de las funciones; mientras que el segundo se compone de las librerías estándar a utilizar y la codificación de las distintas funciones.

El primero de ellos, en el caso en el que únicamente se implementase la función *get_eigenvectMatrix*, se correspondería con la figura 3.9. En ella, observamos que se han importado librerías especializadas – *itkImageRegionIterator.h* y *vnl/algo/vnl_qr.h* son un ejemplo de dichas librerías – y, a su vez, se han declarado distintos tipos de datos utilizando estas librerías – por ejemplo, el tipo de dato *SVDType* se corresponde con el que aportan las librerías especializadas para realizar descomposiciones SVD.

```
#ifdef __cplusplus

#include "itkImageRegionIterator.h"
#include <vnl/algo/vnl_qr.h>
#include <math.h>
#include <vnl/vnl_random.h>

#ifndef __HsiCppFunctions_H
#define __HsiCppFunctions_H

using namespace std;

typedef vnl_matrix<double> Matrix;
typedef vnl_vector<double> Vector;
typedef vnl_svd<double> SVDType;

class HsiCppFunctions {

        public:

                void get_eigen(double* matrixIn, double* matrixU, int rows, int columns);
};

#endif
#endif
```

Fig. 3.9. Código del archivo HsiCppFunctions.h *para la función* get_eigenvectMatrix

Por último, tenemos que codificar la función que realice el cálculo de los autovectores asociados a la matriz de entrada en el archivo *HsiCppFunction.cpp*. Para ello, como se observa en la figura 3.10, en primer lugar importamos el archivo *HsiCppFunctions.h,* que contiene las librerías, los tipos de datos y los prototipos de las funciones que vamos a utilizar e implementar. A continuación, importamos las librerías propias de la codificación de las distintas funciones; en este caso, se han importado aquellas que nos permiten manejar cadenas de caracteres – *cstring y string.h* – y las que nos permiten tener una interfaz de entrada - salida, ya sea por pantalla o en fichero (*iostream* y *fstream,* respectivamente). Tras esto, se realiza la codificación de la función *get_eigen,* que nos devuelve los autovectores que se desean calcular.

```
#include <HsiCppFunctions.h>
#include <iostream>
#include <fstream>
#include <cstring>
#include <string.h>


void HsiCppFunctions::get_eigen(double* matrixIn, double* matrixU, int rows, int columns) {

        int i;
        int j;
        Matrix SubsetMatrix(rows,columns);
        Matrix U(rows,columns);

        for(i=0; i < rows; i++){
                for(j=0; j < columns; j++){
                        SubsetMatrix.put(i,j,matrixIn[i*columns+j]);
                }
        }

        SVDType s(SubsetMatrix);
        U = s.U();

        for(i=0; i < rows; i++){
                for(j=0; j < columns; j++){
                        matrixU[i*columns+j] = U.get(i,j);
                }
        }
}
```

Fig. 3.10. Código del archivo HsiCppFunctions.cpp *para la función* get_eigenvectMatrix

En último lugar, cabe destacar que, durante la codificación de las distintas funciones, gracias al uso de las librerías VXL, ITK y OTB, la descomposición SVD y la obtención de los autovectores asociados a la matriz de entrada se realizan en dos líneas de código. En la figura 3.11 se observa un detalle de esta funcionalidad.

```
SVDType s(SubsetMatrix);
U = s.U();
```

Fig. 3.11. Detalle de la descomposición SVD y del cálculo de autovectores

Para finalizar la explicación de las funciones relacionadas con el cálculo de los distintos parámetros asociados a la descomposición SVD – inversas, pseudo-inversas, autovalores, etc – cabe mencionar que, una vez desarrollada la función que calcula los autovectores, la única modificación a realizar es la de sustituir la línea *U = s.U();* por la que corresponda con el parámetro deseado. Como ejemplo, en la figura 3.12 observamos el detalle de la descomposición para el cálculo de la matriz pseudo-inversa necesaria en el algoritmo VCA.

```
SVDType s(SubsetMatrix);
Pinverse = s.pinverse();
```

Fig. 3.12. Detalle de la descomposición SVD y del cálculo de la matriz pseudo-inversa

Integración de la librería en RVC – CAL

Una vez explicada la manera de implementar todas las funciones que se han necesitado durante el desarrollo de la librería, se va a exponer el método utilizado para integrar la librería desarrollada en RVC – CAL.

Dicho método se fundamenta en la definición de la librería desarrollada como una librería de funciones nativas, o lo que es lo mismo, funciones externas. Estas funciones nativas se recogen en un archivo de extensión .cal – la misma extensión que tienen los actores, ya que ambos están implementados en RVC – CAL – y, posteriormente, cuando se quiera utilizar dicha librería, se importará como un paquete externo, tal y como se explicó en el capítulo 2 de este mismo documento – como recordatorio, en la figura 3.13 se muestra la sentencia que se ha utilizado durante este Proyecto Fin de Grado para importar el archivo *hsi_analysis.cal* (nombre que se le ha asignado en este proyecto a la librería de funciones nativas).

```
import hsi_analysis.hsi_analysis.*;
```

Fig. 3.13. Importación del paquete de funciones nativas hsi_analysis.cal.

Por otro lado, a modo de resumen, la figura 3.14 recoge el archivo con las funciones necesarias para implementar el algoritmo PCA. A su vez, en la figura 3.15 se observa la facilidad de codificar dicho algoritmo utilizando únicamente las funciones implementadas en la librería.

```
package hsi_analysis;

unit hsi_analysis:
    uint(size=32) MAX = 10000;
    uint(size=32) rowsMax = 4096;
    uint(size=32) columnsMax = 4096;
    uint(size=32) bandsMax = 4096;

    @native function mean_vector(double vector[MAX], int(size = 32) positions) --> double
    end
    @native procedure vector_minus_value(double vectorIn[MAX], double value, double vectorMinus[MAX], int(size=32) positi
    end
    @native procedure transpose(double matrixIn[MAX][MAX], double matrixTranspose[MAX][MAX], double rows, double columns)
    end
    @native procedure matrix_mult(double matrix1[MAX][MAX], double matrix2[MAX][MAX], double matrixResult[MAX][MAX], doub
    end
    @native procedure get_eigenvectMatrix(double matrixIn[MAX][MAX],double matrixEigenvectors[MAX][MAX], int(size=32) row
    end
    @native procedure matrix_reduce(double matrix1[MAX][MAX], double matrix2[MAX][MAX], double rows1, double columns1, do
    end
    @native procedure matrix_get_column(double matrix1[MAX][MAX], double vector[MAX], double rows1, double columns1, doub
    end
    @native procedure matrix_set_column(double matrix1[MAX][MAX], double vector[MAX], double rows1, double columns1, doub
    end
end
```

Fig. 3.14. Contenido del archivo hsi_analysis.cal.

```
PCA_alogrithm: action ==>

do
    foreach int(size=32) i in 0 .. bands-1 do
        matrix_get_column(image_IN, averageVector, rc, bands, i);
        averageVectorResult := mean_vector(averageVector, rc);
        vector_minus_value(averageVector, averageVectorResult,minusVectorResult, rc);
        matrix_set_column(image_Prima, minusVectorResult,rc,bands,i);
    end
    transpose(image_Prima, image_Prima_Transp, rc, bands);
    matrix_mult(image_Prima_Transp, image_Prima, image_Mult, bands, rc, rc, bands);
    get_eigenvectMatrix(image_Mult, matrix_eigen, bands, bands);
    matrix_mult(image_IN, matrix_eigen, image_PCA, rc, bands, bands, bands);
    matrix_reduce(image_PCA, PCA_out, rc, bands, rc, num_PC);

    pca_done := true;
end
```

Fig. 3.15. Implementación del algoritmo PCA utilizando la librería

En cuanto a la implementación del archivo *hsi_analysis.cal,* hay que tener en cuenta los siguientes puntos:

- En primer lugar, el nombre de las funciones en este archivo tiene que coincidir con el nombre de las funciones en el archivo *hsi_analysis.c*.
- Si en el archivo *hsi_analysis.c* no existe valor de retorno (*void*), entonces se considera que es un procedimiento y, en consecuencia, la sentencia de declaración del archivo *hsi_analysis.cal* es un *procedure*. En caso contrario, la sentencia es una *function* y el valor devuelto se indica al final de la instanciación. Un ejemplo de este último caso se da en la función *mean_vector,* en la que se devuelve un valor *double.*
- Los tamaños de los parámetros determinados en las instanciaciones de las funciones nativas son los tamaños máximos con los que operarán las funciones externas.

Todo lo explicado anteriormente nos permite generar código desde RVC – CAL, en el que se utilicen las funciones implementadas de manera externa. A continuación, se va a exponer el método a seguir para compilar y ejecutar la aplicación generada satisfactoriamente. Esta explicación se hará en dos partes ya que, debido a la incorporación del lenguaje C++ y de librerías especializadas, en este Proyecto Fin de Grado se han tenido que realizar más modificaciones de las que serían necesarias si únicamente se hubiese implementado la librería en lenguaje C.

En primer lugar, la integración básica de una librería implementada en su totalidad en C consiste en modificar los archivos de configuración para la compilación del código que se ha generado desde RVC – CAL. En concreto, el único archivo que debemos modificar es el que se corresponde con las librerías de funciones nativas utilizadas. Este fichero es el *CMakeLists.txt* que se encuentra en la carpeta *libs/orcc-native* de la carpeta de salida del código generado por RVC – CAL.

En este archivo hay que indicar que se añade una nueva librería para que se realice correctamente la compilación. Para ello, hay que añadir la línea que aparece resaltada en la figura 3.16, indicando así que el archivo *hsi_analysis.c* pertenece a la lista de librerías nativas para la compilación del programa generado por ORCC.

```
# Orcc library files
set(orcc_native_sources
    src/access_file.c
    src/source.c
    src/writer.c
    src/compare.c
    src/compareyuv.c
    src/native.c
    src/native_util.c
    src/hsi_analysis.c
)
```

Fig. 3.16. Modificación estándar del archivo CMakeLists.txt *del directorio* libs/orcc-native.

Por último, tenemos que incorporar el archivo *hsi_analysis.c* al directorio en el que se encuentran el resto de las librerías nativas autogeneradas por ORCC. En concreto, este directorio se corresponde con *libs/orcc-native/src*. Tras esto, la configuración, compilación y ejecución de la aplicación desarrollada utilizará sin ningún problema las librerías externas que se hayan incorporado.

Debido a que en el desarrollo de este Proyecto Fin de Grado se han utilizado librerías especializadas implementadas en C++, se han realizado una serie de modificaciones extra en el archivo *CMakeLists.txt* modificado anteriormente para poder realizar la compilación de manera satisfactoria.

En primer lugar, se realiza una búsqueda para ver si las librerías especializadas se han instalado y configurado correctamente – con el fin de facilitar esta instalación se aporta en el anexo 1 una guía de instalación y configuración de dichas librerías –. Como se puede observar en la figura 3.17, se realiza una búsqueda individualizada para cada librería (la librería ITK recibe el nombre de FLTK en dicha búsqueda).

A
```
FIND_PACKAGE(OTB)
include(${OTB_USE_FILE})
IF(OTB_FOUND)
  MESSAGE("Can build OTB project with OTB.OTB_DIR: ${OTB_DIR}")
  MESSAGE("Can build OTB project with OTB.OTB_DIR: ${OTB_USE_FILE}")
ELSE(OTB_FOUND)
  MESSAGE(FATAL_ERROR
      "Cannot build OTB project without OTB.  Please set OTB_DIR.")
ENDIF(OTB_FOUND)
```

B
```
FIND_PACKAGE(FLTK)
INCLUDE_DIRECTORIES(${FLTK_INCLUDE_DIR})
```

C
```
FIND_PACKAGE(VXL)
SET(VXL_PROVIDE_OLD_CACHE_NAMES 1)
SET(VXL_PROVIDE_STANDARD_OPTIONS 1)
INCLUDE(${VXL_CMAKE_DIR}/UseVXL.cmake)
IF(VXL_FOUND)
  MESSAGE("Can build VXL project with VXL.VXL_DIR: ${VXL_DIR}")
  MESSAGE("Can build VXL project with VXL.VXL_DIR: ${VXL_USE_FILE}")
ELSE(VXL_FOUND)
  MESSAGE(FATAL_ERROR
      "Cannot build VXL project without VXL.  Please set VXL_DIR.")
ENDIF(VXL_FOUND)
```

Fig. 3.17. Búsqueda de las librerías especializadas: (A) Librería OTB, (B) Librería ITK y (C) Librería VXL

Tras realizar la búsqueda de las librerías especializadas, el siguiente paso es añadir los ficheros que se han utilizado para integrar código C++ con el lenguaje propio de la librería – C –. Para ello, seguiremos el mismo procedimiento que para integrar una librería estándar. En concreto, añadiremos los ficheros fuente necesarios al apartado del *CMakeLists.txt* correspondiente a las funciones nativas – este paso se corresponde con las líneas resaltadas de la figura 3.18.

```
# Orcc library files
set(orcc_native_sources
    src/access_file.c
    src/source.c
    src/writer.c
    src/compare.c
    src/compareyuv.c
    src/native.c
    src/native_util.c
    src/hsi_analysis.c
    src/HsiCppFunctions.cpp
    src/cppWrapper.cpp
)
```

Fig. 3.18. Modificación particular del archivo CMakeLists.txt *del directorio* libs/orcc-native.

La última modificación a realizar del archivo *CMakeLists.txt* tiene por objetivo llevar a cabo correctamente el *linkado* de las librerías especializadas; esto es, que durante la ejecución del programa, las funciones y los tipos de dato que contienen dichas librerías se puedan utilizar correctamente. La modificación que hay que realizar es la que aparece señalada en la figura 3.19.

```
# Do the linking
target_link_libraries(orcc-native orcc-runtime ${extra_libraries}  vnl_algo vul vil vcl vnl)
```

Fig. 3.19. Linkado de librerías en el archivo CMakeLists.txt

Una vez modificado por completo el fichero *CMakeLists.txt*, para que todo funcione correctamente tenemos que mover los archivos que componen la librería a sus directorios correspondientes. Los directorios que utilizaremos serán dos:

- *libs/orcc-native/src*: este directorio contiene los ficheros fuente de las funciones nativas. En dicho directorio se almacenarán tanto las librerías generadas automáticamente por ORCC, como las importadas por el usuario. Los ficheros que incluiremos en este directorio serán: *hsi_analysis.c*, *cppWrapper.cpp* y *HsiCppFunctions.cpp*.

- *libs/orcc-native/include*: a diferencia del directorio anterior, en *include* se añadirán los ficheros que contienen las cabeceras de las funciones utilizadas. En nuestro caso, este directorio únicamente se modificará debido a la inclusión de lenguaje C++. Por ello, los ficheros a incluir serán: *cppWrapper.h* y *HsiCppFunctions.h*.

Integración de PAPI

Para finalizar este capítulo, únicamente queda explicar el método utilizado para integrar la herramienta software de análisis de rendimiento PAPI. Para que la integración sea más sencilla se ha utilizado una herramienta denominada *papify*[11], desarrollada por Alejo Iván Arias – miembro del grupo GDEM.

Esta herramienta está pensada para funcionar con código autogenerado en C por ORCC – el compilador de lenguaje RVC – CAL explicado en el capítulo 2 de este mismo documento.

El funcionamiento de esta herramienta está explicado en profundidad en su página de *github*. A continuación, se aporta un resumen de los aspectos más importantes propios de la configuración de *papify*. Dicha configuración se realiza en un archivo autogenerado con extensión *.xcf,* contenido en el directorio *src* de la carpeta que contiene el código autogenerado desde RVC – CAL. Este archivo, por defecto, únicamente contiene el mapeo con el que se ha configurado el programa a ejecutar pero, a su vez, permite añadir nuevas opciones de configuración.

En concreto, la herramienta *papify* nos permite, utilizando el fichero nombrado anteriormente, integrar la librería PAPI con tres niveles de detalle:

- A nivel de sistema.
- A nivel de actor.
- A nivel de acción.

[11] https://github.com/alejoar/papify

En el primero de ellos, únicamente se indicarán – tal y como se observa en la figura 3.20 – los eventos que se quieren monitorizar y, tras esto, *papify* nos permitirá monitorizar todas las acciones de todos los actores que componen el sistema completo.

```
<Papi>
    <Events>
        <Event id="PAPI_TOT_INS"/>
        <Event id="PAPI_L1_DCM"/>
        <Event id="PAPI_TOT_CYC"/>
    </Events>
</Papi>
```

Fig. 3.20. Monitorización del sistema utilizando papify. *(Fuente: https://github.com/alejoar/papify).*

El segundo nivel de detalle nos permite monitorizar únicamente los actores indicados. Utilizando esta configuración, se monitorizarán todas las acciones de los actores que se indiquen. Un ejemplo de esta configuración se observa en la figura 3.21.

```
<Papi>
    <Events>
        <Event id="PAPI_TOT_INS"/>
        <Event id="PAPI_L1_DCM"/>
        <Event id="PAPI_TOT_CYC"/>
    </Events>
    <Instances>
        <Instance id="SomeActor"/>
        <Instance id="SomeOtherActor"/>
    </Instances>
</Papi>
```

Fig. 3.21. Monitorización de actores concretos utilizando papify. *(Fuente: https://github.com/alejoar/papify*)

Por último, si únicamente se desean monitorizar una serie de acciones dentro de un actor, se debe indicar utilizando el formato que se muestra en la figura 3.22.

```
<Papi>
    <Events>
        <Event id="PAPI_TOT_INS"/>
        <Event id="PAPI_L1_DCM"/>
        <Event id="PAPI_TOT_CYC"/>
    </Events>
    <Instances>
        <Instance id="SomeActor">
            <Action id="SomeAction"/>
            <Action id="SomeOtherAction"/>
        </Instance>
        <Instance id="SomeOtherActor"/>
    </Instances>
</Papi>
```

Fig. 3.22. Monitorización de acciones concretas utilizando papify. *(Fuente: https://github.com/alejoar/papify*)

En nuestro caso particular, se ha optado por la primera opción ya que, al estar comenzando con el análisis del consumo, se ha decidido que obtener una visión global de los recursos que consumen todos los actores para, de esta manera, hacernos una primera idea de dónde se localizan los cuellos de botella generales.

Para finalizar la explicación, cabe destacar que los resultados obtenidos por *papify* se almacenan en ficheros de extensión *.csv*. Cada actor monitorizado genera su propio archivo *.csv,* indicando en dichos resultados la acción, el evento y el valor del registro asociado al evento que se corresponde con cada ejecución de cada acción.

CAPÍTULO 4. ANÁLISIS DE RESULTADOS

A lo largo de este capítulo se llevará a cabo un estudio de los resultados obtenidos tras implementar y utilizar la librería desarrollada. En primer lugar se realizará un análisis individual de cada etapa, atendiendo a la bondad de los resultados y a su tiempo de ejecución. Tras esto, se estudiará el comportamiento de la cadena completa – la cual se ha construido utilizando las interfaces de comunicación y de entrada-salida explicadas por Raquel Lazcano en [1] –. Este estudio tendrá en cuenta aspectos como la bondad de los resultados, el tiempo total de procesamiento y, aplicando PAPI, el consumo de recursos del sistema completo. Además, se ha de mencionar que, como último caso de estudio, se recoge un análisis de tiempos y consumo de recursos aplicando división en núcleos.

Cabe destacar que durante el desarrollo del presente capítulo se harán numerosas referencias al trabajo realizado por Raquel Lazcano en [1]. Esto se debe a que, para analizar la cadena completa de procesado, se ha incluido la parte desarrollada durante su Proyecto Fin de Grado.

A su vez, conviene resaltar el uso de la herramienta *HyperMix* durante la elaboración de este capítulo. *HyperMix* es una herramienta de procesado de imágenes hiperespectrales desarrollada por la Universidad Politécnica de Extremadura [32]. Su utilización nos ha servido para comparar sus tiempos de ejecución con los del programa desarrollado en RVC – CAL. Al ser una herramienta de código abierto ha podido ser instrumentada, es decir, medir el tiempo de ejecución o imprimir los resultados obtenidos en un archivo. Al mismo tiempo, se ha utilizado el visor de abundancias y *endmembers* hiperespectrales proporcionado por *HyperMix* para analizar la bondad de los algoritmos desarrollados y para comprobar su fiabilidad. Por ello, las figuras aportadas a lo largo del presente capítulo han sido obtenidas utilizando dicho visor.

Para concluir la introducción del presente capítulo, cabe mencionar que las imágenes utilizadas como imágenes de prueba – denominadas *fractals* a lo largo del capítulo – han sido obtenidas de la página oficial de *HyperMix*[12]. Un dato importante referente a dichas imágenes es que son sintéticas, es decir, han sido creadas artificialmente partiendo de firmas espectrales de una base de datos – en este caso, *United States Geological Survey* –. Nos encontramos con un total de cinco *fractals* distintas y, de cada una de ellas, nos ofrecen una versión sin ruido añadido y cinco imágenes con una relación señal a ruido que varía desde 110 a 10. Cada una de estas imágenes está formada por 100 filas, 100 columnas y 221 bandas, es decir, un total de 10000 píxeles y una resolución espectral de 221 bandas.

4.1. Análisis por fases

En este apartado, tal y como ocurre en el capítulo 3 de este mismo documento, se recogen únicamente dos de las cuatro etapas del procesado espectral. En concreto, se analizan los resultados obtenidos para el algoritmo PCA – propio de la etapa de *reducción dimensional* – y para el algoritmo VCA – perteneciente a la etapa de *extracción de* endmembers.

A continuación, se explica el procedimiento seguido a la hora de realizar pruebas de bondad sobre el algoritmo PCA. La batería de pruebas de este algoritmo se resume en aplicar una reducción dimensional a 25, 50, 75 y 100 bandas tanto a la *fractal_1* como a la *fractal_2*. Para comprobar el correcto funcionamiento de nuestro programa se compararon los resultados obtenidos de éste con los que arroja *HyperMix*.

[12] http://www.hypercomp.es/hypermix/Downloads

Para realizar dicha comparativa, se utilizó un programa denominado *Meld Diff Viewer*[13]. Este programa compara archivos de texto por lo que, como se ha mencionado anteriormente, uno de los cambios que se realizó durante la instrumentación de *HyperMix* fue la de escribir en un fichero de texto el resultado final – en este caso, la matriz de salida del PCA –. A su vez, se utilizó en el algoritmo desarrollado con RVC – CAL la función *write_txt* explicada en [1]. En ambos casos, el objetivo es escribir la matriz de salida en un fichero de extensión .txt.

Gracias a la instrumentación de *HyperMix*, y a la utilización de la función *write_txt* y el programa *Meld Diff Viewer* se obtuvieron los siguientes resultados:

- La salida de *HyperMix* para la misma *fractal* y el mismo número de bandas es idéntica en todas sus ejecuciones.
- La salida del programa desarrollado en RVC – CAL para la misma *fractal* y el mismo número de bandas es idéntica en todas sus ejecuciones.
- La salida generada por ambos programas es idéntica entre sí.

Tras obtener estos resultados, concluimos que el algoritmo PCA funciona de manera equivalente a como lo hace el programa *HyperMix*. También se observó que dicho algoritmo sigue siempre el mismo patrón a la hora de ordenar las distintas bandas de la imagen reducida, puesto que las 25 primeras bandas de todas las pruebas de una misma *fractal* coinciden con la reducción de dicha *fractal* a 25 bandas. Esto nos demuestra que la información útil siempre se mantendrá en las primeras bandas y que, tal y como se explicó en el capítulo 3 de este mismo documento, la información redundante y el ruido se mantendrán en las últimas bandas.

Para demostrar tanto la equivalencia de resultados entre *HyperMix* y el programa desarrollado en RVC – CAL como el reparto de información dentro de las bandas obtenidas, se aporta en la tabla 4.1 un resumen con las cinco primeras bandas y las cinco últimas de la *fractal_2,* reducida a 25 bandas, en su versión sin ruido añadido.

Analizando la tabla 4.1, se observa visualmente que los resultados son idénticos entre sí (el resultado mostrado a la izquierda de cada banda corresponde con *HyperMix* y a la derecha con el programa en RVC – CAL). También se observa que, como se ha mencionado anteriormente, las primeras bandas contienen información nítida y, en consecuencia, se distinguen perfectamente las distintas formas que componen la imagen pero, a su vez, las últimas bandas están compuestas principalmente por ruido.

Para finalizar el análisis del algoritmo PCA, en la tabla 4.2 se recogen los resultados del análisis de tiempos de ejecución realizado. Este estudio se ha llevado a cabo utilizando la denominada *fractal_2* para 25 y 100 bandas. Para realizar las medidas que se muestran en dicha tabla se han realizado diez pruebas consecutivas en cada uno de los programas – *HyperMix* y el desarrollado en este Proyecto Fin de Grado para RVC – CAL.

Como se observa en la tabla 4.2, la diferencia de tiempos entre medir 25 bandas y medir 100 es prácticamente nula. Concretamente, el tiempo máximo y el tiempo mínimo que tarda en analizar cualquiera de los casos tienen una separación menor a 1 segundo. A su vez, observamos que nuestro programa presenta una mejora con respecto a *HyperMix* de aproximadamente 0.4 segundos. Teniendo en cuenta que no se ha realizado paralelización del algoritmo, esta mejora nos sugiere que la idea de utilizar una librería en RVC – CAL es bastante prometedora.

[13] http://meldmerge.org/

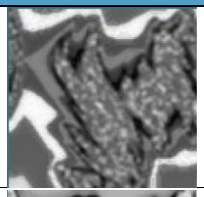
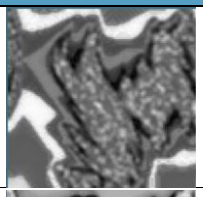
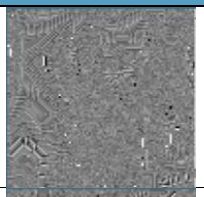
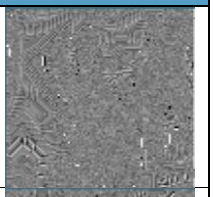
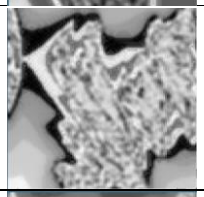
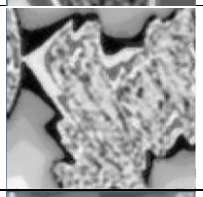
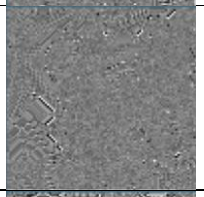
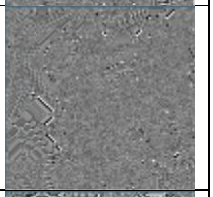
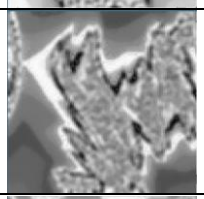
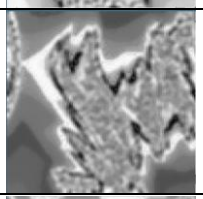
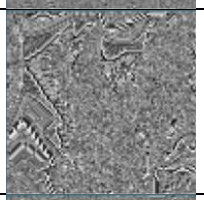
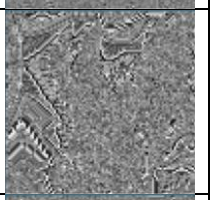
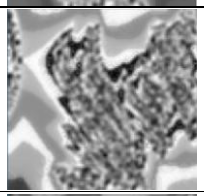
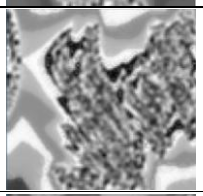
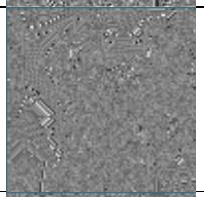
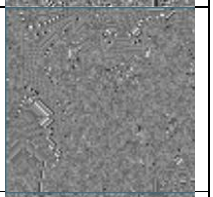
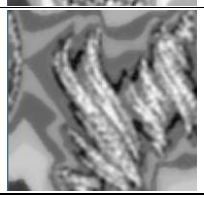
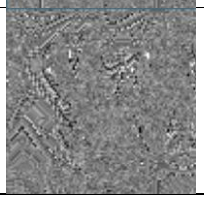
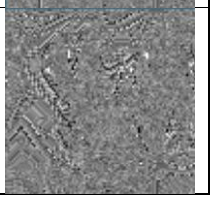
Banda	*HyperMix*	RVC – CAL	Banda	*HyperMix*	RVC – CAL
0			20		
1			21		
2			22		
3			23		
4			24		

Tabla 4.1. Comparativa de bondad PCA

Prueba		*HyperMix*	RVC – CAL
25 bandas	Mínimo	11.2592	10.774764
	Media	11.28131	10.7954899
	Máximo	11.3091	10.812785
100 bandas	Mínimo	11.2719	10.792723
	Media	11.38448	10.8169086
	Máximo	11.6964	10.864475

Tabla 4.2. Comparativa de tiempos PCA – las medidas se dan en segundos

En conclusión, tras analizar el algoritmo PCA en tiempo y bondad y compararlo con *HyperMix*, hemos observado que, utilizando la librería desarrollada para RVC – CAL – sin utilizar división en procesadores – obtenemos los mismos resultados en un tiempo menor. Esto pone de manifiesto la utilidad de desarrollar la librería y el gran potencial que tiene la misma si se optimizan los algoritmos para que el tiempo de procesado se acerque al tiempo real. Como se puede observar, salta a la vista la limitación que supone realizar un procesado secuencial, ya que al tardar más de diez segundos, dicho tipo de procesado queda descartado.

A continuación, se analizan los resultados obtenidos para el algoritmo VCA de la fase de *extracción de* endmembers. Para comprender las pruebas realizadas, cabe mencionar que dicho algoritmo tiene una etapa aleatoria en la que se genera iterativamente un vector aleatorio que comprende valores entre 0 y 1. Por ello, el análisis del algoritmo VCA se ha dividido en las siguientes tres fases:

- En primer lugar, se analizará el algoritmo sin tener en cuenta dicha etapa aleatoria. Para ello, se ha almacenado el resultado de los vectores aleatorios generados por *HyperMix* y, tras esto, se han introducido dichos vectores en la etapa aleatoria del programa codificado utilizando la librería desarrollada. La finalidad de esta prueba es comprobar si los resultados de ambos programas son equivalentes cuando trabajan sin aleatoriedad.
- Tras esto, analizaremos el algoritmo completo, comparando los resultados aleatorios de ambos programas. De esta manera, si los resultados obtenidos son similares tras varias repeticiones – los *endmembers* calculados tienen que ser equivalentes a pesar de la aleatoriedad del algoritmo – concluiremos que el algoritmo completo funciona correctamente.
- Por último, tal y como se hizo para el algoritmo anterior, analizaremos y compararemos los tiempos de ejecución de ambos programas.

Para el primer caso se han realizado cuatro pruebas: utilizando la *fractal_1* reducida mediante PCA a 100 bandas, se han calculado primero 15 y luego 20 *endmembers* mientras que, utilizando la *fractal_2* reducida a 25 bandas, se han calculado primero 5 y después 10 *endmembers*.

Para realizar las comprobaciones de equivalencia de resultados, tal y como se realizó en el algoritmo PCA, se utilizó el programa *Meld Diff Viewer,* de tal manera que se comparasen los *endmembers* resultantes de *HyperMix* con los del programa desarrollado en RVC – CAL. Tras realizar dicha comparativa, los resultados de los cuatro casos fueron que ambos programas generan exactamente los mismos *endmembers*.

Como ejemplo gráfico, en la figura 4.1 se aportan los 5 *endmembers* calculados para el caso de la *fractal_2* reducida a 25 bandas. A la izquierda se aporta el resultado de *HyperMix,* y a la derecha el resultado arrojado por el programa que utiliza la librería desarrollada a lo largo de este Proyecto Fin de Grado.

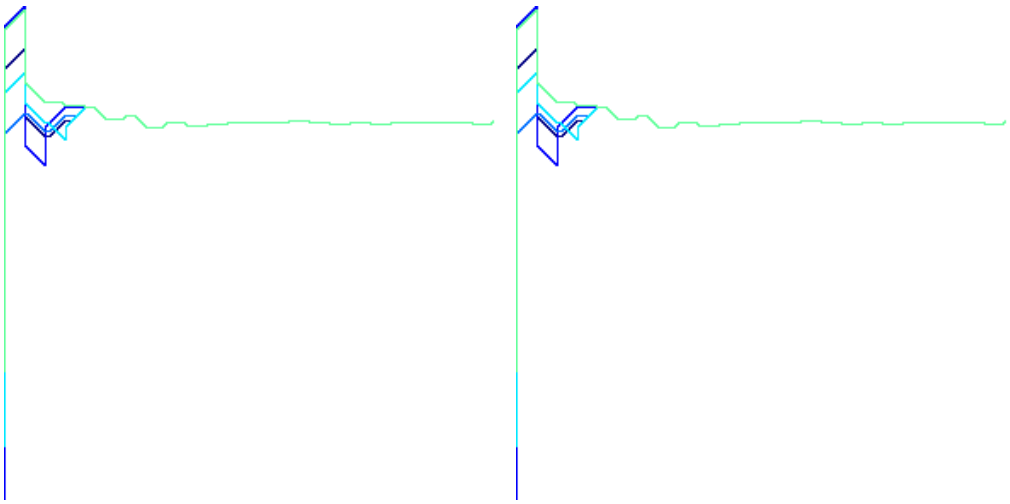


Fig. 4.1. Comparativa de 5 endmembers *calculados para la* fractal_2 *reducida a 25 bandas.* Hypermix *(izq) – RVC – CAL (der)*

Tras observar que los resultados de ambos programas sin la componente aleatoria son idénticos, con el fin de verificar el correcto funcionamiento del algoritmo completo se analiza el comportamiento de la componente aleatoria. Para ello, se realizaron un total de 16 pruebas, en las cuales se comparaban una serie de índices seleccionados por el algoritmo – cada índice se corresponde unívocamente con una firma espectral o endmember concreto –. Las pruebas realizadas han sido cinco repeticiones en *HyperMix* y otras cinco en el programa desarrollado para RVC – CAL. Tras esto, se han ordenado los resultados obtenidos en una tabla y se ha comprobado cuántos índices se repiten un número concreto de veces. La tabla 4.3 recoge un resumen de estos resultados para cada una de las pruebas realizadas. Como se puede observar, la mayoría de los resultados se concentran por encima del 8 y por debajo del 3. Este resultado indica que existe una serie de *endmembers* fijos – que se repiten en la mayoría de las repeticiones – y que el resto de *endmembers* aparecen esporádicamente.

Prueba	10	9	8	7	6	5	4	3	2	1
F1_25B_5End	2	2	-	-	-	1	1	-	-	3
F1_25B_10End	4	2	2	1	1	-	-	1	2	6
F1_25B_15End	2	3	2	2	3	1	3	4	5	15
F1_25B_20End	3	3	3	1	7	4	2	3	7	19
F1_100B_5End	3	-	1	1	-	-	-	-	2	1
F1_100B_10End	3	2	1	2	2	-	-	-	5	8
F1_100B_15End	2	4	4	1	1	3	3	3	1	10
F1_100B_20End	3	1	6	2	3	5	5	4	7	10
F2_25B_5End	2	1	1	1	-	-	-	1	1	1
F2_25B_10End	7	1	-	-	1	2	1	-	-	1
F2_25B_15End	5	1	2	1	2	4	4	1	4	9
F2_25B_20End	6	1	2	2	3	7	4	6	5	4
F2_100B_5End	2	1	-	1	1	-	1	1	-	1
F2_100B_10End	7	1	-	-	1	2	-	1	-	2
F2_100B_15End	5	2	2	2	-	2	5	2	4	8
F2_100B_20End	9	2	1	1	2	4	8	1	2	6

Tabla 4.3. Comparativa de la aleatoriedad de VCA

Tras analizar la tabla resumen, hemos observado la existencia de una serie de *endmembers* que se repiten en la mayoría de las pruebas realizadas. Este suceso parece obvio, ya que si realizamos varias repeticiones de la misma prueba, los *endmembers* resultantes – las firmas espectrales de los componentes de dicha imagen – deberían ser siempre los mismos. Por ello, otro aspecto a analizar dentro de la componente aleatoria de los distintos algoritmos es el orden en el que aparece cada índice.

A continuación se analizan en profundidad los cuatro casos en los que se han obtenido 20 *endmembers*. Esta elección se debe a que son los casos que presentan un mayor número de repeticiones y, en consecuencia, son los mejores casos para observar si la aparición de los distintos *endmembers* sigue un patrón concreto – en este caso se recoge únicamente el índice asociado a cada uno de ellos.

Con el fin de hacer el análisis de cada tabla de una manera más visual, se ha utilizado el código de colores que se muestra en la tabla 4.4.

Repeticiones	Color
10	
9	
8	
7	
6	
5	
4	
3	
2	
1	

Tabla 4.4. Escala de repeticiones para diez pruebas

El primer caso a estudiar es el que utiliza la *fractal_1* tras aplicarle una reducción a 25 bandas utilizando el algoritmo PCA. En las pruebas detalladas que se recogen en la tabla 4.5, podemos observar que los resultados de la parte superior de la tabla se corresponden o bien con los valores de repetición más altos – rojo, azul oscuro y verde claro – o bien con los valores más bajos de la escala – blanco y grises –. Este resultado se corresponde con la deducción obtenida de la tabla resumen: la mayoría de los *endmembers* calculados se repiten. Además, añade la particularidad de que dichas repeticiones se producen en las primeras iteraciones del algoritmo.

A continuación, en la tabla 4.6 se muestran los resultados obtenidos para la prueba con la *fractal_1* reducida a 100 bandas mediante PCA. A diferencia del caso anterior, observamos que los resultados con mayor número de repeticiones se concentran en la parte media y en la parte alta de la tabla. Esto puede deberse al exceso de información redundante al utilizar más bandas de las necesarias.

Si observamos en conjunto tanto la tabla 4.5 como la tabla 4.6, comprobamos que índices como el 4999, el 5275, el 82 o el 7200 están muy presentes en ambas tablas. Al estar analizando la misma *fractal* pero con una reducción PCA distinta, podemos concluir que el índice de los *endmembers* calculados se mantiene de una prueba a otra indicando, de esta manera, que las firmas espectrales de cada elemento de la imagen también se ordenan siguiendo un patrón.

Orden de *endmember*	Pruebas *HyperMix*					Pruebas RVC – CAL				
	1	2	3	4	5	1	2	3	4	5
1	3913	4999	82	3913	2828	2828	4999	2828	5275	82
2	4999	5275	5275	4999	4999	4999	5275	4999	82	6889
3	9585	9585	9585	9585	9585	9585	9585	9495	9585	5376
4	82	4212	4999	82	82	1451	82	7200	4999	4999
5	7200	1451	1451	5275	356	82	7200	158	7200	159
6	9495	9596	9495	9495	9495	7200	1451	9954	646	1451
7	9954	7200	7200	7200	5322	9954	9954	82	9399	7200
8	5376	9954	9954	9954	1451	9495	9298	9585	158	9954
9	460	360	158	158	9954	158	158	1450	9954	9585
10	2313	239	48	48	2313	48	48	2313	47	527
11	3260	26	822	26	43	833	28	26	242	47
12	239	48	2313	3262	721	1419	1419	536	26	42
13	43	1419	125	721	125	28	26	242	1419	822
14	426	2313	1419	125	242	242	242	1419	2313	2313
15	833	822	721	42	537	2313	822	426	125	28
16	536	43	536	342	536	125	833	47	28	537
17	47	1222	26	822	1419	26	42	328	42	833
18	1419	28	43	436	48	426	121	833	621	1419
19	26	242	426	536	833	42	426	125	833	328
20	722	722	833	1222	3260	239	125	42	822	124

Tabla 4.5. Caso fractal_1 *reducida con PCA a 25 bandas*

Orden de *endmember*	Pruebas *HyperMix*					Pruebas RVC – CAL				
	1	2	3	4	5	1	2	3	4	5
1	4999	3938	4999	1451	4999	4999	1451	82	5275	5275
2	5275	5275	9596	8818	5275	2828	8818	5275	82	4999
3	9585	9596	5275	4999	360	82	4999	9585	9585	9585
4	82	4999	1451	82	82	9585	82	4999	4999	82
5	9495	158	158	9495	9954	7200	9495	7200	7200	7200
6	5121	1450	6432	158	7200	9954	7200	158	9596	9954
7	1451	7200	7200	9954	9596	358	9954	9495	158	1451
8	358	9954	9954	7200	1450	1450	358	9954	9954	9495
9	9954	9585	9585	5376	9585	9399	5076	1450	1450	57
10	536	28	28	536	28	125	822	28	28	125
11	1222	125	2313	822	125	536	536	822	822	28
12	822	822	125	1317	822	822	3262	125	436	822
13	436	1419	822	833	2313	26	722	342	2313	242
14	42	537	26	426	26	1419	721	328	721	833
15	26	42	242	43	42	42	537	26	125	328
16	722	2313	43	26	1222	426	833	436	42	426
17	328	1222	1024	1419	1024	328	426	1419	47	47
18	1419	1317	1419	125	342	242	42	42	26	26
19	48	47	833	242	1419	47	47	833	342	1419
20	242	342	426	48	426	833	242	48	833	42

Tabla 4. 6 Caso fractal_1 *reducida con PCA a 100 bandas*

Con el fin de corroborar las conclusiones alcanzadas en el análisis detallado en las dos tablas anteriores, se ha realizado exactamente el mismo estudio para la *fractal_2*. Los resultados obtenidos aparecen en la tabla 4.7 para la reducción a 25 bandas y en la tabla 4.8 para el caso de una reducción a 100 bandas.

En primer lugar, en la figura 4.7 podemos observar que la conclusión anterior de que los valores más repetidos se acumulan en las primeras posiciones se cumple totalmente. En concreto, los índices que se repiten más a menudo en este caso son los siguientes: 4799, 799, 3199, 9963, 9931 y 483, con 10 repeticiones cada uno, y el valor 9996, con 9 repeticiones en total.

Orden de *endmember*	Pruebas *HyperMix*					Pruebas RVC – CAL				
	1	2	3	4	5	1	2	3	4	5
1	4799	799	9403	799	799	799	1613	1613	9403	1613
2	799	4799	1613	4799	4799	799	9403	9403	1613	3199
3	3199	1613	9963	1613	1416	1416	9963	9963	9963	9403
4	9963	3199	4799	9963	9963	9963	4799	4799	4799	799
5	1613	9502	9931	9931	9502	9502	799	9931	9931	9963
6	9931	9963	799	3199	9931	3199	3199	3199	799	9931
7	9502	9931	8679	9502	8679	9931	55	799	9996	4799
8	8679	253	9996	8679	3199	9996	9931	55	55	9996
9	9996	9495	3199	9996	9996	55	9996	9996	3199	55
10	483	384	432	384	384	483	384	2	110	483
11	110	483	384	334	1333	112	483	1333	1333	2
12	126	26	84	1333	110	95	112	84	334	1337
13	95	1333	483	112	126	110	84	115	483	26
14	383	112	1833	483	483	633	182	105	105	283
15	1333	115	383	110	95	14	583	483	26	105
16	731	334	114	95	305	383	110	114	112	84
17	105	182	26	731	1833	2	26	583	2	110
18	333	731	1333	432	383	84	283	26	583	583
19	2	333	283	333	583	583	105	305	182	1333
20	84	432	115	182	84	305	633	182	305	182

Tabla 4.7. Caso fractal_2 *reducida con PCA a 25 bandas*

Por otro lado, la tabla 4.8 corrobora que los índices más repetidos se distribuyen en las primeras posiciones ya que, en todas las pruebas, las seis primeras filas siempre tienen un color rojo –indicativo de que dicho índice se repite en las 10 pruebas realizadas –. Los índices que, en este caso, se repiten un mayor número de veces son los siguientes: 9403, 1613, 9963, 4799, 799, 9931, 9963, 3199 y 110.

Como podemos comprobar, tanto los índices más repetidos en la reducción a 25 bandas como en la reducción a 100 son prácticamente los mismos. Por ello, podemos confirmar la suposición que se hizo durante el estudio de la *fractal_1*: en la mayoría de las pruebas se repiten los mismos índices y, además, dichos índices aparecen en las primeras iteraciones del algoritmo VCA. Por ello, podemos concluir que los *endmembers* más significativos aparecen en las primeras iteraciones y, en consecuencia, debe existir un número óptimo de *endmembers* que nos proporcione la totalidad de la información asociada a la imagen analizada.

Orden de *endmember*	Pruebas *HyperMix*					Pruebas RVC – CAL				
	1	2	3	4	5	1	2	3	4	5
1	9403	799	1613	1613	4799	4799	4799	1613	1613	1613
2	1613	4799	3199	9403	799	799	799	799	9403	9403
3	9963	1613	9403	4799	9963	3199	9963	3199	4799	4799
4	4799	9963	4799	9963	1613	9963	1613	4799	9963	9963
5	799	3199	9963	3199	3199	1613	9403	9403	9931	799
6	9931	9403	799	9931	9403	9403	9931	9963	3199	9931
7	8679	9931	9931	8679	9931	9931	9996	9931	799	3199
8	9996	253	9996	9996	9996	9996	3199	156	8679	9996
9	3199	9996	8679	799	156	156	57	9996	9996	156
10	110	110	110	110	110	110	583	110	2	105
11	384	384	333	731	283	2	26	2	110	110
12	333	731	731	384	483	333	105	84	283	2
13	831	333	384	483	583	583	283	333	483	333
14	1333	1333	432	633	334	432	182	731	84	483
15	84	831	334	112	432	112	110	105	53	182
16	182	383	483	333	182	483	2	182	26	84
17	731	1833	95	432	1333	334	483	112	583	1527
18	432	112	1333	95	333	1333	333	483	182	633
19	112	334	182	334	95	182	633	26	105	112
20	95	26	112	182	2	26	334	633	633	26

Tabla 4.8. Caso fractal_2 *reducida con PCA a 100 bandas*

Tras realizar el análisis de la bondad del algoritmo y ver que arroja resultados satisfactorios, realizamos una comparativa de los *endmembers* extraídos y observamos que, pasada una banda concreta del PCA, todos los *endmembers* tenían el mismo valor. Un ejemplo gráfico se observa en la figura 4.2 en la que se muestran los cinco *endmembers* calculados para la reducción a 25 bandas de la *fractal_2*.

Como se puede observar en dicha figura, más de un tercio de la firma espectral de los cinco *endmembers* es común.

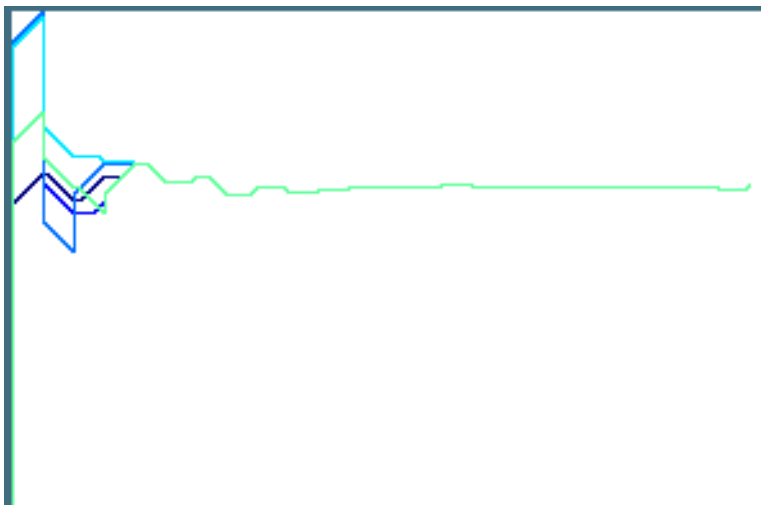


Fig. 4.2. Representación gráfica de 5 endmembers *calculados para la* fractal_2 *reducida a 25 bandas*

Tras observar este suceso, se comprobó el comportamiento del cálculo de *endmembers* al reducir el número de bandas a calcular. En la figura 4.3 se aporta una gráfica que muestra en detalle las bandas para las que difieren los valores de los distintos *endmembers*. Cabe mencionar que para dicho cálculo se ha realizado una reducción a siete bandas.

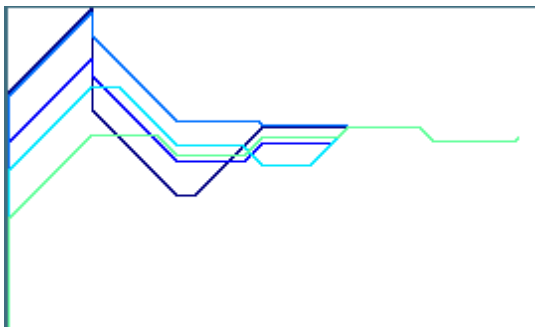


Fig. 4.3. Representación gráfica de 5 endmembers *calculados para la* fractal_2 *reducida a 7 bandas*

Por último, utilizando el resultado obtenido por Raquel Lazcano en [1] al analizar la *fractal_2* mediante el algoritmo HySime de la etapa de *estimación de* endmembers, se realizó el cálculo óptimo de *endmembers* para dicha *fractal*. En la figura 4.4 se muestra el resultado de extraer 6 *endmembers* para la *fractal_2* en su versión sin ruido.

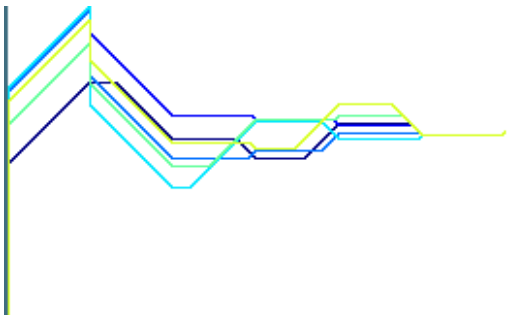


Fig. 4.4. Representación gráfica de 6 endmembers *calculados para la* fractal_2 *reducida a 7 bandas*

Tras realizar dicho análisis se observó que, utilizando el número óptimo de *endmembers* a calcular, el valor de la banda 7 para todos los *endmembers* coincide. Por ello, la conclusión global de dicho análisis es que el valor obtenido durante la etapa de *estimación de* endmembers nos aporta tanto el número de bandas a calcular como el número de *endmembers* que debemos extraer para tener la información completa asociada a la imagen original. En la figura 4.5 se aporta la representación gráfica de este último caso.

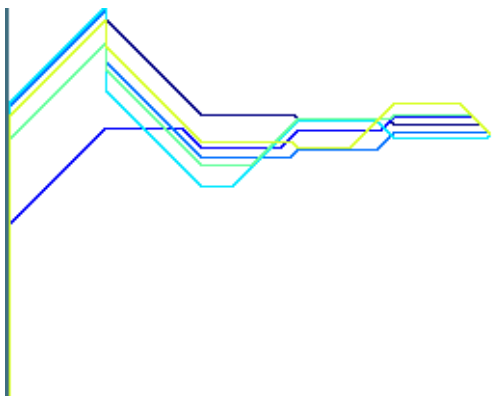


Fig. 4.5. Representación gráfica de 6 endmembers *calculados para la* fractal_2 *reducida a 6 bandas*

Para finalizar el análisis de la etapa de *extracción de* endmembers, se ha comparado el tiempo de ejecución del programa desarrollado utilizando la librería RVC – CAL con el tiempo de la herramienta *HyperMix*. Para ello, se han realizado diez pruebas en cada programa para cuatro casos distintos. La elección de los casos ha sido: utilizando la *fractal_2,* calcular el número máximo y mínimo de *endmembers* para el número máximo y mínimo de bandas calculadas. Un resumen de los resultados obtenidos se recoge en la tabla 4.9.

Como podemos observar en dicha tabla, los tiempos obtenidos utilizando la librería desarrollada a lo largo del presente Proyecto Fin de Grado son menores que los obtenidos mediante *HyperMix.* Concretamente, en el caso de la reducción a 25 bandas, se reduce el tiempo de ejecución en un 50% y, en el caso de las 100 bandas, la mejora oscila entre un 15 y un 25%.

Tras analizar en profundidad la comparativa de tiempos para el algoritmo VCA reflejada en la tabla 4.9, se ha observado que, a diferencia de los tiempos de ejecución de la herramienta *HyperMix*, los tiempos de la aplicación desarrollada para RVC – CAL son prácticamente constantes. Esto nos permite planificar una división en núcleos de manera más segura ya que, si ejecutamos la aplicación bajo los mismos parámetros de manera continuada, sabremos casi con exactitud el tiempo que tardará esta fase en completarse.

Prueba			***HyperMix***	**RVC – CAL**
25 bandas	5 *endmembers*	Mínimo	0.140558	0.083729
		Media	0.1797986	0.0844941
		Máximo	0.188166	0.085542
	20 *endmembers*	Mínimo	0.318507	0.17249
		Media	0.3449568	0.1749116
		Máximo	0.376212	0.181685
100 bandas	5 *endmembers*	Mínimo	1.54422	1.336594
		Media	1.591868	1.3410811
		Máximo	1.62837	1.347199
	20 *endmembers*	Mínimo	2.0143	1.587207
		Media	2.070461	1.6002695
		Máximo	2.13395	1.640191

Tabla 4.9. Comparativa de tiempos VCA – las medidas se dan en segundos

Como ocurría en el caso del análisis del algoritmo PCA, durante el estudio del comportamiento del VCA se ha observado que los resultados obtenidos utilizando la librería desarrollada son equivalentes a los obtenidos utilizando *HyperMix,* y que el tiempo de ejecución necesario es sustancialmente menor incluso sin utilizar división en procesadores.

4.2. Análisis de la cadena completa

Tras realizar el análisis individual de las distintas fases de la cadena, se unificó la librería desarrollada durante el presente Proyecto Fin de Grado con la implementada en [1] por Raquel Lazcano. Tras esto, se codificaron dos configuraciones distintas de la cadena completa y se analizó la bondad y el tiempo de procesado de cada una por separado.

En primer lugar se elaboró una cadena completa implementada en un único actor, con el fin de detectar posibles incompatibilidades entre etapas. Este modelo de procesado se analiza en profundidad a continuación.

Por otro lado, se implementó una cadena en la que cada etapa estaba dividida en un actor diferente. En ella, se ha analizado la influencia de la comunicación entre actores, así como la viabilidad de dividir en procesadores dichas etapas. Los resultados arrojados utilizando esta metodología están recogidos en [1].

En consecuencia, con el objetivo de probar la bondad de los resultados de la cadena completa en un único actor, se han dividido las pruebas realizadas en dos bloques:

- En el primero de ellos, utilizando la *fractal_1*, tal y como se llevó a cabo en el análisis del algoritmo VCA, se ha omitido la componente aleatoria para comprobar si varias ejecuciones bajo los mismos parámetros arrojan el mismo resultado y, a su vez, observar si existen diferencias respecto al comportamiento individual por etapas.

- Por el contrario, en el segundo caso se ha utilizado la *fractal_2* para probar el funcionamiento normal de la cadena completa y ver si los *endmembers* calculados y los mapas de abundancia generados coinciden con los obtenidos en el análisis individual por etapas.

Las pruebas realizadas durante el primer bloque se resumen en dos: el análisis de un caso estándar – reducción PCA a 25 bandas y cálculo de 15 *endmembers* – y el estudio del caso óptimo. En él se han utilizando los parámetros obtenidos en la etapa de configuración utilizando el algoritmo HySime: tanto el número de bandas a reducir como el número de *endmembers* a calcular es 6.

Este bloque ha sido analizado utilizando únicamente el programa *Meld Diff Viewer* explicado anteriormente. Al utilizar dicho programa, se ha comparado la salida de cada una de las distintas fases – imagen reducida para el PCA, *endmembers* calculados para el VCA y mapas de abundancias para el LSU (analizado individualmente por Raquel Lazcano en [1]) – y los resultados obtenidos han sido idénticos para las cinco repeticiones realizadas, tanto en el caso estándar como en el caso óptimo.

Una vez realizado el análisis del primer bloque de pruebas para la cadena completa, podemos concluir que no existen incompatibilidades entre los algoritmos escogidos para las distintas etapas.

A modo ilustrativo, en la tabla 4.10 se aporta el resultado obtenido para el caso óptimo de la *fractal_1*. En ella, podemos observar los mapas de abundancias que se corresponden con los seis *endmembers* calculados – el color blanco se asocia con una coincidencia absoluta, mientras que el color negro indica que en esa zona de la imagen no se encuentra dicho elemento.

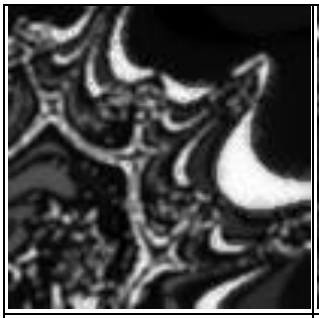
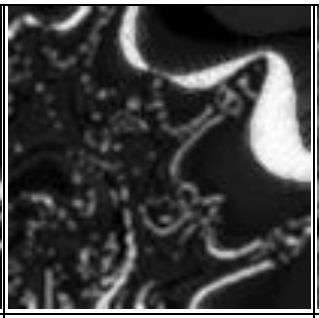
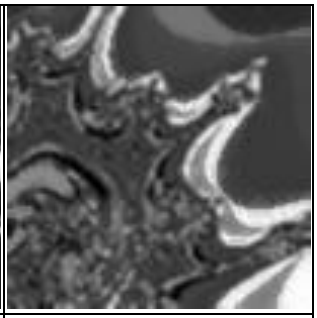
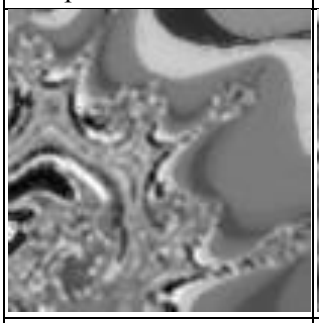
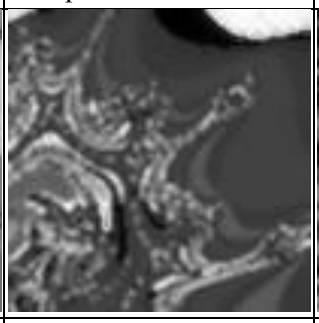
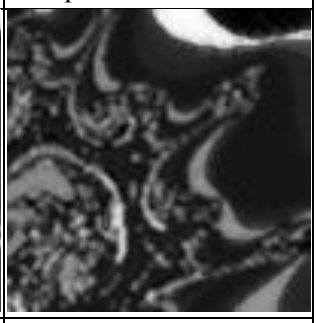
		
Mapa de abundancia 1	Mapa de abundancia 2	Mapa de abundancia 3
		
Mapa de abundancia 4	Mapa de abundancia 5	Mapa de abundancia 6

Tabla 4.10. Mapas de abundancias para el caso óptimo de la fractal_1

Tras esto, se realizó el segundo bloque de pruebas, en el que se incluye la parte aleatoria que se omitió anteriormente. Este bloque, a su vez, contiene dos pruebas diferenciadas: la primera consiste en el análisis de un caso estándar, en el cual se redujo la *fractal_2* a 25 bandas y, posteriormente, se calcularon un total de 10 *endmembers*; por otra parte, se realizó el estudio del caso óptimo, que consiste en reducir la imagen a 6 bandas y calcular 6 *endmembers* – resultado arrojado por la etapa de *estimación de* endmembers analizada por Raquel Lazcano en [1].

Para cada una de las pruebas se realizaron cinco repeticiones, en las cuales se almacenó la salida de cada etapa. Con el fin de analizar si la aleatoriedad del sistema funciona correctamente, se reunieron los índices obtenidos en el algoritmo VCA – dichos índices, como se ha explicado anteriormente, se corresponden con los distintos *endmembers* finales.

En primer lugar, se analizó la salida del algoritmo PCA para ambos casos, comparando los ficheros generados en cada una de las pruebas. Para ello, como se ha realizado durante el desarrollo de la totalidad del presente capítulo, se ha utilizado la herramienta software *Meld Diff Viewer*. Dicha herramienta ha lanzado resultados satisfactorios, en los que se comprueba que las cinco repeticiones de cada uno de los casos eran idénticas entre sí.

Para facilitar el análisis de los resultados obtenidos en el algoritmo VCA, tal y como se hizo durante el estudio de la fase de *extracción de* endmembers, se ha utilizado una escala de colores. Como en este caso únicamente hay cinco pruebas, la escala de colores se ha reducido a la mitad con respecto a la utilizada anteriormente. Dicha escala se muestra en la tabla 4.11.

A continuación, la tabla 4.12 muestra los resultados obtenidos durante el estudio del caso estándar mientras que, por otro lado, en la tabla 4.13 se recogen los obtenidos para el caso óptimo.

Como podemos observar, todos los *endmembers* obtenidos se repiten en casi la totalidad de las pruebas realizadas. A su vez, si comparamos los resultados obtenidos con la prueba individualizada para la *fractal_2* del apartado anterior, podemos corroborar que los índices obtenidos son los mismos que los que se obtienen al lanzar el algoritmo VCA de manera independiente.

Repeticiones	Color
5	
4	
3	
2	
1	

Tabla 4.11. Escala de repeticiones para cinco pruebas

Orden de *endmember*	Prueba 1	Prueba 2	Prueba 3	Prueba 4	Prueba 5
1	4799	799	799	1613	4799
2	799	4799	4799	9403	799
3	3199	1613	1613	9963	3199
4	1613	9963	3199	4799	8679
5	9963	9403	9963	799	9963
6	9403	9931	9403	9931	9931
7	9931	8679	9931	3199	1613
8	8679	3199	8679	9996	9996
9	9996	9996	9996	55	9502
10	182	182	182	182	182

Tabla 4.12. Comparativa de endmembers *para el caso estándar de la cadena completa – 1 actor*

Orden de *endmember*	Prueba 1	Prueba 2	Prueba 3	Prueba 4	Prueba 5
1	1613	9403	9403	4799	799
2	9403	1613	1613	9931	4799
3	9931	3199	9931	799	1613
4	4799	4799	4799	1613	9931
5	9963	9931	9963	9963	3199
6	3199	9963	3199	3199	9963

Tabla 4.13. Comparativa de endmembers *para el caso óptimo de la cadena completa – 1 actor*

Por último se analizó el comportamiento de la cadena completa para la generación de los mapas de abundancias correspondientes a los distintos *endmembers* finales. Como podemos observar en las tablas 4.12 y 4.13, los *endmembers* suelen ser los mismos en todas las pruebas, pero ordenados de distinta manera. Para comprobar si los resultados arrojados por la última etapa de la cadena son correctos, se compararon los ficheros generados utilizando el programa *Meld Diff Viewer* y se observó que el mapa asociado a cada índice es siempre el mismo.

A modo de ejemplo, en la tabla 4.14 se aportan los mapas de abundancias obtenidos para las pruebas 1 y 2 del caso óptimo. A continuación se aporta un resumen de las equivalencias entre los mapas de la primera y de la segunda prueba, que recoge las relaciones existentes entre los mapas mostrados en la tabla 4.14, mencionada anteriormente:

- El primer mapa de la primera prueba es equivalente al segundo mapa de la segunda prueba.
- El segundo de la primera coincide con el primero de la segunda.
- El tercer mapa de la primera prueba equivale al quinto de la segunda.
- Los cuartos mapas son idénticos entre sí.
- El quinto mapa generado en la primera prueba se corresponde con el sexto generado en la segunda.
- Por último, el sexto mapa de abundancias de la primera prueba se identifica con el tercero de la segunda.

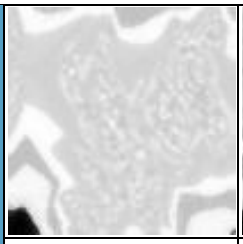
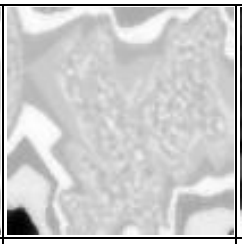
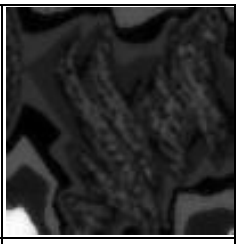
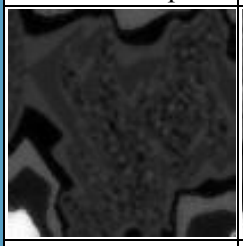
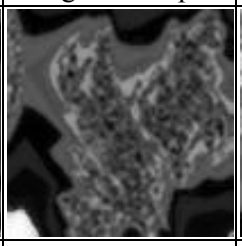
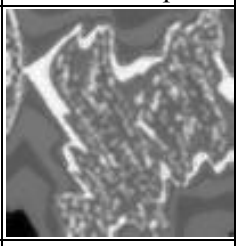
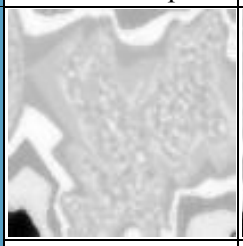
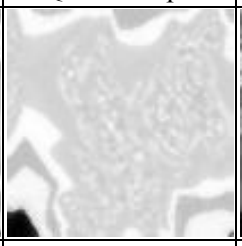
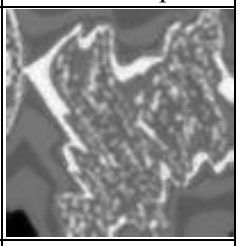
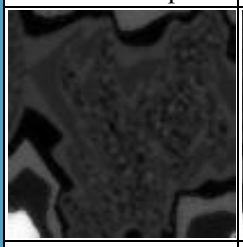
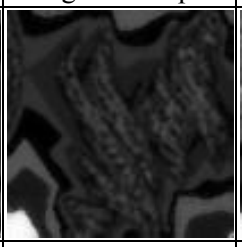
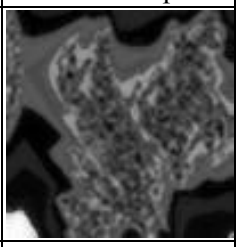
Primera prueba			
	Primer mapa	Segundo mapa	Tercer mapa
			
	Cuarto mapa	Quinto mapa	Sexto mapa
Segunda Prueba			
	Primer mapa	Segundo mapa	Tercer mapa
			
	Cuarto mapa	Quinto mapa	Sexto mapa

Tabla 4.14. Comparativa de mapas de abundancia para el caso óptimo de la cadena completa – 1 actor

Tras comprobar que el funcionamiento de la cadena completa es el esperado – se corresponde con los resultados obtenidos durante el análisis individual de etapas – se realizó un estudio de la velocidad de procesamiento total que se conseguía alcanzar.

Dicho estudio se realizó para los casos extremos utilizados durante todo el desarrollo del presente Proyecto Fin de Grado:

- El primero de ellos supone el caso estándar más rápido de todos los estudiados. Utilizando la *fractal_2,* se realiza una reducción a 25 bandas y se calcula un total de 5 *endmembers* para, posteriormente, obtener los mapas de abundancias asociados a los mismos.

- Tras esto, se estudió el caso contrario, es decir, el que requiere un mayor trabajo computacional. Utilizando también la *fractal_2,* se aplica PCA para reducir la imagen original a 100 bandas y, a continuación, se realiza el cálculo de 20 *endmembers* y de los mapas de abundancias correspondientes.

Los resultados de los dos casos mencionados se recogen en la tabla 4.15. En ella se comparan los resultados obtenidos de la cadena completa con la suma por etapas en cada caso. Los valores utilizados para obtener dicha suma son los mostrados en el análisis individual de los algoritmos PCA y VCA descritos en este documento y del algoritmo LSU estudiado por Raquel Lazcano en [1]. Como podemos observar, la cadena completa se ejecuta en menos tiempo que la ejecución por etapas. Esto se debe a que existen ciertas operaciones que son compartidas por las distintas etapas y, en consecuencia, al implementar todo en un único actor se ha optado por eliminar las operaciones redundantes ahorrando, de esta manera, entre 0.3 y 0.6 segundos en cada ejecución.

Prueba			Cadena completa	Suma por etapas
25 bandas	5 *endmembers*	Mínimo	10.571357	10.8738143
		Media	10.5899674	10.89787353
		Máximo	10.617797	10.9245685
100 bandas	20 *endmembers*	Mínimo	12.231839	12.598229
		Media	12.2554437	12.6566602
		Máximo	12.285597	12.818484

Tabla 4.15. Comparativa de tiempos de la cadena completa en 1 actor – las medidas se dan en segundos

Para finalizar el análisis de los tiempos de la cadena completa únicamente queda por medir el tiempo de ejecución para el caso óptimo. En la tabla 4.16 se recogen los resultados obtenidos.

Prueba		Tiempo
Caso óptimo	Mínimo	10.509184
	Media	10.5136651
	Máximo	10.525474

Tabla 4. 16 Tiempos de ejecución del caso óptimo de la cadena completa en 1 actor

4.3. Análisis de rendimiento

Para finalizar el análisis de los resultados obtenidos se analizó el consumo de recursos de la cadena completa para el caso óptimo de la *fractal_2*. Con el fin de hacer este estudio más completo, se dividió en dos partes:

- La primera de ellas, recogida en [1], se trata de un análisis base, es decir, partimos de los casos en los que se ha implementado la cadena en un actor y en varios actores, pero ejecutándolo en un único procesador.

- En segundo lugar, tras analizar los resultados obtenidos por Raquel Lazcano en [1] durante el estudio del tiempo de ejecución del sistema para las distintas configuraciones posibles de división en núcleos, se decidió incluir en este apartado un estudio del consumo de recursos para el caso más rápido y el más lento en cuanto a velocidad de ejecución de las distintas distribuciones. Esta parte se recoge en el presente Proyecto Fin de Grado.

Para realizar el análisis descrito a continuación se ha utilizado la herramienta *papify,* explicada anteriormente. A su vez, con el objetivo de analizar los resultados de una manera más rápida, se ha empleado una nueva herramienta – también desarrollada por Alejo Iván Arias – denominada *papiplot*[14]. Dicha herramienta analiza los archivos generados por *papify* y, teniendo en cuenta dichos datos, genera tablas resumen y gráficas de barras en las que se recogen los recursos consumidos para cada actor dentro del sistema y, a su vez, para cada acción dentro de cada actor.

En la tabla 4.17 se observan las distintas distribuciones estudiadas por Raquel Lazcano en [1]. De ellas, la más rápida es la número 4, con un tiempo de 13,6958 segundos; en esta distribución, las etapas S*ource*, PCA, LSU y *Display* están en el procesador 1 y la etapa VCA está en el procesador 2. Por otro lado, la distribución número 10 es la más lenta de las analizadas, con un tiempo de 14,1114 segundos; en este caso, las etapas *Source* y *Display* se encuentran en el procesador 1 y las etapas PCA, VCA y LSU en el procesador 2.

En comparativa con el tiempo de procesado de 13.5456 segundos obtenido en la prueba base (distribución 1 de la tabla 4.17, en la que todos los actores se ejecutan en el mismo procesador), el retardo asociado a cada una de las etapas mencionadas anteriormente es de 0.15 segundos para el caso más rápido y de 0.56 para el caso más lento.

Actores	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Source	0	1	0	0	0	0	1	1	1	1	0	0	0	0	0	0
PCA	0	0	1	0	0	0	1	0	0	0	1	1	1	0	0	0
VCA	0	0	0	1	0	0	0	1	0	0	1	0	0	1	1	0
LSU	0	0	0	0	1	0	0	0	1	0	0	1	0	1	0	1
Display	0	0	0	0	0	1	0	0	0	1	0	0	1	0	1	1
	0.- procesador 1										*1.- procesador 2*					

Tabla 4. 17 División en procesadores

[14] https://github.com/alejoar/papiplot

Durante el análisis del consumo de recursos de estos dos casos, se han monitorizado los siguientes registros PAPI:

- *PAPI_TOT_INS*: representa el número total de instrucciones que se ejecutan en el procesador.
- *PAPI_BR_INS*: almacena el número de instrucciones condicionantes – *if*, *else*, etc – presentes en el programa
- *PAPI_L1_DCM*: Accesos fallidos a memoria caché de datos.
- *PAPI_L1_ICM*: Accesos fallidos a memoria caché de instrucciones.

El análisis de la variación del consumo de recursos con respecto a la distribución de los actores entre los distintos procesadores se va a realizar en dos pasos: en el primero de ellos se realiza una comparativa entre el caso más rápido y el caso más lento y, a continuación, ambos casos se comparan con los resultados obtenidos para el caso base de 5 actores trabajando en un mismo procesador. De esta manera, podemos obtener una idea general de las acciones dentro de un actor, y de los actores dentro del sistema, que más se ven afectados por la división en procesadores. A su vez, se comprobará si existe una regla general asociada al retardo en la velocidad de procesamiento al utilizar varios núcleos en paralelo.

Para hacer más sencillo el seguimiento del análisis de rendimiento, se estudiará cada actor por separado y, por último, se realizará un análisis general del funcionamiento del sistema completo.

La primera etapa que se va a analizar se denomina *Source*. En esta fase se lee el archivo binario que contiene la información de la *fractal* y, conforme se lee dicho fichero, se envía la información a la etapa de *reducción dimensional* donde se implementa el algoritmo PCA.

A continuación, se muestran dos tablas que recogen la información para el caso más rápido – tabla 4.18 – y para el caso más lento – tabla 4.19 –. En ellas podemos observar que esta etapa se encuentra dividida en 6 acciones distintas – dichas acciones se relacionan con la apertura y cierre del fichero, su lectura y su envío – y que la gran mayoría de los recursos se consumen durante la lectura del fichero fuente.

Como podemos observar en dichas tablas, el consumo de recursos relacionado con esta fase es muy parejo, pero en el caso más rápido está un poco por debajo. Esto se debe a que es la primera etapa y ambas configuraciones parten del fichero de entrada que está almacenado en disco.

Acción	TOT_INS	BR_INS	L1_DCM	L1_ICM
getFileSize	2539	631	137	57
readNBytes	1023884463	237578988	705050	788167
sendData_launch_aligned	25572328	1840862	235612	147152
sendData_done	2550260	966816	121282	221539
sendData_launch	20371523	1398181	181865	112967
closeFile	748	241	54	70
Total	1072381861	241785719	1244000	1269952

Tabla 4.18. Consumo de recursos del actor Source *caso más rápido*

Acción	TOT_INS	BR_INS	L1_DCM	L1_ICM
getFileSize	2539	651	143	56
readNBytes	1023884709	237585380	765030	751272
sendData_launch_aligned	25572356	1838827	254671	175089
sendData_done	2550259	965361	135712	209132
sendData_launch	20371549	1395230	198223	135964
closeFile	748	238	49	63
Total	1072382160	241785687	1353828	1271576

Tabla 4.19. Consumo de recursos del actor Source *caso más lento*

A continuación, en la tabla 4.20 se muestran los valores asociados al caso en el que todos los actores se ejecutan en el mismo procesador. Como podemos observar, no existen grandes diferencias en el consumo de recursos de esta etapa. Esto sucede, como hemos mencionado con anterioridad, debido a que, en la primera etapa, las tres situaciones parten de la lectura de un fichero almacenado en disco.

Acción	TOT_INS	BR_INS	L1_DCM	L1_ICM
getFileSize	2539	625	135	57
readNBytes	1023887822	237552479	1158138	926800
sendData_launch_aligned	25572319	1816167	254667	171726
sendData_done	2550257	930727	146782	216246
sendData_launch	20371523	1379406	194891	141375
closeFile	748	236	52	74
Total	1072385208	241679640	1754665	1456278

Tabla 4.20. Consumo de recursos del actor Source *caso 5 actores – 1 procesador*

Tras analizar la etapa de entrada al sistema, la siguiente fase a analizar es la de *reducción dimensional,* en la que se ha implementado el algoritmo PCA. Esta fase recibe los datos del actor *Source* y, tras ejecutar el algoritmo correspondiente, envía la imagen reducida a los actores en los que se han implementado las etapas de *extracción de* endmembers y *estimación de abundancias*.

El actor PCA se divide en cinco acciones distintas – relacionadas con la recepción, el algoritmo PCA y el envío de la imagen reducida –. En las tablas 4.21 y 4.22 se reúne la información relacionada con el consumo de recursos para el caso más rápido y para el caso más lento, respectivamente.

Acción	TOT_INS	BR_INS	L1_DCM	L1_ICM
receive_data_aligned	59512511	5639974	436573	161516
receive_data	46145888	4324280	341095	139625
PCA_alogrithm	28093897043	1031968697	656351001	9154
send_aligned	6384429	1325695	189322	267225
send	76285	15879	2163	3088
Total	28206016156	1043274522	657320154	580608

Tabla 4.21. Consumo de recursos del actor PCA caso más rápido

Acción	TOT_INS	BR_INS	L1_DCM	L1_ICM
receive_data_aligned	59512420	5631990	461525	182604
receive_data	46145837	4318064	362392	147361
PCA_alogrithm	28093897549	1031948688	656692643	25828
send_aligned	6384478	1326564	211139	267405
send	76285	15932	2602	3124
Total	28206016569	1043241238	657730301	626322

Tabla 4.22. Consumo de recursos del actor PCA caso más lento

Esta etapa concentra la mayor parte de los recursos consumidos durante la ejecución del sistema. Como podemos observar, no existe una gran diferencia en el consumo de recursos en ambos casos pero, como ocurría en la etapa anterior, el caso más rápido está por debajo que el caso más lento.

A continuación, en la tabla 4.23 se observa el consumo de recursos del caso base de este estudio. En él observamos que dicho consumo también es muy parejo pero, en el caso más rápido con división en dos núcleos, los accesos a memoria de instrucciones fallidos son bastante más bajos.

Acción	TOT_INS	BR_INS	L1_DCM	L1_ICM
receive_data_aligned	59512372	5609692	468261	182683
receive_data	46145799	4299645	359797	146885
PCA_alogrithm	28093897526	1031936052	656651346	25933
send_aligned	6384421	1292473	211297	273818
send	76285	15336	2476	3070
Total	28206016403	1043153198	657693177	632389

Tabla 4.23. Consumo de recursos del actor PCA caso 5 actores – 1 procesador

La tercera etapa implicada en el análisis de consumo de recursos es la de *extracción de* endmembers, donde se ha implementado el algoritmo VCA. En esta etapa se recibe la imagen reducida desde el actor *PCA* y, posteriormente, se envían los *endmembers* calculados al actor en el que se implementa el algoritmo LSU, propio de la fase de *estimación de abundancias*. Las tablas 4.24 y 4.25 muestran, respectivamente, la utilización de los recursos para los casos más rápido y más lento.

En este caso, el consumo de recursos también es similar e incluso el caso más lento tiene un mejor rendimiento en cuanto a accesos a memoria caché fallidos se refiere pero, al ser una etapa que tiene una ejecución mucho más rápida que la anterior, su aportación al total de recursos consumidos es, en proporción, mucho menor que la aportación del algoritmo PCA.

Acción	TOT_INS	BR_INS	L1_DCM	L1_ICM
receive_data_aligned	5732262	1242409	241394	263896
receive_data	68562	14903	2748	3525
VCA_algorithm	57821758	3708441	613830	2972
send_aligned	3876	815	136	194
Total	63626458	4966568	858108	270587

Tabla 4.24. Consumo de recursos del actor VCA caso más rápido

Acción	TOT_INS	BR_INS	L1_DCM	L1_ICM
receive_data_aligned	5732283	1241992	232785	251495
receive_data	68563	14980	2683	3305
VCA_algorithm	57814935	3707376	614510	2873
send_aligned	3876	826	122	188
Total	63619657	4965174	850100	257861

Tabla 4.25. Consumo de recursos del actor VCA caso más lento

Como podemos observar en la tabla 4.26, el consumo de recursos del caso en el que los cinco actores se ejecutan en un mismo procesador sigue la misma línea que en las etapas anteriores: los resultados arrojados son muy similares a los obtenidos con división de procesadores.

Acción	TOT_INS	BR_INS	L1_DCM	L1_ICM
receive_data_aligned	5732257	1206850	253096	262203
receive_data	68562	14383	2975	3381
VCA_algorithm	57817580	3707064	613728	3091
send_aligned	3876	796	141	194
Total	63622275	4929093	869940	268869

Tabla 4.26. *Consumo de recursos del actor VCA caso 5 actores – 1 procesador*

En la fase final del procesamiento de una imagen hiperespectral – *estimación de abundancias* – se ha implementado el algoritmo LSU. Dicho algoritmo tiene por entrada tanto la imagen reducida del PCA como los *endmembers* calculados en el VCA; a su vez, la salida – los mapas de abundancias calculados – se envía al actor *Display*. Las tablas 4.27 y 4.28 reflejan el consumo de recursos de esta etapa asociado al caso más rápido y más lento.

Acción	TOT_INS	BR_INS	L1_DCM	L1_ICM
receive_data_aligned	5564254	1238635	325737	264713
receive_data	66574	14736	3796	3347
receive_endmembers_aligned	3480	755	206	179
LSU_algorithm	11415633	555928	46526	619
send_abundances_aligned	6493137	1318862	330969	275687
send_abundances	77571	15963	4068	3275
Total	23620649	3144879	711302	547820

Tabla 4.27. Consumo de recursos del actor LSU caso más rápido

Acción	TOT_INS	BR_INS	L1_DCM	L1_ICM
receive_data_aligned	5564259	1237674	325327	272779
receive_data	66573	14874	3726	3230
receive_endmembers_aligned	3480	751	178	176
LSU_algorithm	11411702	555499	47007	592
send_abundances_aligned	6493134	1315278	286909	286507
send_abundances	77571	16722	3468	3548
Total	23616719	3140798	666615	566832

Tabla 4.28. Consumo de recursos del actor LSU caso más lento

De nuevo, observamos que los recursos consumidos en ambos casos son muy similares. De hecho, en esta etapa se compensan los accesos fallidos a memoria caché de datos del caso más rápido con los accesos fallidos a memoria caché de instrucciones del otro.

Como se ha realizado en el resto de etapas, en la tabla 4.29 se aportan los datos obtenidos para el caso base; en ella observamos que los recursos consumidos por este caso son equivalentes a los consumidos cuando se utilizan varios procesadores.

Acción	TOT_INS	BR_INS	L1_DCM	L1_ICM
receive_data_aligned	5564246	1206714	329801	287004
receive_data	66573	14281	3749	3260
receive_endmembers_aligned	3480	745	182	180
LSU_algorithm	11409615	555265	46652	576
send_abundances_aligned	6493135	1295194	302306	297375
send_abundances	77571	15989	3704	3521
Total	23614620	3088188	686394	591916

Tabla 4.29. Consumo de recursos actor LSU caso 5 actores – 1 procesador

En último lugar tenemos el actor *Display*. En esta etapa, se reciben los mapas de abundancias generados en el actor LSU y se genera el fichero binario junto a la cabecera correspondiente.

En la tabla 4.30 se incluyen los recursos consumidos por el caso más rápido, y la tabla 4.31 representa los recursos utilizados por el caso más lento. Si los comparamos, observamos que vuelven a ser muy parejos y que difieren principalmente en los accesos a memoria caché.

Acción	TOT_INS	BR_INS	L1_DCM	L1_ICM
receive_data_aligned	169524209	35543400	1579900	9976809
receive_data	67158	14883	3681	3195
Total	169591367	35558283	1583581	9980004

Tabla 4.30. Consumo de recursos del actor Display *caso más rápido*

Tabla 4.30. Consumo de recursos del actor Display *caso más rápido*

Acción	TOT_INS	BR_INS	L1_DCM	L1_ICM
receive_data_aligned	170411346	35785059	1178980	10104915
receive_data	67158	14936	3773	3519
Total	170478504	35799995	1182753	10108434

Tabla 4.31. Consumo de recursos del actor Display *caso más lento*

Como se observa en la tabla 4.32, los recursos asociados a esta etapa propios del caso base son, de nuevo, muy similares a los obtenidos utilizando división en procesadores.

Acción	TOT_INS	BR_INS	L1_DCM	L1_ICM
receive_data_aligned	170347958	35843214	1205069	10218092
receive_data	67158	14630	3844	3440
Total	170415116	35857844	1208913	10221532

Tabla 4.32. Consumo de recursos del actor Display *caso 5 actores – 1 procesador*

A modo de resumen, las tablas 4.33 y 4.34 recogen los recursos totales consumidos por cada actor y, en la última línea, el consumo total del sistema para cada uno de los recursos monitorizados. Como podemos observar si comparamos ambas tablas, los recursos totales monitorizados en ambos casos son casi equivalentes; únicamente difieren en los accesos a memoria caché.

Actor	TOT_INS	BR_INS	L1_DCM	L1_ICM
Source	1072381861	241785719	1244000	1269952
PCA	28206016156	1043274522	657320154	580608
VCA	63626458	4966568	858108	270587
LSU	23620649	3144879	711302	547820
Display	169591367	35558283	1583581	9980004
Total	29535236491	1328729971	661717145	12648971

Tabla 4. 33 Consumo de recursos total del caso más rápido

Actor	TOT_INS	BR_INS	L1_DCM	L1_ICM
Source	1072382160	241785687	1353828	1271576
PCA	28206016569	1043241238	657730301	626322
VCA	63619657	4965174	850100	257861
LSU	23616719	3140798	666615	566832
Display	170478504	35799995	1182753	10108434
Total	29536113609	1328932892	661783597	1271576

Tabla 4. 34 Consumo de recursos total del caso más lento

Por último, si comparamos los resultados obtenidos con el caso base, observamos que los resultados obtenidos en cuanto a número de instrucciones se mantienen totalmente estables; esto es lógico si se tiene en cuenta que el programa sigue siendo el mismo, la codificación no ha variado y sólo cambia el lugar donde se ejecuta cada etapa de la cadena.

Por otro lado, si observamos los accesos a memoria caché, vemos una sutil mejora cuando ejecutamos la aplicación en varios procesadores; esto puede deberse a que, si ejecutamos todo en un mismo procesador, no tenemos memoria caché suficiente para ejecutar la totalidad de la cadena de procesado.

Actor	TOT_INS	BR_INS	L1_DCM	L1_ICM
Source	1072385208	241679640	1754665	1456278
PCA	28206016403	1043153198	657693177	632389
VCA	63622275	4929093	869940	268869
LSU	23614620	3088188	686394	591916
Display	170415116	35857844	1208913	10221532
Total	29536053622	1328707963	662213089	13170984

Tabla 4. 35 Consumo de recursos total para el caso 5 actores – 1 procesador

Ya que el consumo de recursos para los tres casos es muy similar, en las figuras 4.6 y 4.7 se muestran dos gráficas representativas del consumo de cada actor para el caso más rápido con división de procesadores.

Si analizamos dichas figuras – y teniendo en cuenta que se representan utilizando una escala logarítmica – podemos concluir que existe un cuello de botella en el actor PCA. Dicho actor consume más de diez veces los recursos consumidos por cualquiera de los otros actores. Este suceso hace que nos demos cuenta rápidamente de la necesidad de mejorar u optimizar el código del algoritmo PCA con el fin de aumentar su velocidad de ejecución – conviene recordar que también era el algoritmo más lento de los tres que componen la cadena – y reducir su consumo de recursos.

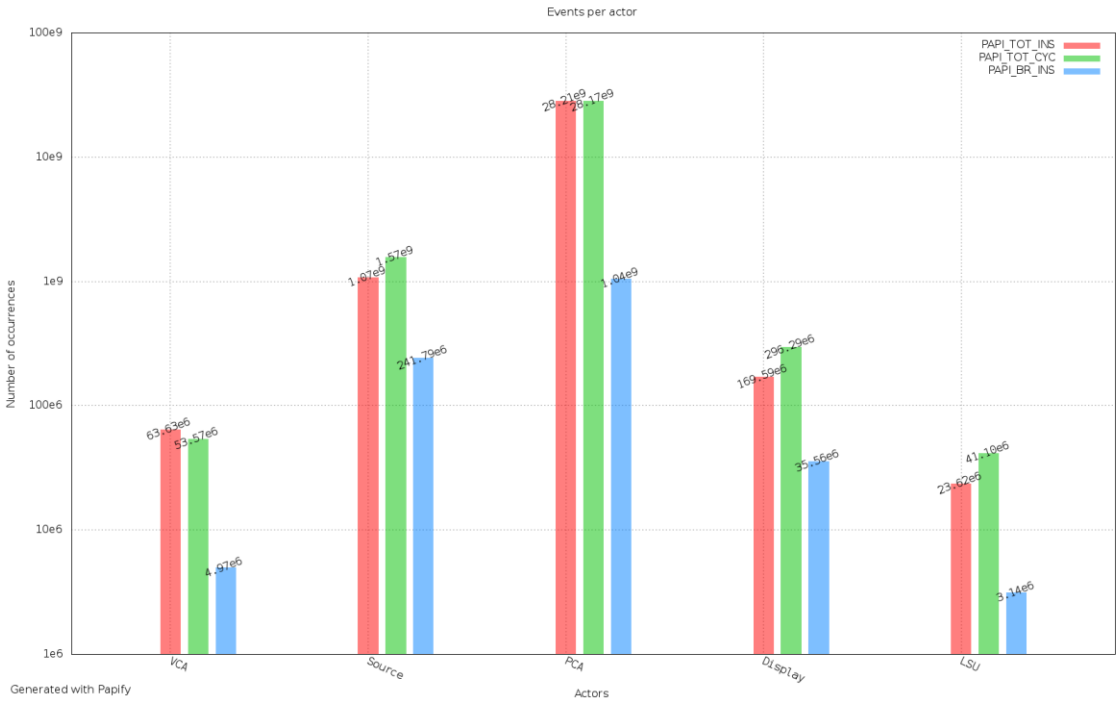


Fig. 4.6. Comparativa de recursos consumidos por cada actor en el caso más rápido. Parte 1

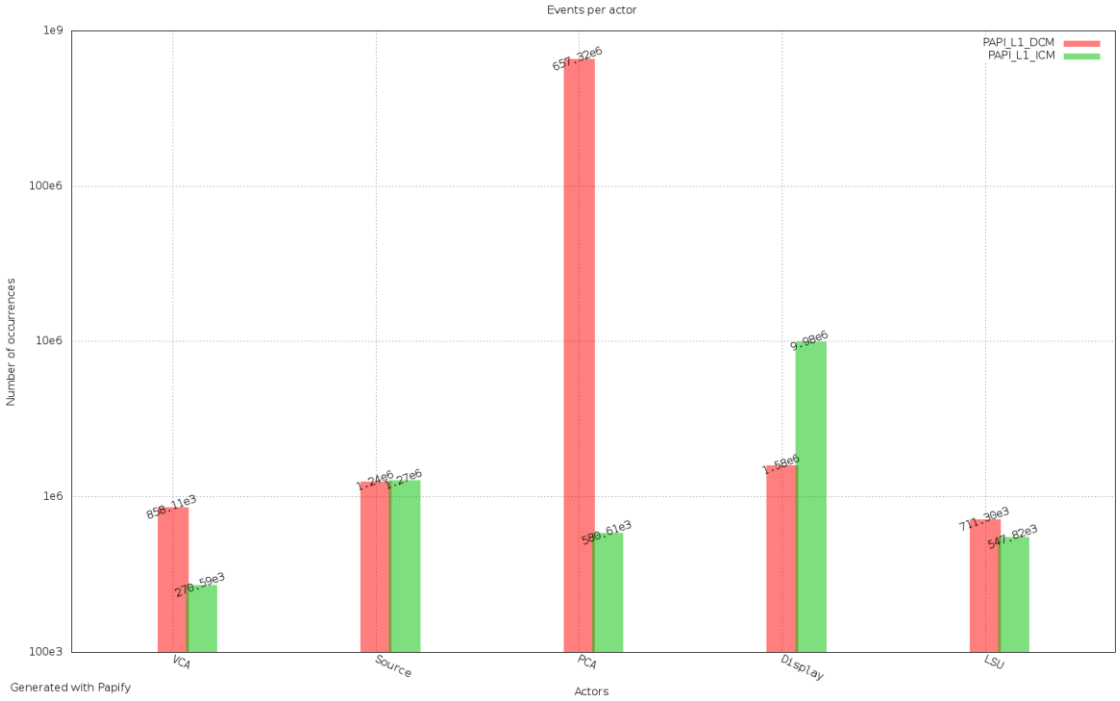


Fig. 4.7. Comparativa de recursos consumidos por cada actor en el caso más rápido. Parte 2

CAPÍTULO 5. CONCLUSIONES Y LÍNEAS FUTURAS

5.1. Conclusiones

Tras realizar el análisis de los resultados arrojados al utilizar la librería desarrollada para implementar una cadena completa de procesado de imágenes hiperespectrales, se han observado ciertos aspectos que se han considerado especialmente relevantes:

- Tras realizar una comparativa, etapa por etapa, de los resultados arrojados por *HyperMix* y de los obtenidos en las etapas implementadas en RVC – CAL, se ha observado que el comportamiento en cuanto a resultados finales es idéntico.
- En cuanto al tiempo de ejecución de los distintos algoritmos, la utilización de la librería desarrollada durante el presente Proyecto Fin de Grado conlleva una reducción sustancial de dicho tiempo.
- Los *endmembers* obtenidos por el algoritmo VCA coinciden en la mayoría de las ejecuciones para una misma *fractal,* pero el orden de obtención de los mismos puede variar de una ejecución a otra.
- Los algoritmos implementados hasta el momento están pensados para realizar un análisis secuencial de las imágenes. Por ello, el mejor tiempo de procesado se obtiene cuando se realiza el análisis de las imágenes de manera completamente secuencial, es decir, toda la cadena implementada en un único actor.
- El consumo de recursos de una cadena de desmezclado espectral implementada en 5 actores – *Source*, *PCA*, *VCA*, *LSU* y *Display* – es prácticamente equivalente cuando se ejecuta en un único procesador y cuando se ejecuta en varios.

Todo esto nos lleva a extraer una serie de conclusiones finales:

- Se ha alcanzado una mejora del tiempo de ejecución de la cadena de desmezclado espectral, llegando, en el mejor de los casos – la cadena implementada en un único actor o en 5 actores pero ejecutada en un procesador –, a ser ligeramente superior a los 10 segundos. Este tiempo, observado desde el punto de vista del proyecto europeo HELICoiD, se traduce en que, en este momento, si el sistema de adquisición de imágenes tarda más de 10 segundos en capturar una imagen, estaríamos alcanzando el tiempo real.
- Con el fin de aprovechar al máximo el potencial que nos ofrece implementar una librería en RVC – CAL, es necesario buscar una manera de optimizar los algoritmos propios de cada etapa para que sean fácilmente paralelizables ya que, como hemos observado durante el análisis de los resultados, al paralelizar el sistema por etapas completas el tiempo de procesado no mejora.

Por último, tras desarrollar una cadena de procesado de imágenes hiperespectrales completa, queda demostrada la utilidad de la librería desarrollada, la reducción de la dificultad y del tiempo de desarrollo de los distintos algoritmos propios de la cadena de desmezclado. A su vez, se ha hecho patente la mejora que supone la utilización de dicha librería en cuanto a tiempo de procesado se refiere, ya que, sin llegar a optimizar el código para realizar una paralelización óptima, se han obtenido mejores tiempos que los resultantes al utilizar el programa de comparación *HyperMix*.

5.2. Líneas futuras

Tras finalizar el presente Proyecto Fin de Grado y comentar con Raquel Lazcano – con quien se ha colaborado a lo largo de todo el proyecto y cuya memoria se recoge en [1] – los resultados obtenidos y las conclusiones alcanzadas, han surgido una serie de líneas de trabajo futuras relacionadas con ambos proyectos. Las más importantes se citan a continuación:

- Desarrollar nuevos algoritmos de procesado espectral con el fin de completar la librería desarrollada.
- Investigar la posibilidad de desarrollar nuevos algoritmos u optimizar los ya existentes de tal manera que permitan una alta capacidad de paralelización ya que, como demuestra Raquel Lazcano en [1], la configuración etapa-actor no arroja resultados satisfactorios en lo que a tiempo de procesado se refiere. El objetivo de esta línea de investigación sería aprovechar al máximo RVC – CAL para implementar dichos algoritmos paralelizables y, en consecuencia, reducir drásticamente el tiempo de procesado.
- Eliminar las librerías especializadas utilizadas durante este Proyecto para realizar operaciones matriciales de gran complejidad. De esta manera, conseguiríamos implementar la totalidad de la librería en lenguaje C y, en consecuencia, aumentaría considerablemente la interoperabilidad de dicha librería entre las distintas plataformas existentes en el mercado actual.
- Implementar una cadena de procesado completa en una plataforma multinúcleo para, de esta manera, aportar una idea más clara del comportamiento del sistema de desmezclado al proyecto HELICoiD – proyecto en el cual, como se ha mencionado anteriormente, se utilizará una plataforma multinúcleo.
- En este momento, los algoritmos utilizados funcionan para los casos en los que se trabaja con una imagen hiperespectral completa. Para aumentar la eficiencia de la cadena de procesado e incrementar el potencial de la librería, se pueden desarrollar funciones propias de algoritmos que trabajen con imágenes incompletas.
- Por último, una vez desarrollados los algoritmos con una paralelización eficiente y tras llevar a cabo la implementación de una cadena completa en una plataforma multinúcleo, se puede estudiar el consumo de energía de la cadena en dichas plataformas con el fin de desarrollar analizadores hiperespectrales portátiles.

REFERENCIAS

[1] R. Lazcano, "Generación de una librería RVC-CAL para la etapa de estimación de abundancias en el proceso de análisis de imágenes hiperespectrales ", Proyecto Fin de Grado, Escuela Técnica Superior de Ingeniería y Sistemas de Telecomunicación, Universidad Politécnica de Madrid, 2014.

[2] G. Lu, B. Fei, "Medical hyperspectral imaging: a review", Journal of biomedical optics, 19(1), 2014.

[3] B. Fei, H. Akbari, L.V. Halig, "Hyperspectral imaging and spectral-spatial classification for cancer detection", *Biomedical Engineering and Informatics (BMEI)*, 2012 5th International Conference on. IEEE, 2012.

[4] S. V. Panasyuk et al., "Medical hyperspectral imaging to facilitate residual tumor identification during surgery," *Cancer Biol. Ther,* 6(3), pp. 439–446, 2007.

[5] S. Kiyotoki et al., "New method for detection of gastric cancer by hyperspectral imaging: a pilot study", *Journal of biomedical optics*, 18(2), 2013.

[6] Z. Liu, H. Wang, Q. Li, "Tongue tumor detection in medical hyperspectral images", *Sensors*, 12(1), pp. 162-174, 2011.

[7] S. Sánchez, "Diseño e implementación de una cadena completa para desmezclado de imágenes hiperespectrales en tarjetas gráficas programables (GPUs)", tesis doctoral, Dept. de tecnología de los computadores y de las comunicaciones, Escuela Politécnica de Cáceres, Universidad de Extremadura, 2013.

[8] M. Bertogna, "Real-Time Scheduling Analysis for Multiprocessor Platforms", Ph.D. Thesis, Scuola Superiore Sant'Anna, 2007.

[9] B. Andersson, S. Baruah, J. Jonsson, "*Static-priority scheduling on multiprocessors*", *Proceedings of the IEEE Real-Time Systems Symposium*, IEEE Computer Society Press, December 2001.

[10] B. Andersson, "Static-priority scheduling on multiprocessors", Ph.D. Thesis, Department of Computer Engineering, Chalmers University, 2003.

[11] E. Juárez, R. Salvador, "Requirements and processing sub-system description report", Universidad Politécnica de Madrid, Madrid, España, Tech. Rep, July, 2014.

[12] M.-L. Wong, T.-T. Wong, K.-L. Fok, "Parallel evolutionary algorithm on graphics processing unit", *Proceedings of IEEE Congress on Evolutionary Computation 2005 (CEC 2005)*, vol.3, pp. 2286-2293, IEEE, Los Alamitos, 9 April, 2005.

[13] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "*GPU computing*" Proceedings of the IEEE, vol. 96, pp. 879-899, 2008.

[14] N. Masuda, T. Ito, T. Tanaka, A. Shiraki, T. Sugie, "Computer generated holography using a graphics processing unit", *Opt. Express*, vol. 14, pp. 603-608, 2006.

[15] K.-S. Oh, K. Jung, "GPU implementation of neural networks", *Pattern Recognition*, vol. 37, no. 6, pp. 1311-1314, 2004.

[16] S. Bernabé, S. López, A. Plaza, R. Sarmiento, "GPU implementation of an automatic target detection and classification algorithm for hyperspectral image analysis", *IEEE Geoscience and Remote Sensing Lett.*, vol. 10, no. 2, March 2013.

[17] S. Bhattacharyya, E. Johan, J. W. Janneck, C. Lucarz, M. Mattavelli, M. Raulet, "Overview of the MPEG Reconfigurable Video Coding Framework", *Journal Of Signal Processing Systems 63*, pp 251-263, © 2009 Springer Science + Business Media, LLC. Manufactured in The United States. DOI: 10.1007/s11265-009-0399-3.

[18] J. Eker and J. W. Janneck, "Cal language report", *Tech. Rep. UCB/ERL M03/48*, University of California at Berkeley, December 2003.

[19] C. Lucarz, I. Amer, M. Mattavelli, "Reconfigurable video coding: Objectives and technologies", *16th IEEE International Conference on Image Processing (ICIP)*, pp. 749-752, Nov. 2009.

[20] M. Wipliez, G. Roquier, J.-F. Nezan, "Software code generation for the RVC-CAL language", *Journal of Signal Processing Systems 63,* vol 2, pp 203-213, © 2009 Springer Science + Business Media, LLC. Manufactured in The United States. DOI: 10.1007/s11265-009-0390-z.

[21] M. Wipliez, G.Roquier, M. Raulet, J.-F. Nezan, O. Deforges, "Code generation for the MPEG Reconfigurable Video Coding framework: From CAL actions to C functions", *IEEE International Conference on Multimedia and Expo 2008*, IEEE, Hannover, Germany, DOI: 10.1109/ICME.2008.4607618.

[22] E. Bezati, M. Mattavelli, M. Raulet, "RVC-CAL dataflow implementations of MPEG AVC/H. 264 CABAC decoding", *Design and Architectures for Signal and Image Processing (DASIP)*, 2010 Conference on. IEEE, 2010.

[23] S. Browne, J. Dongarra, G. Ho, P. Mucci, "A Portable Programming Interface for Performance Evaluation on Modern Processors", *International Journal of High Performance Computing Applications*, 14:3 (Fall 2000), pp. 189 - 204.

[24] P. J. Mucci, S. Browne, C. Deane, G. Ho, "PAPI: A Portable Interface to Hardware Performance Counters", *Proceedings of Department of Defense HPCMP Users Group Conference*, University of Tennessee, June 1999.

[25] H. McCraw, J. Ralph, A. Danalis, J. Dongarra, "Power Monitoring with PAPI for Extreme Scale Architectures and Dataflow-based Programming Models", *Workshop on Monitoring and Analysis for High Performance Computing Systems Plus Applications, IEEE Cluster 2014*, IEEE, Madrid, Spain, ICL-UT-14-04, September 2014.

[26] K. London, S. Moore, P. Mucci, K. Seymour, R. Luczak, "The PAPI Cross-Platform Interface to Hardware Performance Counters", *Department of Defense Users' Group Conference Proceedings*, Biloxi, Mississippi, June 18-21, 2001.

[27] J. Dongarra, K. London, S. Moore, P. Mucci, D. Terpstra, "Using PAPI for Hardware Performance Monitoring on Linux Systems", *Conference on Linux Clusters: The HPC Revolution*, Linux Clusters Institute, Urbana, Illinois, June 25-27, 2001.

[28] M. Johnson, H. McCraw, S. Moore, P. Mucci, J. Nelson, D. Terstra, V. Weaver, "PAPI-V: Performance Monitoring for Virtual Machines", *CloudTech-HPC 2012*, Pittsburgh, PA, September 10-13, 2012.

[29] V. Weaver, M. Johnson, K. Kasichayanula, J. Ralph, P. Luszczek, D. Terpstra, S. Moore, "Measuring Energy and Power with PAPI", *International Workshop on Power-Aware Systems and Architectures*, Pittsburgh, PA, September 10, 2012.

[30] A. Ifarraguerri, C.-I. Chang, "Unsupervised hyperspectral image analysis with projection pursuit", *IEEE Trans. Geosci. Remote Sens*. vol. 38, no. 6, pp. 127–143, 2000.

[31] M. Rojas, "Caracterización de imágenes hiperespectrales utilizando *Support Vector Machine* y técnicas de extracción de características", Proyecto Fin de Carrera, Escuela Politécnica de Cáceres, Universidad de Extremadura, 2009.

[32] L.I. Jiménez, "Desarrollo de nuevos algoritmos para procesamiento de imágenes hiperespectrales en ORFEO Toolbox", Proyecto Fin de Carrera, Escuela Politécnica de Cáceres, Universidad de Extremadura, 2011.

ANEXOS

ANEXO 1. MANUAL DE CONFIGURACIÓN E INSTALACIÓN

En el presente anexo se proporciona un tutorial de instalación y configuración del entorno necesario para la correcta utilización de la librería desarrollada en este Proyecto Fin de Grado. Se ha de resaltar que este tutorial se ha desarrollado y, por tanto, está probado para un ordenador con las siguientes características:

- Sistema operativo: Ubuntu 12.04 LTS
- Compilador: gcc versión 4.6.3
- Eclipse 4.3 (Kepler) y/o Eclipse 4.4 (Luna)
- ORCC versión 2.1.1
- Librería OTB versión 3.12.0
- Librería ITK versión 4.5.1
- Librería VXL versión 1.14.0

El tutorial se divide en dos partes:

- Tutorial de configuración e instalación de las librerías ITK, OTB y VXL. Este tutorial es necesario para poder hacer un correcto uso de la librería desarrollada.
- Tutorial de configuración e instalación de Eclipse y ORCC. Este tutorial sólo es necesario si se quiere comprobar que el proceso anterior se ha realizado con éxito, puesto que, para ello, se necesita compilar un proyecto ORCC. Por tanto, su seguimiento es opcional.

A.1.1. Librerías ITK, OTB y VXL

Los pasos a seguir son los siguientes:

1. Descargar el fichero comprimido *instalador.tar.gz*
2. Extraer el contenido en el directorio *Home*, tal y como se observa en la figura A.1.1.

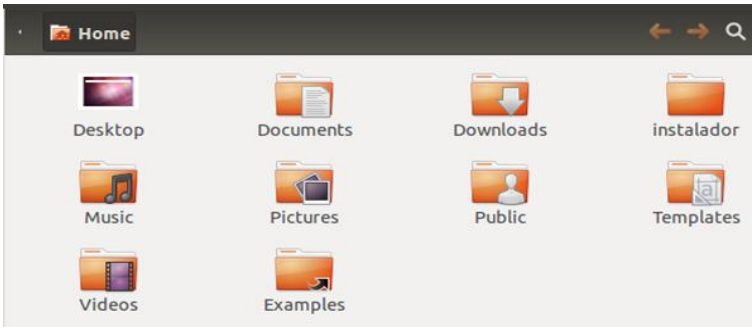


Fig. A.1.1. Contenido del directorio Home

El contenido de esta carpeta se puede observar en la figura A.1.2.

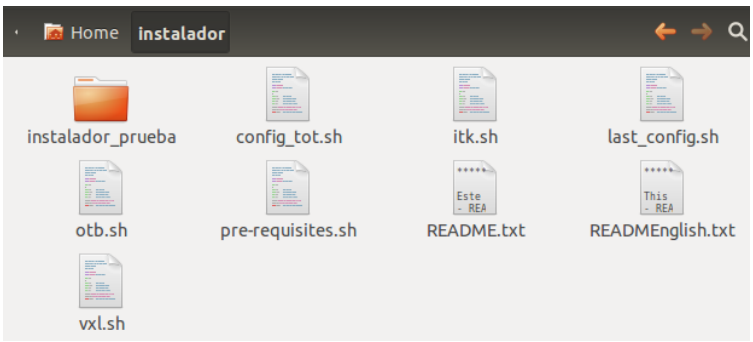


Fig. A.1.2. Contenido del directorio instalador

Como podemos observar en dicha figura, la carpeta *instalador* debe contener ocho ficheros y un directorio:

- Ficheros:
  - ✓ *pre-requisites.sh*
  - ✓ *itk.sh*
  - ✓ *otb.sh*
  - ✓ *vxl.sh*
  - ✓ *last_config.sh*
  - ✓ *config_tot.sh*
  - ✓ *README.txt*
  - ✓ *READMEnglish.txt*
- Directorio: *instalador_prueba*

De los ficheros, todos los que tienen la extensión *.sh* son ficheros de configuración, mientras que los *.txt* son ficheros de texto en los que se explica cómo usar dichos *.sh* -en español y en inglés-.
Respecto al directorio, éste aporta un proyecto ORCC - Eclipse para la comprobación de la correcta ejecución del proceso de instalación.

3. Abrir un terminal y navegar hasta la nueva carpeta – figura A.1.3 -.

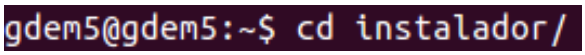


Fig. A.1.3. Navegación a la nueva carpeta

4. Ejecutar el primer fichero de instalación, escribiendo en el terminal el comando mostrado en la figura A.1.4.

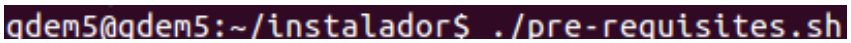


Fig. A.1.4. Ejecución del primer instalador

La duración de este instalador es variable pero, si tiene su ordenador actualizado, no tardará más de unos quince minutos.
Cuando su terminal tenga una salida como la de la figura A.1.5, se ha de introducir la contraseña de la sesión abierta en el PC.

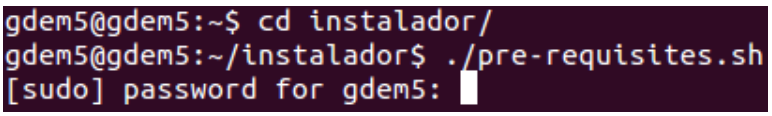


Fig. A.1.5. Contraseña del comando sudo

Asimismo, a lo largo de la instalación se solicitará la confirmación del usuario para ciertas operaciones, como puede observarse en la figura A.1.6.

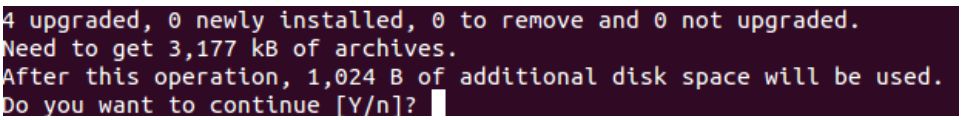


Fig. A.1.6. Confirmación de operación

Cuando esto ocurra, se ha de escribir *Y* en el terminal - o *S*, si el ordenador está configurado en español- y pulsar *Enter*.
Estas serán las únicas dos interacciones del usuario con el configurador, además de ejecutar cada fichero.

5. Esperar a que finalice el configurador anterior. Cuando lo haga, el terminal debe tener una salida parecida a la que aparece en la figura A.1.7.

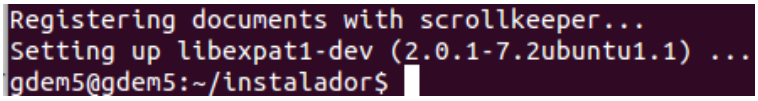


Fig. A.1.7. Finalización de pre-rquisites.sh

6. Ejecutar el siguiente fichero de instalación, escribiendo en el terminal el comando mostrado en la figura A.1.8.

```
gdem5@gdem5:~/instalador$ ./itk.sh
```

Fig. A.1.8. Ejecución del segundo instalador

Este configurador es el más complejo de todo el instalador, por lo que su duración puede llegar a extenderse hasta las tres horas.
Al igual que en el instalador anterior, la interacción del usuario con el proceso de instalación se reduce a introducir la contraseña del comando *sudo* y confirmar (*Y* / *S*) la instalación de paquetes extra.
Este instalador creará una nueva carpeta en el directorio *Home*, llamada *itk* (ver figura A.1.9), en la que se realizará la descarga y configuración de dicha librería.

Fig. A.1.9. Creación de la carpeta itk

Como hemos mencionado anteriormente, la compilación de la librería ITK puede llegar a extenderse hasta las tres horas, por lo que, en cuanto comience el proceso de compilación, el usuario puede dejar de prestar atención a este instalador. En concreto, la compilación alcanza el 60% rápidamente -en escasos minutos- pero, a partir de ahí, el proceso se ralentiza. Por tanto, el usuario puede dejar de prestar toda su atención al instalador cuando su proceso alcance un nivel similar al mostrado en la figura A.1.10.

```
inking CXX static library ../../../../lib/libITKSpatialObjects-4.5.a
 61%] Built target ITKSpatialObjects
 61%] Generating test/ITKQuadEdgeMeshHeaderTest1.cxx
canning dependencies of target ITKQuadEdgeMeshHeaderTest1
 61%] Building CXX object Modules/Core/QuadEdgeMesh/CMakeFiles/ITKQuadEdgeMesh
inking CXX executable ../../../bin/ITKQuadEdgeMeshHeaderTest1
 61%] Built target ITKQuadEdgeMeshHeaderTest1
canning dependencies of target ITKQuadEdgeMeshTestDriver
 61%] Building CXX object Modules/Core/QuadEdgeMesh/test/CMakeFiles/ITKQuadEdg
```

Fig. A.1.10. Proceso de compilación de la librería ITK

7. Esperar a que la instalación de la librería ITK finalice. Cuando lo haga, el terminal debería mostrar una salida similar a la mostrada en la figura A.1.11.

```
-- Installing: /usr/local/include/ITK-4.5/itkFrameAverageVideoFilter.hxx
-- Installing: /usr/local/include/ITK-4.5/itkFrameDifferenceVideoFilter.h
-- Installing: /usr/local/include/ITK-4.5/itkImageFilterToVideoFilterWrapper.hxx
-- Installing: /usr/local/include/ITK-4.5/itkFrameDifferenceVideoFilter.hxx
-- Installing: /usr/local/include/ITK-4.5/itkDecimateFramesVideoFilter.h
-- Installing: /usr/local/include/ITK-4.5/itkImageFilterToVideoFilterWrapper.h
-- Installing: /usr/local/include/ITK-4.5/itkFrameAverageVideoFilter.h
-- Installing: /usr/local/include/ITK-4.5/itkDecimateFramesVideoFilter.hxx
-- Installing: /usr/local/lib/cmake/ITK-4.5/Modules/ITKVideoFiltering.cmake
gdem5@gdem5:~/instalador$
```

Fig. A.1.11. Finalización de itk.sh

8. Ejecutar el comando que aparece en la figura A.1.12 para instalar la librería OTB. La duración aproximada de este instalador es de unos quince minutos.

```
gdem5@gdem5:~/instalador$ ./otb.sh
```

Fig. A.1.12. Ejecución del tercer instalador

Al igual que en los instaladores anteriores, la interacción del usuario con el proceso de instalación se limita a introducir la contraseña del comando *sudo* y confirmar la instalación de paquetes extra.
Este instalador configura la librería directamente en las carpetas del sistema, por lo que no se crea ninguna carpeta adicional en el directorio *Home*.

9. Esperar a que termine el instalador de la librería OTB. Cuando el instalador finalice, en la ventana del terminal debería aparecer una salida similar a la de la figura A.1.13.

```
-- Installing: /usr/local/include/otb/Visualization/otbMsgReporterGUI.h
-- Installing: /usr/local/bin/otbViewer
-- Removed runtime path from "/usr/local/bin/otbViewer"
gdem5@gdem5:~/instalador$
```

Fig. A.1.13. Finalización de otb.sh

10. Ejecutar en el terminal el comando que se muestra en la figura A.1.14.

```
gdem5@gdem5:~/instalador$ ./vxl.sh
```

Fig. A.1.14. Ejecución del cuarto instalador

Este instalador tendrá una duración aproximada de 40 minutos. Por ello, una vez comenzada la compilación (ver figura A.1.15), el usuario no necesitará dedicar su atención a la misma, ya que su intervención no será necesaria.

```
Linking CXX shared library ../../../lib/libvdgl.so
[ 44%] Built target vdgl
Scanning dependencies of target vtol
[ 44%] Building CXX object contrib/gel/vtol/CMakeFiles/vtol.dir/vtol_topology_obj
[ 44%] Building CXX object contrib/gel/vtol/CMakeFiles/vtol.dir/vtol_topology_cac
[ 44%] Building CXX object contrib/gel/vtol/CMakeFiles/vtol.dir/vtol_chain.o
[ 44%] Building CXX object contrib/gel/vtol/CMakeFiles/vtol.dir/vtol_two_chain.o
[ 44%] Building CXX object contrib/gel/vtol/CMakeFiles/vtol.dir/vtol_one_chain.o
[ 44%] Building CXX object contrib/gel/vtol/CMakeFiles/vtol.dir/vtol_zero_chain.o
[ 44%] Building CXX object contrib/gel/vtol/CMakeFiles/vtol.dir/vtol_edge.o
```

Fig. A.1.15. Proceso de compilación de la librería VXL

Cabe destacar que, al igual que con la librería ITK, el instalador creará una nueva carpeta en el directorio *Home*, tal y como se puede observar en la figura A.1.16. En esta carpeta será donde se configurará e instalará la librería VXL.

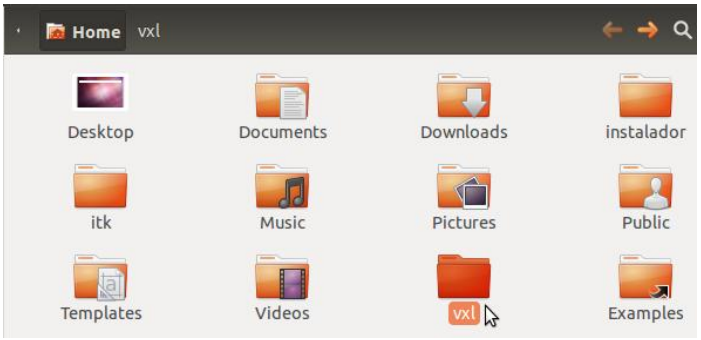


Fig. A.1.16. Creación de la carpeta vxl

11. Esperar a que finalice la instalación de la librería VXL. Cuando finalice, el terminal debería tener una salida similar a la observada en la figura A.1.17:

```
-- Installing: /usr/local/lib/libvpyr.so
-- Removed runtime path from "/usr/local/lib/libvpyr.so"
-- Installing: /usr/local/share/vxl/cmake/VXLConfig.cmake
-- Installing: /usr/local/share/vxl/cmake/VXLBuildSettings.cmake
-- Installing: /usr/local/share/vxl/cmake/VXLLibraryDepends.cmake
-- Installing: /usr/local/share/vxl/cmake/VXLStandardOptions.cmake
-- Installing: /usr/local/share/vxl/cmake/UseVXL.cmake
-- Installing: /usr/local/share/vxl/cmake/UseVGUI.cmake
gdem5@gdem5:~/instalador$
```

Fig. A.1.17. Finalización de vxl.sh

12. Ejecutar en el terminal el comando mostrado en la figura A.1.18.

```
gdem5@gdem5:~/instalador$ ./last_config.sh
```

Fig. A.1.18. Ejecución del último instalador

Este instalador realizará las últimas configuraciones, y su ejecución es prácticamente instantánea. No obstante, cabe destacar que una de las acciones realizadas por este instalador es la modificación del fichero *.profile* y, para que estos cambios sean efectivos, es necesario reiniciar el sistema, por lo que el instalador finaliza con una instrucción para reiniciar el sistema, tal y como puede observarse en la figura A.1.19. Por tanto, no olvide cerrar y guardar todos sus documentos antes de ejecutar este instalador, pues su ordenador se reiniciará automáticamente.

```
# Rebooting system -> Apply changes in .profile
sudo reboot
```

Fig. A.1.19. Última instrucción de last_config.sh

Una vez reiniciado el sistema, la instalación ya está completa. Para comprobar que todo se ha realizado correctamente, en la carpeta del instalador se proporciona un proyecto ORCC – Eclipse[15]. Si este proyecto compila y se ejecuta, la instalación se habrá realizado correctamente. Para ello, a continuación proporcionamos un tutorial de instalación y prueba de este sistema. Como hemos mencionado al inicio de este documento, este tutorial es opcional.

Por último, antes de finalizar este tutorial, hemos de mencionar que, además de este proceso de configuración, también se proporciona un configurador alternativo, en caso de que el usuario desee reducir su interacción con el proceso al mínimo. Para ello, en lugar de ejecutar los cinco instaladores anteriores, el usuario ejecutaría únicamente uno, que recibe el nombre de *config_tot.sh* (ver figura A.1.20). Este instalador ejecuta secuencialmente los cinco configuradores utilizados anteriormente (ver figura A.1.21), de tal forma que, en este caso, la interacción del usuario con el proceso se reduce a introducir la contraseña del comando *sudo* y a aceptar la instalación de paquetes extra.

Sin embargo, hemos de destacar que no recomendamos la utilización de este instalador, puesto que su duración podría llegar a extenderse más allá de las cinco horas y, además, en caso de fallo, la localización del mismo es mucho más sencilla con el otro procedimiento.

```
gdem9@GDEM9:~/instalador$ ./config_tot.sh
```

Fig. A.1.20. Ejecución del configurador global

config_tot.sh

```
#########################################################################
#                                                                       #
#                     Fichero de configuración total                    #
#                                                                       #
#  Se recomienda no utilizar este fichero:                              #
#  Su ejecución puede llegar a extenderse durante más de cinco horas    #
#                                                                       #
#########################################################################

sh ./pre-requisites.sh
sh ./itk.sh
sh ./otb.sh
sh ./vxl.sh
sh ./last_config.sh
```

Fig. A.1.21. Contenido de config_tot.sh

[15] El proyecto proporcionado contiene ya el código autogenerado por Orcc para Eclipse Luna. Si usted está utilizando esta versión, puede saltarse directamente al último paso de este tutorial. Si no es su caso, deberá ejecutar el tutorial completo.

A.1.2. Eclipse y ORCC

Instalación de Eclipse

1. Descargar el entorno Eclipse de la página oficial: http://www.eclipse.org/downloads/
   En concreto, se descargará la versión *Eclipse IDE for C/C++ Developers*.
2. Extraer el fichero comprimido descargado en el directorio que estime oportuno -por ejemplo, en *Home* o en el Escritorio.
   Nota: Durante el desarrollo de este proyecto, Eclipse ha lanzado una nueva versión (Eclipse 4.4 Luna). Por ello, este tutorial da soporte tanto a la versión antigua (Eclipse 4.3 Kepler) como a la nueva (Eclipse 4.4 Luna). El único cambio detectado entre ambas versiones se localiza en el *CMakeLists.txt* que genera ORCC al compilar un proyecto, por lo que aportaremos ambos para que el usuario utilice el que resulte necesario.
3. Descargar Java de la página oficial:
   https://www.java.com/es/download/linux_manual.jsp?locale=es
   Extraer el fichero comprimido descargado y renombrar la carpeta resultante con el nombre *jre*, como puede apreciarse en la figura A.1.22.

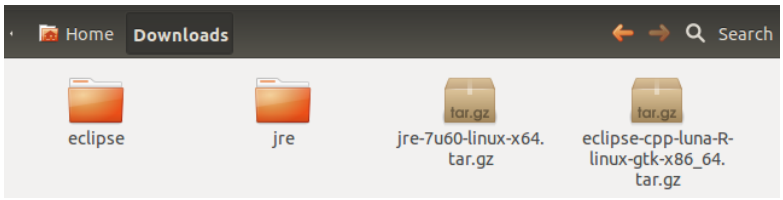


Fig. A.1.22. Descarga de Java

4. Copiar la carpeta *jre* obtenida en el paso anterior dentro de la carpeta Eclipse, como se muestra en la figura A.1.23.

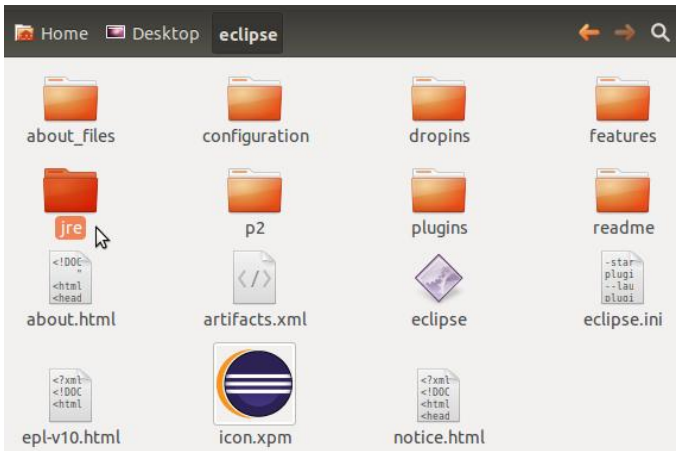


Fig. A.1.23. Inserción de jre *en la carpeta* eclipse

Instalación de ORCC

1. Abrir Eclipse y seleccionar el menú *Help → Install new software... → Add...*
   En *Location*, añadir *http://orcc.sourceforge.net/eclipse* (ver figura A.1.24).

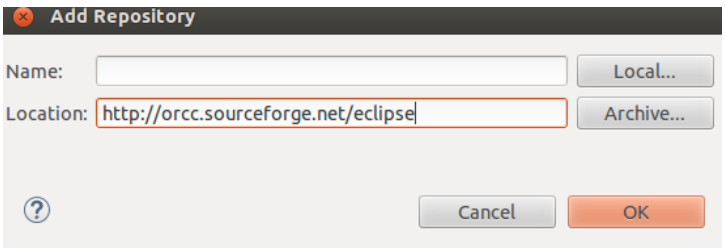


Fig. A.1.24. Ventana de descarga del software ORCC

2. Seleccionar *Open RVC-Cal Compiler* y pulsar *Next* → *Next* (ver figura A.1.25).

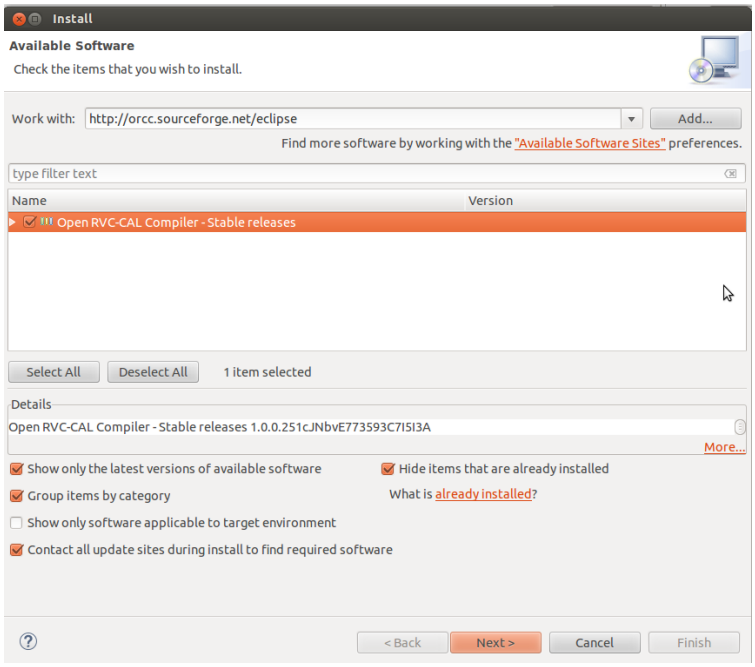


Fig. A.1.25. Selección de RVC – CAL compiler

3. Aceptar la licencia y finalizar, como se muestra en la figura A.1.26.

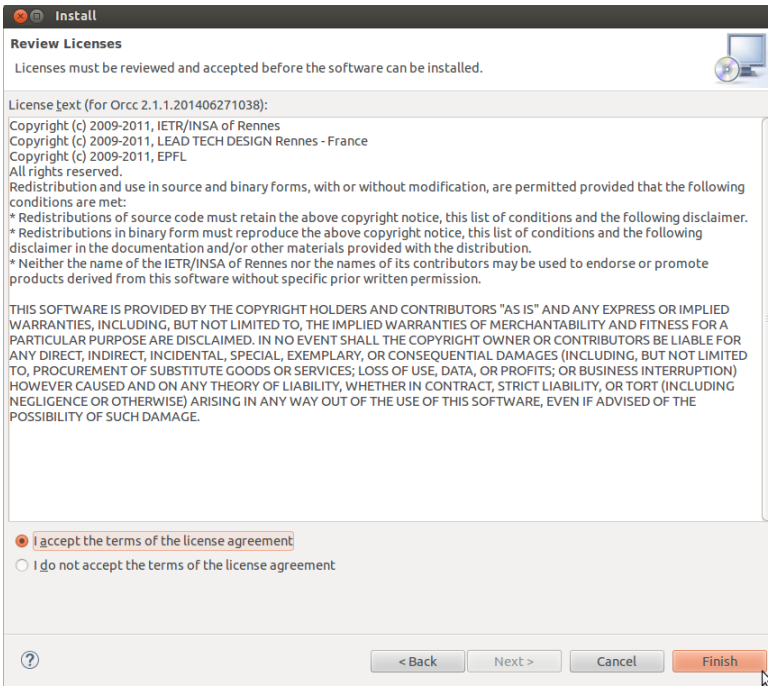


Fig. A.1.26. Licencia de ORCC 2.1.1

4. Darle a Ok cuando aparezca el mensaje que se muestra en la figura A.1.27.

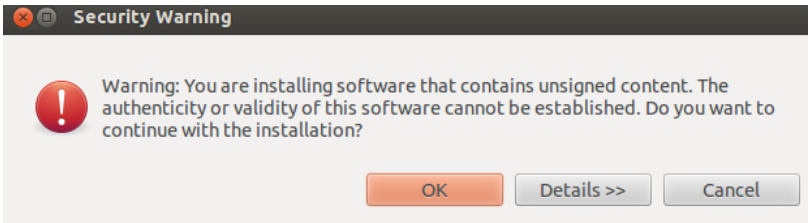


Fig. A.1.27. Mensaje de aviso

5. Reiniciar Eclipse. Si existe algún problema durante esta configuración, consultar la siguiente página: *http://orcc.sourceforge.net/getting-started/install-orcc/*
6. Cuando solicite la ruta al *workspace*, seleccionar la carpeta *project* dentro del directorio *instalador_prueba*. La ruta completa es *instalador/instalador_prueba/project*.
7. Pulsar en *File* → *Import* → *General* → *Existing Projects into Workspace* → *Next*
8. En *root directory*, seleccionar la ruta a la carpeta *hsi_system* que se proporcionaba dentro del instalador. En concreto, la ruta a la carpeta es *instalador/instalador_prueba/project/hsi_system.*
9. En *Projects*, seleccionar el proyecto a importar y seleccionar *Finish*, tal y como se muestra en la figura A.1.28.

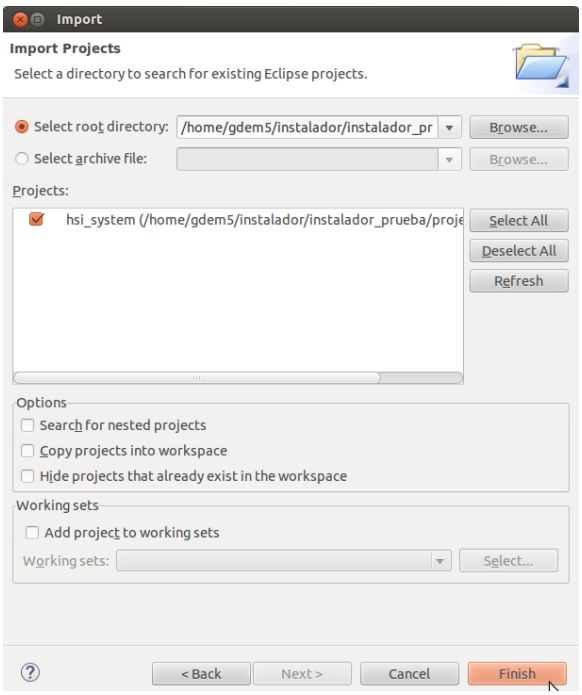


Fig. A.1.28. Ventana de importación de proyectos

10. En el proyecto, navegar hasta *hsi_system.xdf*, seleccionarla y pulsar botón derecho *Run as...* → *Run Configurations* → *ORCC compilation* y configurar la compilación con los parámetros mostrados en la figura A.1.29.

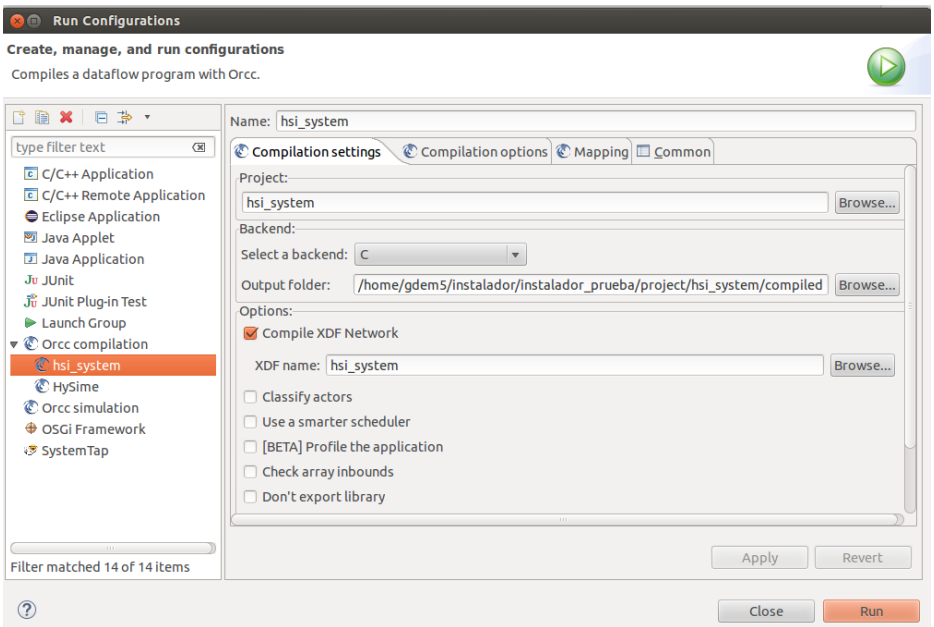


Fig. A.1.29. Configuración de los parámetros de compilación

11. Pulsar *Run*. En la consola debe aparecer un mensaje similar al de la figura A.1.30.

Fig. A.1.30. Resultado de la compilación en ORCC

12. Abrir un terminal y navegar hasta la carpeta *instalador_prueba*, que está dentro del directorio instalador. El contenido de esta carpeta se puede observar en la figura A.1.31.

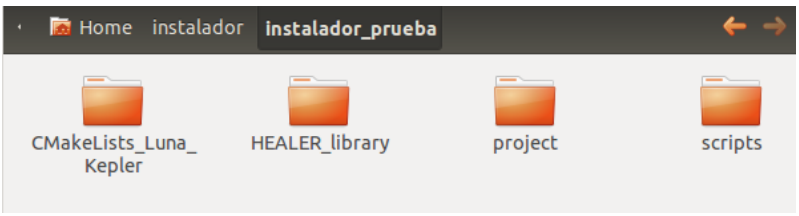


Fig. A.1.31. Contenido de la carpeta instalador_prueba

Como podemos observar, la carpeta *instalador_prueba* contiene cuatro directorios:

- *CMakeLists_Luna_Kepler*: En este directorio aportamos los archivos necesarios para garantizar la compatibilidad del sistema con ambas versiones de Eclipse -Luna y Kepler-, para que, a la hora de la compilación, se emplee el fichero correspondiente a la versión que se esté utilizando.
- *HEALER library*: En este directorio se encuentran todos los ficheros fuente necesarios para la implementación de la librería desarrollada en este Proyecto Fin de Grado, y que serán requeridos durante el proceso de compilación.
- *project*: En este directorio se proporciona un proyecto Eclipse - ORCC cuyo objetivo es la comprobación de la correcta realización de este tutorial. Además, se proporciona un *workspace* ya construido (versión Eclipse Luna) para poder comprobar de forma sencilla el correcto funcionamiento del proyecto.
- *scripts*: En este directorio se proporcionan los ficheros de configuración necesarios para automatizar la compilación del programa. En concreto, se proporcionan dos ficheros: *compiler_kepler.sh* y *compiler_luna.sh*. Cuando se vaya a compilar el proyecto, el usuario únicamente tendrá que navegar hasta esta carpeta y ejecutar el *compiler* adecuado a su versión de Eclipse.

  Este tutorial se ha desarrollado con *Eclipse Luna*, por lo que mostramos la ejecución del proceso de compilación para esta versión, como puede observarse en la figura A.1.32.

13. En el terminal, ejecutar uno de los siguientes comandos:

    ./compiler-kepler.sh si su versión es *Eclipse Kepler*

    ./compiler-luna.sh si su versión es *Eclipse Luna*

```
gdem5@gdem5:~$ cd instalador/instalador_prueba/
gdem5@gdem5:~/instalador/instalador_prueba$ cd scripts/
gdem5@gdem5:~/instalador/instalador_prueba/scripts$ ./compiler_luna.sh
```

Fig. A.1.32. Ejecución de compiler-luna.sh

Este fichero realiza la compilación del proyecto y, si todo se realiza de forma correcta, ejecuta la aplicación generada al finalizar la compilación. Si todo el proceso se ha realizado de forma satisfactoria, la salida de su terminal debería ser parecida a la mostrada en la figura A.1.33.

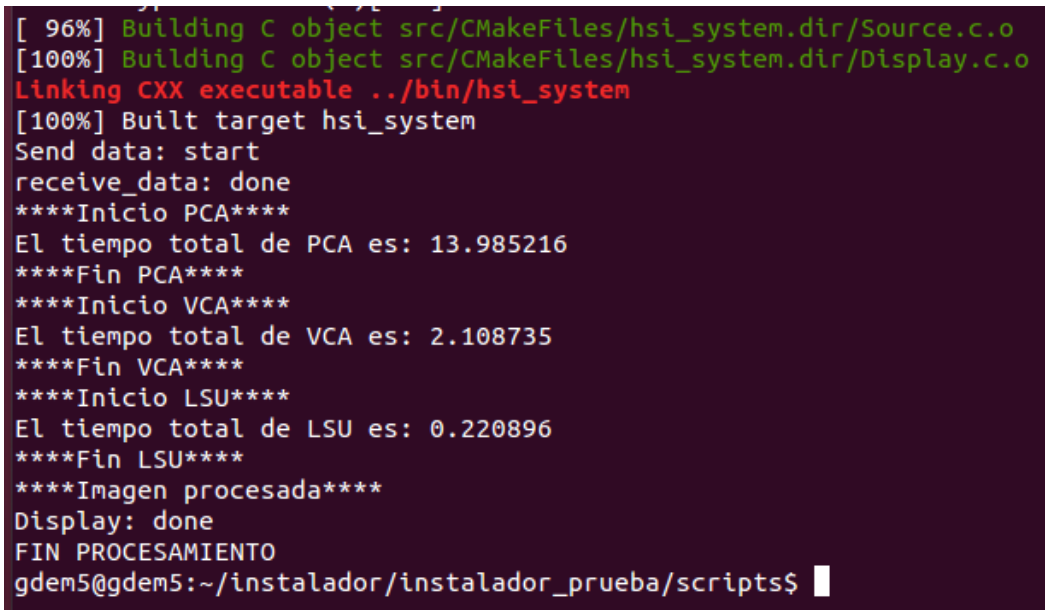


Fig. A.1.33. Ejecución de la aplicación

Éste es el final del manual de instalación y configuración del entorno para la utilización de la librería desarrollada. Si su salida final ha sido similar a la mostrada en la figura A.1.33, el proceso se ha realizado con éxito y ya puede utilizar la librería en su ordenador.

ANEXO 2. MANUAL DE USUARIO: API DE LA LIBRERÍA

En el presente anexo se proporciona un manual de usuario de la librería desarrollada en este Proyecto Fin de Grado. Este manual está compuesto de un índice de las funciones que la componen y una descripción detallada de cada una de ellas, aportando una breve descripción de su funcionalidad y explicando la naturaleza de los parámetros de entrada y salida. Asimismo, también se proporciona un índice de las librerías que se necesitan y que, en consecuencia, se importan en esta librería.
Este documento constituye el API de la librería, que también se proporciona en formato digital, tanto en PDF como en HTML. Cabe destacar que, debido a la naturaleza del proyecto en el que se inscribe esta librería, la documentación está en inglés.

A.2.1. Librerías necesarias

En esta sección, se proporciona una relación de las librerías requeridas por las funciones desarrolladas:

```
#include <assert.h>

#include <dlfcn.h>

#include <dirent.h>

#include <errno.h>

#include <fcntl.h>

#include <malloc.h>

#include <math.h>

#include <memory.h>

#include <pthread.h>

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <sys/time.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <time.h>

#include <unistd.h>

#include "cppWrapper.h"
```

Como se puede observar, excepto *cppWrapper.h*, todas las librerías son librerías de sistema, por lo que no se requiere ninguna descarga adicional. Respecto a *cppWrapper.h*, ésta se proporciona con el proyecto, y debe ser incluida en la carpeta *compiled/libs/orcc-native/include*. No obstante, este proceso está automatizado, por lo que no es necesaria la intervención del usuario (ver Anexo 3).

A.2.2. Índice de funciones

En esta sección se proporciona una relación de los prototipos de las funciones implementadas, así como una breve descripción de cada una de ellas:

float **double_to_float** (double *value*)
Cast between float and double types.

int **double_to_int** (double *value*)
Cast between int and double types.

double **elev** (double *base*, double *power*)
Raise a number to a given power.

void **end_program** ( )
Finish program execution.

double **float_to_double** (float *value*)
Cast between float and double types.

void **gen_conf_data** (int *rows*, int *columns*, int *bands*, int *nbits*, int *nformat*, int *numPC*, int *numEnd*)
Generate the Data.cal file.

void **gen_hdr** (double* *matrixOut*, int *rows*, int *columns*, int *lines*, int *samples*, int *bands*, int *numEnd*, char* *path*, char* *fileName*, int *typeFile*)
Generate a complete hyperspectral image, which has two files: a binary file and a .hdr file.

void **get_diagonal** (double* *matrix*, double* *vectorDiag*, int *positions*)
Get the diagonal vector of a matrix.

void **get_eigenvectMatrix** (double* *matrixIn*, double* *matrixEigenvectors*, int *rows*, int *columns*)
Get the eigenvectors matrix.

void **get_inverse** (double* *matrixIn*, double* *matrix_inverse*, int *rows*, int *columns*)
Get the inverse matrix.

void **get_pinverse** (double* *matrixIn*, double* *matrix_p_inverse*, int *rows*, int *columns*)
Get a pseudo - inverse matrix.

double **get_time** ( )
Get the real time of the system (in seconds, with a resolution of nanoseconds)

int **getIndexMax** (double* *vector*, int *positions*)
Get the position of the largest element vector (absolute values only)

double **int_to_double** (int *value*)
Cast between int and double types.

double **log** (double *value*)
Get the logarithm of a number (base 10): result = log(value)

void **matrix_extend** (double* *matrix1,* double* *matrix2,* double *rows1,* double *columns1,* double *rows2,* double *columns2*)
Extend a matrix.

void **matrix_fill_column** (double* *matrix1,* double *rows1,* double *columns1,* double *column*, double *value*)
Set a column of the matrix to the same value.

void **matrix_get_column** (double* *matrix1,* double* *vector*, double *rows1,* double *columns1,* double *column*)
Get a certain column from a matrix.

void **matrix_get_row** (double* *matrix1,* double* *vector*, double *rows1,* double *columns1,* int *row*)

Get a certain row from a matrix.

void **matrix_minus_matrix** (double* *matrix1,* double* *matrix2,* double* *matrixOut*, int *rows*, int *columns*)

Deduct two matrices.

void **matrix_minus_value** (double* *matrixIn*, double* *matrixOut*, double *value*, int *rows*, int *columns*)

Deduct the same value to all the elements of a matrix.

void **matrix_mult** (double* *matrix1,* double* *matrix2,* double* *matrixResult*, double *rows1,* double *columns1,* double *rows2,* double *columns2*)

Multiply two matrices.

void **matrix_mult_vector** (double* *matrix1,* double* *vector*, double* *vectorResult*, double *rows1,* double *columns1,* double *positions_vector*)

Multiply a matrix by a vector (the result is a vector)

void **matrix_plus_matrix** (double* *matrix1,* double* *matrix2,* double* *matrixOut*, int *rows*, int *columns*)

Add two matrices.

void **matrix_plus_value** (double* *matrixIn*, double* *matrixOut*, double *value*, int *rows*, int *columns*)

Add the same value to all the elements of a matrix.

void **matrix_reduce** (double* *matrix1,* double* *matrix2,* double *rows1,* double *columns1,* double *rows2,* double *columns2*)

Reduce matrix dimensions.

void **matrix_scale** (double* *matrixIn*, double* *matrixOut*, double *value*, int *rows*, int *columns*)

Divide each element of a matrix by the same value.

void **matrix_set_column** (double* *matrix1,* double* *vector*, double *rows1,* double *columns1,* double *column*)

Set a certain column of a matrix.

void **matrix_set_row** (double* *matrix1,* double* *vector*, double *rows1,* double *columns1,* double *row*)

Set a certain row of a matrix.

double **maxSqrtSum** (double* matrix, double rows, double columns**)**

Square each element of each row and add them, row by row.

double **mean_vector** (double* *vector*, int *positions*)

Get the mean value of a given vector.

void **modify_config** (int *num_PC*, int *num_Endmembers*, char* *pathIn*, char* *var1,* char* *var2*)

Modify analysis chain configuration parameters.

void **random_vector** (double* *vector*, double *positions*)

Fill a vector with random values between 0 and 1 (of double type)

void **read_txt(**double* matrixIn, int rows, int columns, char* *fileName)*

Read a .txt file and save its contents in a matrix.

int **rem** (double *dividend*, int *divisor*)

Return the remainder of a division.

double **sq_root** (double *value*)

Get the square root of a number.

void **transpose**(double* *matrixIn*, double* *matrixTranspose*, double *rows*, double *columns*)
Transpose a matrix.

void **vector_minus_value** (double* *vectorIn,* double *value*, double* *vectorMinus,* int *positions*)
Subtract the same value from each position of a vector.

void **vector_minus_vector** (double* *vector1,* double* *vector2,* double* *vectorResult,* int *positions*)
Subtract one vector from another.

double **vector_mult** (double* *vector1,* double* *vector2,* int *positions*)
Multiply two vectors.

void **vector_mult_matrix** (double* *vector,* double* *matrix,* double* *vectorResult*, double *positions_vector*, double *rows1,* double *columns1*)
Multiply a matrix by a vector.

void **vector_scale** (double* *vectorIn*, double* *vectorOut*, double *value*, int *positions*)
Divide each element of a vector by the same value.

void **write_double_binary**(double* *matrixOut*, int *rows*, int *columns*, char* *fileName*)
Write an image into a binary file, with the BSQ format of an hyperspectral image.

void **write_short_binary**(short int* matrixOut, int rows, int columns, char* *fileName)*
Write an image into a binary file, with the BIP format of an hyperspectral image.

void **write_txt**(double* matrixOut, int rows, int columns, char* *fileName)*
Save a matrix into a .txt file.

A.2.3. Descripción detallada

En esta sección se proporciona una descripción detallada de cada una de las funciones enumeradas en la sección anterior, describiendo tanto su funcionalidad como los parámetros de entrada y salida.

1.1. float double_to_float (double *value*)
Cast between float and double types.

Parameters:

value	double value

Returns:
The same value, but as a float

1.2. int double_to_int (double *value*)
Cast between int and double types.

Parameters:

value	double value

Returns:
The same value, but as an int

1.3. double elev (double *base*, double *power*)
Raise a number to a given power.

Parameters:

base	Number we want to raise to a certain power
power	Power

Returns:
Result

1.4. void end_program ( )

Finish program execution

1.5. double float_to_double (float *value*)

Cast between float and double types.

Parameters:

value	float value

Returns:

The same value, but as a double

1.6. void gen_conf_data (int *rows*, int *columns*, int *bands*, int *nbits*, int *nformat*, int *numPC*, int *numEnd*)

Generate the Data.cal file.

This function is only used to generate the output of algorithms related to endmember estimation.

Parameters:

rows	Number of rows of the original hyperspectral image
columns	Number of columns of the original hyperspectral image
bands	Number of bands of the original hyperspectral image
nbits	Number of bits of each pixel
nformat	Number to select the format: 0 - BSQ, 1 - BIP, 2 - BIL
numPC	Number of principal components (PCA algorithm)
numEnd	Number of endmembers

1.7. void gen_hdr (double* *matrixOut*, int *rows*, int *columns*, int *lines*, int *samples*, int *bands*, int *numEnd*, char* *path*, char* *fileName*, int *typeFile*)

Generate a complete hyperspectral image, which has two files: a binary file and a .hdr file.

The last one is the header of the binary file, and it must have the same name, but with the .hdr extension. The file is written (row by row) in the bin directory of the project. The format of the analyzed hyperspectral images is supposed to be BSQ.

Parameters:

matrixOut	Image to save in the binary file
rows	Number of rows of *matrixOut*
columns	Number of columns of *matrixOut*
lines	Number of lines of the initial image
samples	Number of samples of the initial image
bands	Number of bands of the output hyperspectral image
numEnd	Number of endmembers of the output hyperspectral image
path	Path to the hyperspectral image associated with the .hdr generated
fileName	Name of the binary file (without the extension)
typeFile	Number to select the file we want to generate: 1 – Dimensionality reduction, 2 – Endmembers, 3 - Abundances

1.8. void get_diagonal (double* *matrix*, double* *vectorDiag*, int *positions*)

Get the diagonal vector of a matrix

Parameters:

matrix	Matrix whose diagonal we want to obtain (it has to be a square matrix)
vectorDiag	Diagonal vector (result)
positions	Number of rows or columns of *matrix*

1.9. void get_eigenvectMatrix (double* *matrixIn*, double* *matrixEigenvectors*, int *rows*, int *columns*)

Get the eigenvectors matrix

Parameters:

matrixIn	Matrix whose eigenvectors we want to obtain (it has to be a square matrix)
matrixEigenvectors	Eigenvectors matrix (result)
rows	Number of rows of *matrixIn*
columns	Number of columns of *matrixIn*

1.10. void get_inverse (double* *matrixIn*, double* *matrix_inverse*, int *rows*, int *columns*)

Get the eigenvectors matrix

Parameters:

matrixIn	Matrix whose inverse we want to obtain (it has to be a square matrix)
matrix_inverse	Inverse matrix (result)
rows	Number of rows of *matrixIn*
columns	Number of columns of *matrixIn*

1.11. void get_pinverse (double* *matrixIn*, double* *matrix_p_inverse*, int *rows*, int *columns*)

Get the eigenvectors matrix

Parameters:

matrixIn	Matrix whose pseudo - inverse we want to obtain (it has to be a square matrix)
matrix_p_inverse	Pseudo - inverse matrix (result)
rows	Number of rows of *matrixIn*
columns	Number of columns of *matrixIn*

1.12. double get_time ( )

Get the real time of the system (in seconds, with a resolution of nanoseconds)

Returns:

System time

1.13. int getIndexMax (double* *vector*, int *positions*)

Get the position of the largest element vector (absolute values only)

Parameters:

vector	Vector to use in the operation
positions	Number of elements of *vector*

Returns:

Returns the position (index) of the largest element vector

1.14. double int_to_double (int *value*)

Cast between int and double types.

Parameters:

value	int value

Returns:

The same value, but as a double

1.15. double log (double *value*)

Get the logarithm of a number (base 10): result = log(value)

Parameters:

value	Number whose logarithm we want to calculate

Returns:

Result

1.16. void matrix_extend (double* *matrix1*, double* *matrix2*, double *rows1*, double *columns1*, double *rows2*, double *columns2*)

Extend a matrix.

Parameters:

matrix1	Matrix whose dimensions we want to extend
matrix2	Extended matrix (result)
rows1	Number of rows of *matrix1*
columns1	Number of columns of *matrix1*
rows2	Number of rows of *matrix2*
columns2	Number of columns of *matrix2*

1.17. void matrix_fill_column (double* *matrix1*, double *rows1*, double *columns1*, double *column*, double *value*)

Set each element of a column of the input matrix to the same value.

Parameters:

matrix1	Matrix whose column we want to fill
rows1	Number of rows of *matrix1*
columns1	Number of columns of *matrix1*
column	Position of the filled column
value	Value of each element of the filled column

1.18. void matrix_get_column (double* *matrix1*, double* *vector*, double *rows1*, double *columns1*, double *column*)

Get a certain column from a matrix.

Parameters:

matrix1	Matrix whose column we want to obtain
vector	Vector where the required column is saved
rows1	Number of rows of *matrix1*
columns1	Number of columns of *matrix1*
column	Column number

1.19. void matrix_get_row (double* *matrix1*, double* *vector*, double *rows1*, double *columns1*, int *row*)

Get a certain row from a matrix.

Parameters:

matrix1	Matrix whose row we want to obtain
vector	Vector where the required row is saved
rows1	Number of rows of *matrix1*
columns1	Number of columns of *matrix1*
row	Row number

1.20. void matrix_minus_matrix (double* *matrix1*, double* *matrix2*, double* *matrixOut*, int *rows*, int *columns*)

Deduct two matrices.

Parameters:

matrix1	Minuend
matrix2	Subtrahend
matrixOut	Matrix where the result is saved
rows	Number of rows of *matrix1*and *matrix2*
columns	Number of columns of *matrix1*and *matrix2*

1.21. void matrix_minus_value (double* *matrixIn*, double* *matrixOut*, double *value*, int *rows*, int *columns*)

Deduct the same value to all the elements of a matrix.

Parameters:

matrixIn	Matrix whose elements we want to edit
matrixOut	Matrix where the result is saved
value	Value to deduct
rows	Number of rows of *matrixIn*
columns	Number of columns of *matrixIn*

1.22. void matrix_mult (double* *matrix1*, double* *matrix2*, double* *matrixResult*, double *rows1*, double *columns1*, double *rows2*, double *columns2*)

Multiply two matrices.

Parameters:

matrix1	First factor
matrix2	Second factor
matrixResult	Product
rows1	Number of rows of first factor
columns1	Number of columns of first factor
rows2	Number of rows of second factor
columns2	Number of columns of second factor

1.23. void matrix_mult_vector (double* *matrix1*, double* *vector*, double* *vectorResult*, double *rows1*, double *columns1*, double *positions_vector*)

Multiply a matrix by a vector (the result is a vector)

Parameters:

matrix1	First factor
vector	Second factor
vectorResult	Product
rows1	Number of rows of first factor
columns1	Number of columns of first factor
positions_vector	Number of elements of second factor

1.24. void matrix_plus_matrix (double* *matrix1*, double* *matrix2*, double* *matrixOut*, int *rows*, int *columns*)

Add two matrices.

Parameters:

matrix1	First operand
matrix2	Second operand
matrixOut	Matrix where the result is saved
rows	Number of rows of *matrix1* and *matrix2*
columns	Number of columns of *matrix1* and *matrix2*

1.25. void matrix_plus_value (double* *matrixIn*, double* *matrixOut*, double *value*, int *rows*, int *columns*)

Add the same value to all the elements of a matrix.

Parameters:

matrixIn	Matrix whose elements we want to edit
matrixOut	Matrix where the result is saved
value	Value to add
rows	Number of rows of *matrixIn*
columns	Number of columns of *matrixIn*

1.26. void matrix_reduce (double* *matrix1*, double* *matrix2*, double *rows1*, double *columns1*, double *rows2*, double *columns2*)

Reduce matrix dimensions.

Parameters:

Matrix1	Matrix whose dimensions we want to reduce
matrix2	Reduced matrix (result)
rows1	Number of rows of *matrix1*
columns1	Number of columns of *matrix1*
rows2	Number of rows of *matrix2*
columns2	Number of columns of *matrix2*

1.27. void matrix_scale (double* *matrixIn*, double* *matrixOut*, double *value*, int *rows*, int *columns*)

Divide each element of a matrix by the same value.

Parameters:

matrixIn	Each element of this matrix is the dividend of each operation
matrixOut	Result of the division
value	Divider of each operation
rows	Number of rows of *matrixIn*
columns	Number of columns of *matrixIn*

1.28. void matrix_set_column (double* *matrix1*, double* *vector*, double *rows1*, double *columns1*, double *column*)

Set a certain column of a matrix.

Parameters:

matrix1	Matrix whose column we want to modify
vector	Contains the new values of the column we want to edit
rows1	Number of rows of *matrix1*
columns1	Number of columns of *matrix1*
column	Column number

1.29. void matrix_set_row (double* *matrix1*, double* *vector*, double *rows1*, double *columns1*, double *row*)

Set a certain row of a matrix.

Parameters:

matrix1	Matrix whose row we want to modify
vector	Contains the new values of the row we want to edit
rows1	Number of rows of *matrix1*
columns1	Number of columns of *matrix1*
row	Row number

1.30. double maxSqrtSum (double* *matrix*, double *rows*, double *columns*)

Square each element of each row and add them, row by row.

Parameters:

matrix	Matrix to use in the operation
rows	Number of rows of *matrix*
columns	Number of columns of *matrix*

Returns:

The function returns the square root of the largest row sum of squares

1.31. double mean_vector (double* *vector*, int *positions*)

Get the mean value of a given vector.

Parameters:

vector	Vector whose mean value we want to obtain
positions	Size of *vector*

Returns:

Mean value

1.32. void modify_config (int *num_PC*, int *num_Endmembers*, char* *pathIn*, char* *var1*, char* *var2*)

Modify analysis chain configuration parameters.

Parameters:

num_PC	Number of principal components we want to obtain
num_Endmembers	Number of *endmembers* we want to obtain
pathIn	Path to the generated code of the HSI analysis chain
var1	Name of the first variable we want to modify (number of principal components)
var2	Name of the second variable we want to modify (number of *endmembers*)

1.33. void random_vector (double* *vector*, double *positions*)

Fill a vector with random values between 0 and 1 (of double type)

Parameters:

vector	Vector filled with random values
positions	Number of elements of *vector*

1.34. void read_txt(double* *matrixIn*, int *rows*, int *columns*, char* *fileName*)

Read a .txt file (row by row) and save its contents in a matrix.

Hence, the .txt file should contain only a matrix, and it should be located in the folder where the executable file is generated

Parameters:

matrixIn	Matrix where the result is going to be saved
rows	Number of rows of *matrixIn*
columns	Number of columns of *matrixIn*
fileName	Name of the required .txt file. The extension must also be written in this parameter

1.35. int rem (double *dividend*, int *divisor*)

Return the remainder of a division.

Parameters:

dividend	Dividend of the division
divisor	Divisor of the division

Returns:

Remainder of the division

1.36. double sq_root (double *value*)

Get the square root of a number.

Parameters:

value	Radicand

Returns:

Result

1.37. void transpose(double* *matrixIn*, double* *matrixTranspose*, double *rows*, double *columns*)

Transpose a matrix.

Parameters:

matrixIn	Matrix whose transpose we want to obtain
matrixTranspose	Matrix where the result is saved
rows	Number of rows of *matrixIn*
columns	Number of columns of *matrixIn*

1.38. void vector_minus_value (double* *vectorIn*, double *value*, double* *vectorMinus*, int *positions*)

Subtract the same value from each position of a vector.

Parameters:

vectorIn	Each position is the minuend of each subtraction
value	Common subtrahend of each subtraction
vectorMinus	Vector where the result is saved
positions	Size of *vectorIn*

1.39. void vector_minus_vector (double* *vector1*, double* *vector2*, double* *vectorResult*, int *positions*)

Subtract one vector from another.

Parameters:

vector1	Minuend vector
vector2	Subtrahend vector
vectorResult	Vector where the result is saved
positions	Size of *vector1* and *vector2* (they must have the same number of elements)

1.40. double vector_mult (double* *vector1*, double* *vector2*, int *positions*)

Multiply two vectors.

Parameters:

vector1	First factor
vector2	Second factor
positions	Number of elements of each vector

Returns:

Product

1.41. void vector_mult_matrix (double* *vector*, double* *matrix*, double* *vectorResult*, double *positions_vector*, double *rows1*, double *columns1*)

Multiply a matrix by a vector.

Parameters:

vector	First factor
matrix	Second factor
vectorResult	Vector where the result is saved
positions_vector	Number of elements of *vector*
rows1	Number of rows of *matrix*
columns1	Number of columns of *matrix*

1.42. void vector_scale (double* *vectorIn*, double* *vectorOut*, double *value*, int *positions*)

Divide each element of a vector by the same value.

Parameters:

vectorIn	Each element of this vector is the dividend of each operation
vectorOut	Result of the division
value	Divider of each operation
positions	Number of elements of *vectorIn*

1.43. void write_double_binary(double* *matrixOut*, int *rows*, int *columns*, char* *fileName*)

Write an image into a binary file (row by row).

This function must be used to write images implemented with double type of data.

Parameters:

matrixOut	Image to save in the binary file
rows	Number of rows of *matrixOut*
columns	Number of columns of *matrixOut*
fileName	Name of the generated file

1.44. void write_short_binary(short int* *matrixOut*, int *rows*, int *columns*, char* *fileName*)

Write an image into a binary file (row by row).

This function must be used to write images implemented with short int type of data.

Parameters:

matrixOut	Image to save in the binary file
rows	Number of rows of *matrixOut*
columns	Number of columns of *matrixOut*
fileName	Name of the generated file

1.45. void write_txt(double* *matrixOut*, int *rows*, int *columns*, char* *fileName*)

Save a matrix into a .txt file (row by row).

This file is generated in the bin folder of the project

Parameters:

matrixOut	Matrix to be saved in the .txt file
rows	Number of rows of *matrixOut*
columns	Number of columns of *matrixOut*
fileName	Name of the output .txt file. The extension must also be written in this parameter

ANEXO 3. ORGANIZACIÓN DE DIRECTORIOS

En el presente anexo se realizará una breve explicación de la estructura general de los directorios utilizados durante el desarrollo del Proyecto Fin de Grado y, posteriormente, se explicará el contenido y la organización de cada uno de ellos.

En primer lugar, en la figura A.3.1 podemos observar la estructura de directorios utilizada:

Fig. A.3.1. Estructura de directorios utilizada

El contenido de estos directorios es el siguiente:

- *CMakeLists_Luna_Kepler*: Ficheros de configuración de dependencias de librerías de los proyectos desarrollados. Debido a que durante la realización del Proyecto Fin de Grado se lanzó una nueva versión del entorno de desarrollo *Eclipse,* se proporciona un fichero de configuración para cada versión del mismo.
- *doc*: Memoria y anexos desarrollados.
- *HAnDy_library*: Ficheros fuente necesarios para la implementación de la librería desarrollada en este Proyecto Fin de Grado.
- *projects*[16]: Los proyectos desarrollados para componer la cadena de procesado de imágenes hiperespectrales.
- *scripts*: Scripts de configuración, compilación y ejecución de cada uno de los proyectos aportados en el directorio *projects*.

A continuación, aportamos una breve explicación del contenido de cada uno de los directorios mencionados anteriormente.

A.3.1. CMakeLists_Luna_Kepler

En la figura A.3.2 podemos observar los dos ficheros que componen este directorio.

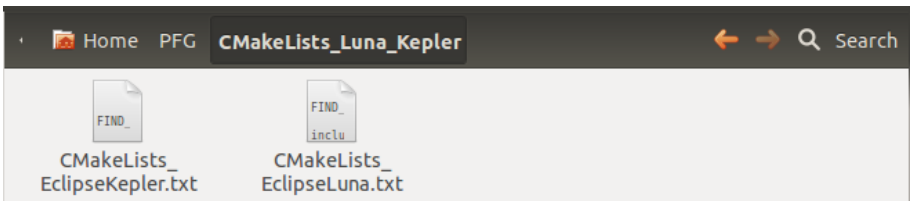


Fig. A.3.2. Contenido del directorio CMakeLists_Luna_Kepler

El primero de ellos es la configuración del programa en el caso de que se utilice Eclipse Kepler; por su parte, el segundo es el equivalente para el caso de utilizar Eclipse Luna.

[16] Todos los proyectos que se proporcionan están desarrollados en Eclipse Luna; si su versión es anterior, deberá regenerar el código Orcc antes de ejecutar el *script* correspondiente.

A.3.2. doc

El contenido de este directorio se muestra en la figura A.3.3, en ella se observa que, en la carpeta *doc*, se recogen tanto la memoria como sus respectivos anexos en formato *.pdf*.

Fig. A.3.3. Contenido del directorio doc

A.3.3. HAnDy_library

En este directorio se encuentran todos los archivos que componen la librería desarrollada a lo largo del Proyecto Fin de Grado. Los ficheros que la componen son los que aparecen en la figura A.3.4.

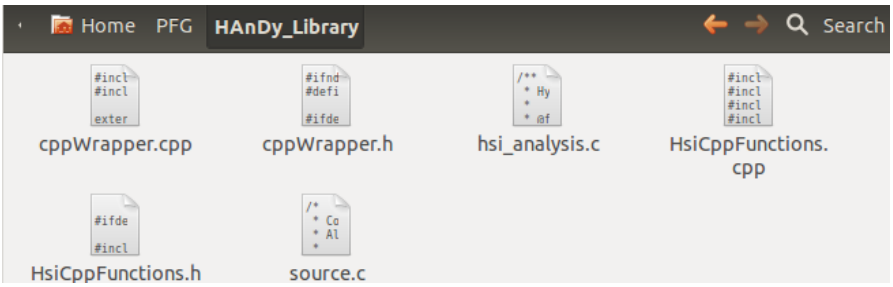


Fig. A.3.4. Contenido del directorio HAnDy_library

Como podemos observar, en la primera versión de esta librería se ha necesitado desarrollar código en lenguaje C y, a su vez, código en lenguaje C++, por lo que ha sido indispensable crear un *wrapper* entre ambos lenguajes para poder hacer compatible su utilización. La razón por la que se utiliza lenguaje C++ es que existen funciones de alta complejidad matemática pertenecientes a las librerías que se instalaron y configuraron en el Anexo 1 (ITK,OTB y VXL), que están implementadas en este lenguaje.

Una vez explicado el por qué de la utilización de dichos ficheros, aportamos una breve explicación del contenido de cada uno de los archivos presentes en este directorio:

- *source.c*: Este archivo contiene las funciones relacionadas con la lectura de ficheros binarios que contienen la información de la imagen hiperespectral. Es necesario para el desarrollo de la interfaz de entrada del sistema.
- *hsi_analysis.c:* En este fichero se encuentran las funciones básicas (en lenguaje C) y las llamadas a funciones complejas (en lenguaje C++) de la librería desarrollada. Este fichero contiene el grueso de las funciones desarrolladas en esta librería.
- *cppWrapper.h*: En este fichero se recogen los prototipos de las *funciones enlace*[17] entre lenguaje C y lenguaje C++.
- *cppWrapper.cpp*: Este fichero sirve de enlace entre ambos lenguajes. Contiene las *funciones enlace* y, a su vez, realiza la llamada a la función en lenguaje C++ correspondiente.
- *HsiCppFunctions.h*: En este fichero se recogen los prototipos de las funciones implementadas en C++.
- *HsiCppFunctions.cpp*: Este fichero incluye todas las funciones desarrolladas en C++ necesarias para el correcto funcionamiento del sistema.

[17] *Función enlace*: Mecanismo que posibilita la llamada a métodos C++ desde funciones C.

A.3.4. Projects

Este directorio contiene los proyectos que se han desarrollado para la comprobación del correcto funcionamiento de la librería. En la figura A.3.5 podemos observar los tres proyectos finales realizados. También se puede ver que aparece un cuarto directorio denominado *RemoteSystemsTempFiles*; esto se debe a que este directorio también es utilizado como *workspace* del entorno de desarrollo Eclipse, y es creado de forma autónoma por dicha herramienta.

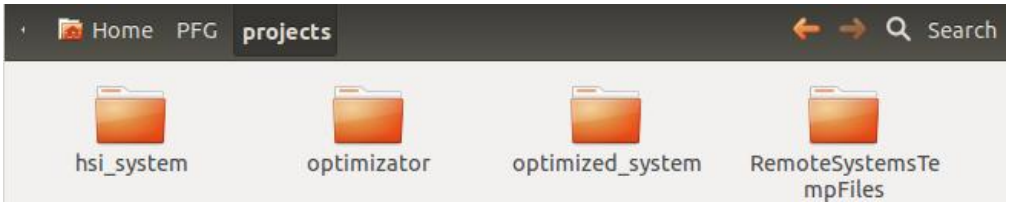


Fig. A.3.5. Contenido del directorio projects

La explicación de los tres proyectos realizados es la siguiente:

- *hsi_system*: En este proyecto se encuentra la cadena completa de análisis de una imagen hiperespectral. Esta cadena se encuentra dividida en actores, de tal forma que en cada uno de ellos se implementa una etapa de la cadena (PCA, VCA y LSU). Asimismo, también se aportan un actor *Source* que permite leer un fichero binario y un actor *Display* que genera la salida - también en un fichero binario -, constituyendo, por tanto, las interfaces de entrada y salida de la cadena de análisis.
- *optimizador*: Este proyecto contiene la etapa opcional (estimación del número de *endmembers* óptimo - *HySime*) generando a su salida un fichero denominado *Data.cal* que funciona como archivo de especificaciones (configuración de bandas y *endmembers* deseados) en el proyecto *hsi_system*.
- *optimized_system*[18]: En este fichero se aúnan todas las fases del análisis de la imagen hiperespectral, uniendo la fase de estimación de *endmembers* con las fases propias del análisis de una imagen. En este proyecto, el análisis de la imagen se hará siempre con el número óptimo estimado de bandas y *endmembers* de una imagen.

Tras la explicación de los proyectos realizados, pasamos ahora a estudiar la organización interna de los mismos. Para ello, explicaremos únicamente la estructura de uno de ellos - por ejemplo, del proyecto *hsi_system*-, puesto que dicha estructura es común a todos los proyectos. En primer lugar, la estructura general de cualquier proyecto es la que aparece en la figura A.3.6.

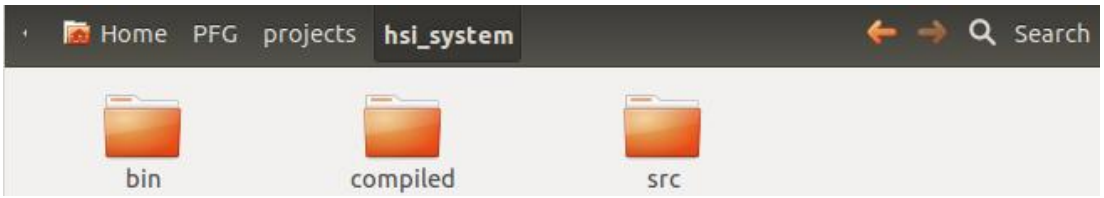


Fig. A.3.6. Contenido del directorio hsi_system

Como podemos observar, la estructura general se divide en tres directorios:

- *bin*: Este directorio contiene archivos de extensión *.ir* necesarios para la generación automática de código. Estos archivos son autogenerados por el entorno Eclipse.
- *src*: En este directorio se encuentran los archivos con extensión *.cal* que se utilizan en Orcc para implementar actores y definir *networks*.
- *compiled*: Este es el directorio de salida del código generado por Orcc y que, por lo tanto, tendrá el programa en lenguaje C y deberá incluir la librería generada.

[18] En este proyecto se modifica el código autogenerado por Orcc, por lo que cualquier cambio efectuado en el código proporcionado será ignorado.

El contenido de este último directorio se puede observar en la figura A.3.7.

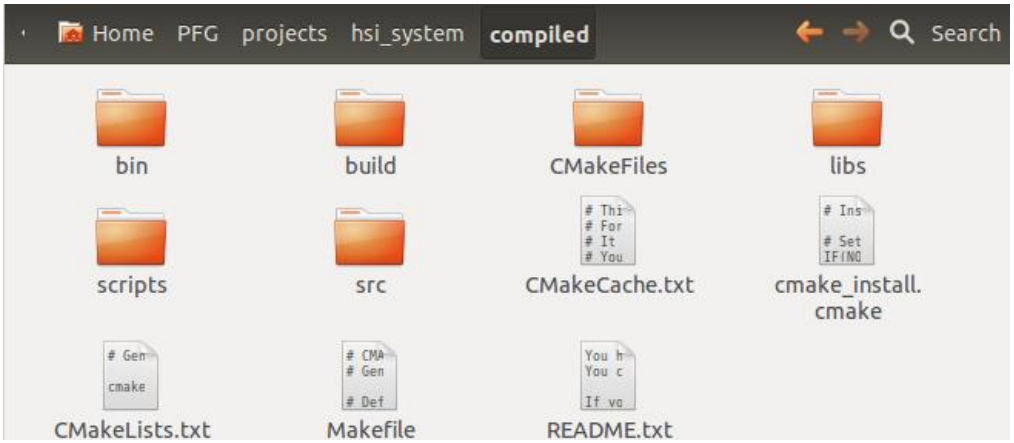


Fig. A.3.7. Contenido del directorio compiled

De los directorios que incluye la carpeta *compiled*, los que son de interés para el usuario son lo siguientes:

- *bin*: En este directorio se genera el ejecutable del programa realizado y, además, se almacena la entrada y se genera la salida de los distintos programas, tal y como podemos observar en la figura A.3.8.

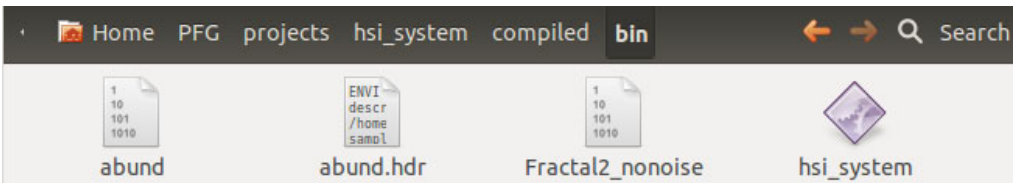


Fig. A.3.8. Contenido del directorio bin *de un proyecto*

- *src*: Como se puede observar en la figura A.3.9, en este directorio se encuentra el código autogenerado por Orcc en lenguaje C, así como un fichero de configuración en formato *.xcf* para organizar la distribución del programa entre varios procesadores.

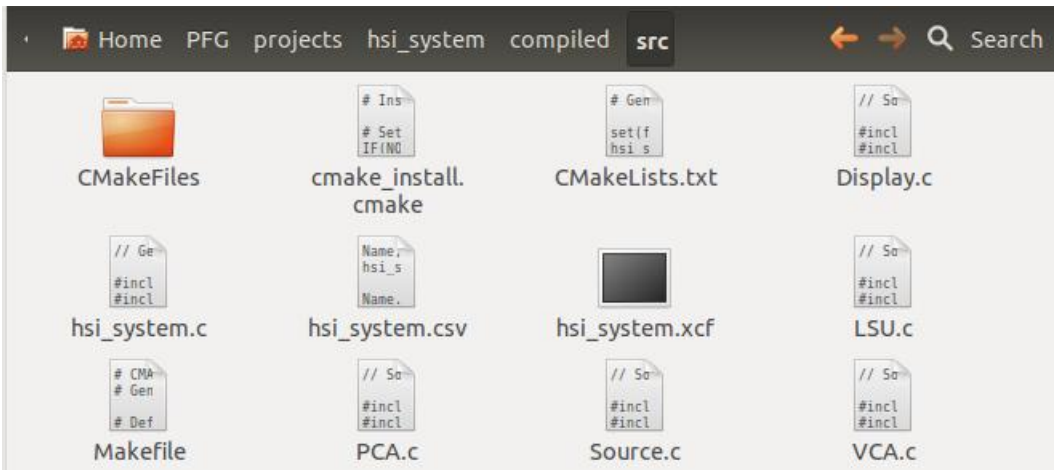


Fig. A.3.9. Contenido del directorio src *de un proyecto*

– *libs*: En este directorio se encuentran todas las librerías necesarias para que el programa se ejecute de manera satisfactoria. Esto implica que la librería desarrollada debe ser incluida en este directorio si queremos que funcione correctamente. En concreto, dicha librería está fomada por funciones nativas, por lo que, para seguir con la organización de Orcc, habría que incluirla -junto al *CMakeLists.txt* de configuración- en el directorio *orcc-native* que se observa en la figura A.3.10.

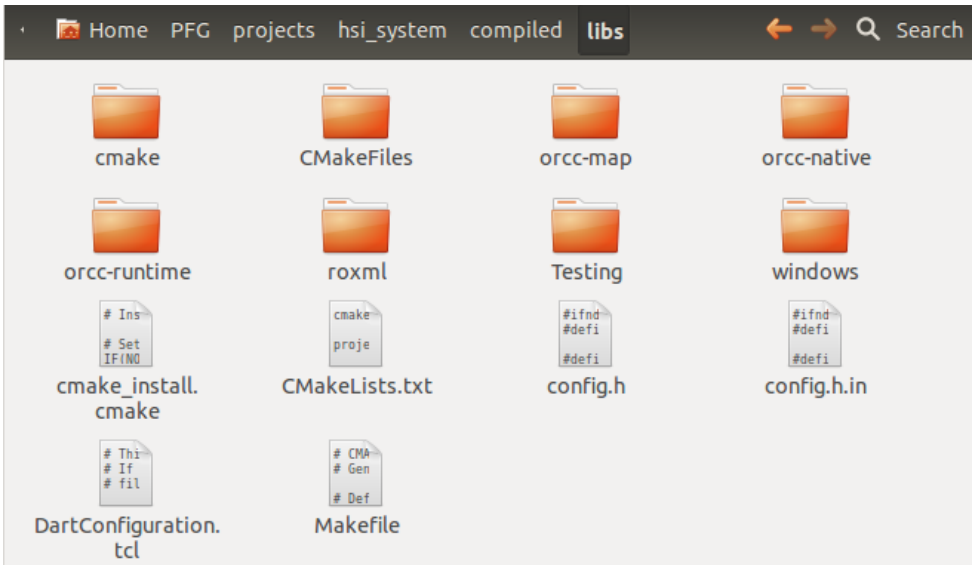


Fig. A.3.10. Contenido del directorio libs *de un proyecto*

A.3.5. Scripts

En este último directorio se aportan los *scripts* de configuración, compilación y ejecución de cada uno de los proyectos. Estos ficheros incluyen la copia automática de todos los ficheros de la librería en sus directorios correspondientes, la compilación, también automática, y la ejecución con los parámetros concretos propios de cada programa. Dicho directorio está organizado, como se puede observar en la figura A.3.11, por proyectos.

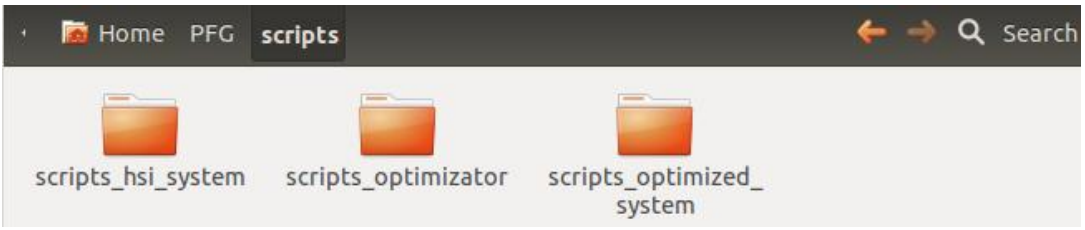


Fig. A.3.11. Contenido del directorio scripts

En cada uno de los directorios se aporta un *script* propio para cada versión de Eclipse, tal y como observamos en la figura A.3.12.

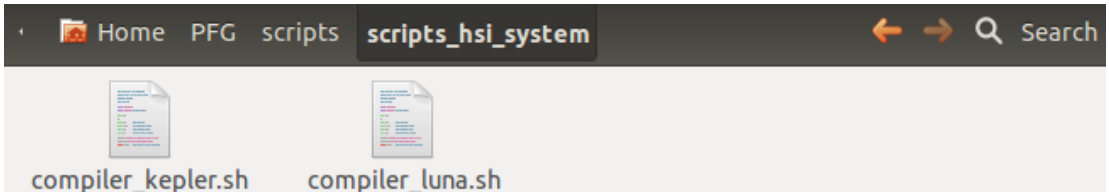


Fig. A.3.12. Contenido del directorio scripts_hsi_system