Departamento de
Informática e Ingeniería
de Sistemas
Universidad Zaragoza

Proyecto Fin de Carrera de Ingeniería Industrial

Procesamiento de imágenes a través de métodos variacionales y de optimización convexa

Francisco Javier Casanaba Benedé

Director: Lina María Paz

Departamento de Informática e Ingeniería de Sistemas
Centro Politécnico Superior

Septiembre de 2013

A mi familia, muy especialmente a mi hermano, quien seguro superará todas las piedras que la ingeniería ponga en su camino.

Hay una fuerza motriz más poderosa que el vapor, la electricidad y la energía atómica: la voluntad.
Albert Einstein

Agradecimientos

Me gustaría que estas líneas sirvieran para expresar mi enorme gratitud a todas aquellas personas que me han ayudado durante la realización de este proyecto, con mención especial a mi tutora Lina María Paz, quien con una generosidad fuera de lo común siempre tuvo un hueco para mí cuando lo necesité.

Sergio, mi compañero de fatigas, se merece un apartado exclusivo en este capítulo, pues su compañía y conocimientos de programación durante estos meses han hecho mucho más agradable mi trabajo.

Finalmente, agradecer a mi familia y amigos su comprensión y apoyo durante una etapa en la que no he podido atenderles todo lo se merecen.

A todos ellos, de corazón, muchas gracias.

Índice

Índice de figuras

Resumen

En muchas aplicaciones de la visión por computador como por ejemplo, la reconstrucción automática de entornos en 3D, se parte del supuesto de la adquisición de imágenes de alta calidad para obtener soluciones de gran precisión. Adicionalmente, una gran variedad de aplicaciones en la robótica usa sensores de visión embebidos en plataformas móviles para llevar a cabo tareas de localización y reconocimiento de lugares. Desafortunadamente, en la mayoría de los casos los sensores de visión usados para estas tareas sufren diferentes efectos que deterioran la calidad de las imágenes, por ejemplo se puede considerar el efecto del blurring en imágenes que ocurre durante la exploración en entornos bajo condiciones de poca iluminación o navegación con plataformas que llevan a cabo movimientos de dinámicas considerables.

Entre los problemas mas interesantes a tratar dentro del procesamiento de imágenes, se encuentran los siguientes: 1-Filtrado de ruido (denoising): es el proceso mediante el cual la imagen debe ser recuperada filtrando el ruido al que se encuentra expuesta inicialmente. 2-Deconvolución (deconvolution): es el proceso de corrección de una imagen generalmente mediante técnicas frecuenciales cuando los píxeles se ven afectados por un movimiento brusco creando un efecto de blurring. 3-Escalado (Zooming): en varias aplicaciones, la adquisición de imágenes se ve limitada al uso de baja resolución debido al ancho de banda de transmisión; el escalado permite interpolar valores de intensidad de píxel para obtener imágenes de alta resolución donde los objetos se pueden apreciar de forma consistente. 4-Restauración de imágenes (inpainting) es un proceso que permite recuperar una parte deteriorada de la imagen o que tiene algún objeto que la oculta, con el objetivo de mejorar su calidad.

En este proyecto se ha desarrollado una aplicación que permite tratar los diferentes problemas del procesamiento de imágenes descritos en los puntos 1-4. El algoritmo principal para la solución de los distintos problemas se basa en la formulación de métodos variacionales y de optimización convexa. Son métodos complejos que permiten usar distintas normas robustas de error (incluso no diferenciables) tales como la norma de Huber y la variación total. El algoritmo usado en este proyecto ha sido adaptado a los diferentes problemas bajo una implementación rápida y eficiente a través del cálculo masivo paralelo usando tarjetas gráficas GPU (graphics processing units). Estas características resultan particularmente atractivas para resolver problemas de la visión por computador donde las soluciones en tiempo real juegan un papel importante.

Capítulo 1

Introducción

El uso creciente de cámaras dentro del campo de la robótica aplicada en entornos industriales ha incentivado el desarrollo del campo de la visión por computador. Actualmente, es habitual encontrar sistemas de cámaras fijas para la supervisión de productos en cadenas de montaje, así como robots equipados con cámaras empleados en otros procesos importantes de fabricación. A diferencia de otros sensores como el láser, las cámaras son escogidas debido a la cantidad información contenida en las imágenes y su bajo costo.
Sin embargo, ¿qué pasa cuando la información contenida en las imágenes sufre un deterioro? Si se toma una imagen en una cadena de montaje relativamente poco iluminada, la cámara debe emplear un tiempo de exposición relativamente alto, sufriendo el efecto comúnmente conocido como "motion blurring". Si tras tomar una imagen, ésta es enviada a un computador para que extraiga su información, es posible que se adhiera ruido a la imagen debido a defectos en el canal de transmisión, a las condiciones climatológicas o defectos de la propia cámara, complicando la tarea de cualquier algoritmo de reconocimiento al que se quiera someter la imagen. En ocasiones, el ancho de banda del canal de transmisión restringe el envío de imágenes de alta resolución siendo necesaria una compresión de datos. Durante la descompresión, las imágenes pierden información y se requieren algoritmos de súper resolución o zooming capaces de estimar la información para minimizar la pérdida de datos. Otro de los efectos interesantes surge debido a la obstrucción de conjuntos vecinos de píxeles o regiones completas de la imagen. Consideremos por ejemplo la obstrucción debida a suciedad en las lentes de las cámaras o el deterioro sufrido en las cintas antiguas de vídeo. La restauración de imagen o inpainting trata de recomponer la imagen prediciendo lo que debería haber en esa zona obstruida.
El objetivo principal de este proyecto es aplicar algoritmos de procesamiento de imágenes para solucionar los diferentes efectos mencionados para su posterior utilización en aplicaciones industriales y de la robótica. Por simplicidad, en este proyec-

to se hará referencia a los diferentes problemas de acuerdo a los nombres recibidos en inglés: denoising, zooming, deconvolution (motion deblurring) e inpainting.
En este proyecto se aborda el modelado de cada problema desde el punto de vista de los modelos continuos variacionales. El porqué de esta selección se debe al gran potencial que tienen dichos métodos para tratar la información densa de forma robusta. Así mismo, las soluciones proporcionadas para dichos métodos están basadas en los avances desarrollados dentro del campo de la optimización convexa. Desde este punto de vista, los algoritmos derivados permiten calcular soluciones globales de forma eficiente debido al tratamiento independiente de los píxeles. Como resultado, los algoritmos usados en este proyecto permiten la explotación del cálculo masivo paralelo en GPGPU (General Purpose Graphich Processing Unit) siendo de gran importancia en las aplicaciones de tiempo real.
En este capítulo se introducen las bases teóricas de los algoritmos empleados en el proyecto. En la sección 1.1, se expondrán los conceptos matemáticos sobre los que se sustentan los algoritmos empleados. Así, se comienza definiendo la transformación de Legendre-Fenchel, cómo y por qué se aplica en la resolución del algoritmo. Se continúa definiendo la dualidad, qué es el problema dual y cómo y por qué resolverlo. Para finalizar, se tratan los problemas de funciones no siempre diferenciables y cómo poder transformarlos en una función convexa resoluble. En la sección 1.2, se incluye la definición de la función de energía, eje principal de los modelos que se han utilizado en este proyecto. Al tratarse de un método numérico, se debe definir un criterio de parada, así surge el concepto del GAP descrito en la sección 1.3. Finalmente, la sección 1.4 muestra la organización de la memoria del proyecto, suponiendo el fin de este capítulo de introducción.

1.1. Conceptos generales

Los modelos usados para cada problema concreto se basan en normas que son convexas pero no diferenciables, de forma que no se pueden utilizar algoritmos de optimización convencionales. Además, el problema general consiste en la minimización de una función de energía, función que no es necesariamente convexa y por tanto no se puede alcanzar una solución global. Con el fin de solucionar esta situación, se aplicará la transformación de Legendre Fenchel (Apartado 1.1.1).
Esta transformación requiere dualizar una función, motivo por el que se explica el concepto de dualización y cómo se debe aplicar. Más tarde se expone un ejemplo de funciones no siempre diferenciables donde se dualiza con el fin de generar un problema diferenciable en todo su rango. Para finalizar, se comenta la necesidad de partir de una función convexa para el algoritmo de solución, pues de lo contrario no se podría obtener una solución general. [2].

1.1.1. La transformación de Legendre-Fenchel

La transformada de Legrendre-Fenchel (LF en adelante) [7] de una función continua pero no necesariamente diferenciable se define como

$$f^*(p) = \sup_{x \in \mathbb{R}} \{px - f(x)\} \tag{1.1}$$

La figura 1.1 ilustra la transformación LF de una función cuadrática. Geométricamente se trata de la búsqueda de un punto $(x, f(x))$ para el cual una recta de pendiente p produzca el máximo corte con el eje vertical. El conjunto de rectas tangentes forma una envolvente que representa a sí misma la función.

$$p = f'(x) \tag{1.2}$$

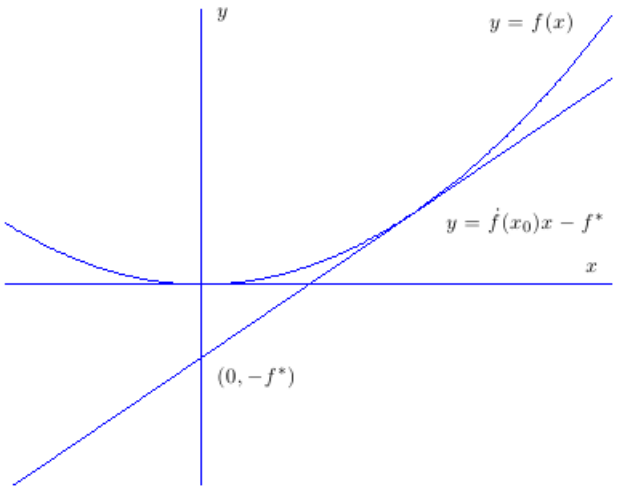


Figura 1.1: Recta tangente a una función cuadrática

La representación general vectorial de la transformación LF para funciones multivariables se define como:

$$f^*(\mathbf{p}) = \sup_{\mathbf{x} \in \mathbb{R}} \{\mathbf{x}^t\mathbf{p} - f(\mathbf{x})\} \tag{1.3}$$

1.1.2. La dualidad

La dualidad es el principio por el que observamos una misma función desde dos diferentes perspectivas: primal y dual. Si se supone que la transformación LF es reversible, se puede afirmar que

$$(x, f(x)) \Longleftrightarrow (p, f^*(p)) \tag{1.4}$$

donde p es la pendiente y *f*(p)* es el llamado conjugado convexo de la función *f(x)*. Un conjugado nos permite construir un problema dual que puede ser más fácil de resolver que el problema primal. Observando la figura 1.1, se puede afirmar que cada punto de la función primal se puede representar por la pendiente de la tangente a ese punto, obteniendo el dual. De esta manera, los puntos del dual son las pendientes del primal, y los puntos del primal son las pendientes del dual. El conjugado de LF es siempre convexo.

1.1.3. Funciones no siempre diferenciables

Observemos la ecuación [2] definida al principio, donde p es la pendiente tangente a $f(x)$ en el punto x definida como la derivada $f'(x)$ ¿Qué sucede si la p no es diferenciable en todo rango de x? La ecuación común de una recta es la siguiente:

$$y = px - c \tag{1.5}$$

Si se modifica esta ecuación para que se refiera a un punto x^* no diferenciable, se puede reescribir como

$$f(x^*) = px^* - c \tag{1.6}$$

El problema dual de este punto no diferenciable se convierte en una función lineal en p. A pesar de la simplicidad de este ejemplo lineal, este mismo principio puede emplearse en el caso de funciones no diferenciables ya que el resultado es una nueva función dual en p que puede ser diferenciable. Una de las grandes ventajas del problema dual es que pese a que el primal puede ser no diferenciable, el problema dual sí que lo es.

1.1.4. Funciones convexas

Los problemas de procesamiento de imágenes tratados en este proyecto se modelan como un problema de minimización de una función de energía. De forma general, el punto $\hat{x}$ es el mínimo global de una función siempre y cuando $\nabla f(\hat{x}) = 0$ y la función sea convexa. Sin embargo, existen muchas funciones que no son siempre diferenciables o convexas, por lo que no se puede realizar el cálculo de su gradiente. La figura 1.2 ilustra un conjunto de funciones convexas.

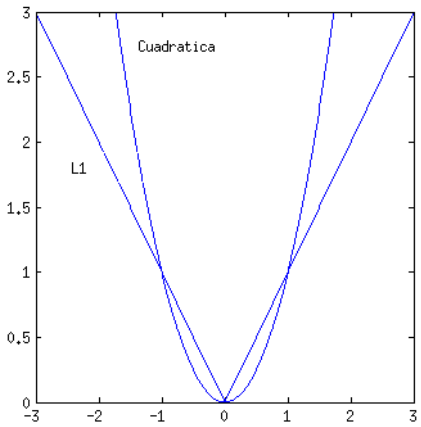


Figura 1.2: Ejemplos de funciones convexas

La función cuadrática es continua y diferenciable en todo su dominio. No así la función valor absoluto, que en el punto $x = 0$ no es diferenciable. En este último caso, se puede demostrar que la transformada de LF es convexa. Por tanto, basta con realizar la transformación de LF a una función para obtener una función convexa.

1.2. La función de energía

Los métodos de procesamiento de imágenes (Image Processing en inglés) se basan en la minimización de una función de energía. Esta función está formada por dos términos: El primero de ellos es el llamado Regularizador. Este término es el que cuantifica la variación dentro de una propia imagen, tratando que las diferentes superficies tengan una textura uniforme y las esquinas y bordes queden lo más definidas posibles. El segundo término es el denominado "Data Term", que es el que cuantifica cuán distinta es la imagen solución (imagen arreglada) de la imagen original (imagen defectuosa). Al aplicar estos métodos se parte de la base de que la imagen defectuosa contiene aún suficiente información para ser restaurada, aunque no en su totalidad, pero sí con un alto porcentaje de mejora sobre su estado inicial. En general, la ecuación de la función de energía a minimizar es la siguiente:

$$\min_{x \in X} E(x) \tag{1.7}$$

$$\min_{x \in X} F(Kx) + G(x) \tag{1.8}$$

Donde $K : X \to Y$ es un mapa lineal entre dos espacios vectoriales X e Y equipados con un operador de producto escalar $< \cdot, \cdot >$ y una norma $| \cdot | = |\cdot, \cdot|$. $G : X \to [0, +\text{ínf})$ y $F^* : X \to [0, +\text{ínf})$ son funciones propias convexas, semi-continuas inferiormente [6].

El primer término de la ecuación es el regularizador, el cual usa como única información la variable primal. El segundo cuantifica la variación de la variable primal respecto a su estado inicial. En cada caso particular de restauración de imágenes, la función de energía será redefinida para una variable primal que corresponderá a la imagen solución definida en el dominio $\Omega \in \mathbb{R}$.

A continuación se dualiza esta función del que resulta un problema de minimización y maximización sobre las variables primal y dual respectivamente.

$$\min_{x \in X} \max_{y \in Y} \{\langle Kx, y \rangle - F^*(y) + G(x)\} \tag{1.9}$$

Este resultado es fundamental para derivar los algoritmos de optimización empleados en este proyecto.

1.3. El GAP

Una vez definida la función, es necesario generar un parámetro que nos muestre de una forma clara y concisa cuánto ha sido maximizada y minimizada la función de energía. Ese valor toma el nombre de GAP.
Para generar una función que defina el GAP se hace uso de la definición de transformada LF, ya mencionada con anterioridad. Así, si se observa la ecuación anterior, se puede apreciar que

$$F(Kx) = \max_{y \in Y} \{\langle Kx, y \rangle - F^*(y)\} \tag{1.10}$$

En esta ecuación se puede modificar el término $\langle Kx, y \rangle$ y transformarlo a $\langle x, K^*y \rangle$, obteniendo por tanto la ecuación

$$\min_{x \in X} \max_{y \in Y} \{\langle x, K^*y \rangle - F^*(y) + G(x)\} \tag{1.11}$$

Si se realiza el siguiente procedimiento de la misma forma que el anterior:

$$\begin{aligned} \max_{x \in X} \{\langle x, K^*y \rangle - G(x)\} &= G^*(K^*y) \\ \min_{x \in X} \{-\langle x, K^*y \rangle + G(x)\} &= -G^*(K^*y) \\ \min_{x \in X} \{-\langle x, -K^*y \rangle + G(x)\} &= -G^*(-K^*y) \\ \min_{x \in X} \{\langle x, K^*y \rangle + G(x)\} &= -G^*(-K^*y) \end{aligned} \tag{1.12}$$

Por lo que aparecen dos optimizaciones derivadas de la función de energía:

$$\begin{aligned} &\min_{x\in X}\{F(Kx)+G(x)\} \\ &\max_{y\in Y}\{-G^*(-K^*y)-F^*(y)\} \end{aligned} \tag{1.13}$$

El GAP se define como la diferencia entre ambas optimizaciones:

$$GAP = \min_{x\in X}\{F(Kx)+G(x)\} - \max_{y\in Y}\{-G^*(-K^*y)-F^*(y)\} \tag{1.14}$$

Es el término más preciso de optimización, pues la función de energía dualizada debe ser minimizada respecto a una variable y maximizada respecto a otra (ver figura 1.3). El GAP es decreciente (salvo en los instantes finales de convergencia, donde aparece algo de ruido) y aporta una idea muy sencilla de cuánto se está optimizando la función. No obstante, el cálculo de GAP en ciertas aplicaciones puede ser muy costoso o incluso imposible, ya que requiere una dualización que no siempre se podrá llevar a cabo. Cuando ésto sucede, simplemente se comprueba que la función que se minimiza efectivamente es decreciente y alcanza un mínimo.

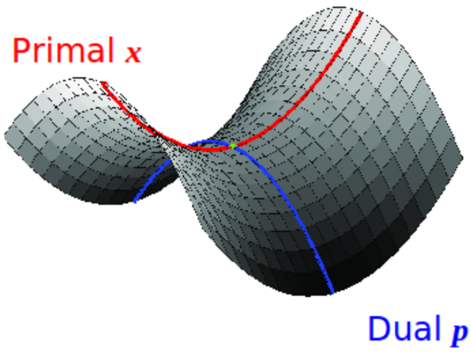


Figura 1.3: Evolución de los procesos de minimización y maximización para las variables Primal y Dual respectivamente. Así, el objetivo es alcanzar el punto silla donde la función Primal es mínima y la función Dual es máxima.

1.4. Organización de la memoria

La memoria de este proyecto se estructurará de forma que, tras un primer capítulo de introducción, se expondrán los diferentes problemas a tratar con sus diferentes secciones. De esta forma:
Denoising: Se comienza explicando la motivación por la que se desea resolver el problema, dando ejemplos prácticos donde podría ser necesaria. Denoising puede

ser resuelto de diferentes formas posibles, dependiendo de la función de energía utilizada. Así, los métodos toman un nombre asociado a su resolución. El regularizador se denomina Total Variation cuando se resuelve según la norma 2. En caso de que se resuelva mediante la norma de Huber, se denominará "Huber". El término $G(x)$ se denomina L1 si se resuelve según la norma 1 o ROF si se resuelve según la norma cuadrática. Así, los tres posibles métodos de resolución que se manejarán son TV-ROF, HUBER-ROF y TV-L1. En los casos en los que sea posible, se calculará el GAP. Una vez explicados los tres métodos, se finaliza haciendo referencia a los resultados situados en el Apéndice F.

Zooming: Se comienza con una motivación explicativa ofreciendo los motivos por los que se plantea la resolución del problema. Después se da una breve introducción al método numérico de Jacobi, necesario para la resolución del algoritmo. Una vez explicado, se deduce, explica y resuelve el algoritmo, incidiendo en las diferencias respecto al caso anterior.

Image Deconvolution: Nuevamente, se introduce el problema mediante un apartado de motivación explicando por qué se produce este fenómeno y por qué es importante solucionarlo. El siguiente apartado explica cómo obtener la máscara que genera el blurring en la imagen. Es muy importante el cálculo de esta máscara porque será necesaria en la resolución del algoritmo. A continuación se resuelve dicho algoritmo.

Inpainting: Es el último de los problemas a tratar. La estructura seguida es similar a la de los casos anteriores. Se comienza con una explicación en la que se comenta porqué puede surgir y por qué solucionarlo. A continuación, se deduce y resuelve el algoritmo.

Anexos: Donde se encuentran tanto explicaciones acerca de la programación, de matemáticas que no se introducen en la memoria por su complejidad pero que son necesarias para la resolución del algoritmo, resultados de los diferentes algoritmos y un manual de usuario final, que es la conclusión definitiva del trabajo. En este proyecto se ha trabajado en la solución de distintos problemas, obteniendo importantes datos en su resolución tales como el gap, el tiempo de ejecución, las iteraciones que tarda en converger o cómo de fiable es el resultado final respecto a imagen original, la que no sufre modificación. Estos resultados dependen del algoritmo obviamente, pero también de diferentes parámetros que pueden estar acotados por la convergencia del método, pero que poseen cierta holgura y pueden variar lo óptimo del método. Así, se ha generado una aplicación donde se permite al usuario escoger entre los cuatro problemas citados y proceder a su resolución. Cada uno de ellos tiene dos posibles modos de resolución. El primero de ellos, llamado Programa Principal permite al usuario introducir la imagen, dañarla en los casos que sea necesario y arreglarla con los parámetros que se requieran. Como resultado se obtiene la imagen resultado y una gráfica de evolución de GAP o

función de energía respecto a iteraciones según el caso. La segunda pestaña es la de Evaluación de Parámetros que es capaz de calcular el parámetro óptimo para una imagen concreta.

Capítulo 2

Denoising

2.1. Motivación

Denoising se refiere al proceso de filtrado del ruido de una imagen. En la industria es muy probable encontrar imágenes que deban someterse a un tratamiento de este tipo. Por ejemplo, en una cadena de manufactura de muebles hay gran cantidad de virutas esparcidas por el aire, que pueden perturbar la imagen. También en el envío de datos se puede añadir ruido que la perturbe. Mediante un algoritmo primal-dual somos capaces de recuperar razonablemente la imagen original, pudiendo someterla así a cualquier algoritmo de reconocimiento, de medición o de control de calidad. A continuación se va a tratar el problema utilizando tres modelos diferentes que permiten analizar el impacto al cambiar la norma tanto en el regularizador como en el data term: modelo ROF, el huber ROF y el TVL1.

2.2. El modelo TV-ROF

El modelo ROF se define como el problema variacional

$$\min_u \int_\Omega |Du| + \frac{\lambda}{2} \parallel u - g \parallel_2^2 \tag{2.1}$$

Donde u es la imagen resultado, g la imagen original (la que tiene ruido) y λ es un parámetro utilizado para definir la acomodación entre los dos términos de la ecuación. Esta ecuación es continua, como una imagen se divide en píxeles y éstos son discretos, se reescribe de la siguiente forma

$$\min_{u \in U} \parallel \nabla u \parallel_1 + \frac{\lambda}{2} \parallel u - g \parallel_2^2 \tag{2.2}$$

El conjugado convexo de una norma L_1 es una función indicadora definida en la ecuacion 2.3

$$\delta(p) \begin{cases} \delta(p) = 0 & si \parallel p \parallel \leq 1 \\ \delta(p) = \infty & \text{en cualquier otro caso} \end{cases} \tag{2.3}$$

Al aplicar la transformación de LF sobre el término de regularización representado por la norma $\parallel . \parallel_1$ se obtiene:

$$\parallel \nabla u \parallel_1 = \max_{p \in P}(\langle p, \nabla u \rangle - \delta_p(p)) \tag{2.4}$$

Se puede expresar este problema de la forma primal-dual de la siguiente manera:

$$\min_{u \in U} \max_{p \in P} \langle p, \nabla u \rangle + \frac{\lambda}{2} \parallel u - g \parallel_2^2 - \delta_p(p) \tag{2.5}$$

Esta función es la función de energía sobre la que se va a trabajar. Para calcular su mínimo respecto al primal y su máximo respecto al dual, se debe derivar parcialmente respecto a esas variables. La derivada de la energía respecto a la variable dual se establece como:

$$\partial_p E(u,p) = \partial_p(\langle p, \nabla u \rangle + \frac{\lambda}{2} \parallel u - g \parallel_2^2 - \delta_p(p)) \tag{2.6}$$

Por tanto se puede demostrar el siguiente resultado:

$$\partial_p E(u,p) = \nabla u \tag{2.7}$$

Se realiza el mismo procedimiento para la variable primal:

$$\partial_u E(u,p) = \partial_u(\langle p, \nabla u \rangle + \frac{\lambda}{2} \parallel u - g \parallel_2^2 - \delta_p(p)) \tag{2.8}$$

De la misma manera, se obtiene que:

$$\partial_u E(u,p) = -divp + \lambda(u - g) \tag{2.9}$$

El Apéndice B incluye un desarrollo completo de estas derivadas. Las ecuaciones 2.7 y 2.9 son necesarias para derivar los algoritmos iterativos de gradiente descendente y ascendente para actualizar las variables primal y dual respectivamente. En el problema de maximización de la variable dual se requiere la aproximación del gradiente de manera que siempre se avance en la dirección ascendente, ésto significa ir en la dirección del gradiente, por tanto, el paso de actualización se deriva como:

$$\partial_p E(u,p) = \nabla u = \frac{p^{n+1} - p^n}{\sigma} \tag{2.10}$$

El denominador de esta última ecuación es es el resultado de la proyección sobre una bola [7] de la variable dual de manera que se cumple con la restricción impuesta por la función indicadora δ_p definida en la ecuación 2.3. Con un procedimiento similar, se aproxima la derivada de la función respecto a la variable primal. Dado que se trata de un problema de minimización, la derivada se aproxima como un paso de gradiente descendente, obteniéndose el siguiente resultado:

$$\partial_u E(u,p) = -(-divp + \lambda(u^{n+1} - g)) = \frac{u^{n+1} - u^n}{\tau} \tag{2.11}$$

Donde el signo negativo que se le asigna a $\partial_u E(u,p)$ se debe a que en cada paso se busca la dirección en el sentido contrario del gradiente.

Para agilizar el proceso y lograr una rápida convergencia, se puede introducir un paso de relajación. A continuación se muestra la estructura general del algoritmo primal-dual:

$$\begin{cases} p^{n+1} = \frac{p^n + \sigma\nabla u}{\max(1, |p^n + \sigma\nabla u|)} \\ u^{n+1} = \frac{u^n + \tau div p^{n+1} + \tau\lambda g}{1+\tau\lambda} \\ \theta_n = \frac{1}{\sqrt{1+2\gamma\tau}}, \quad \tau_{n+1} = \theta_n\tau_n, \quad \sigma_{n+1} = \frac{\sigma_n}{\theta_n}\sigma_n \\ \hat{u}^{n+1} = u^{n+1} + \theta_n(x^{n+1} - x^n) \end{cases} \tag{2.12}$$

Donde en cada iteración se actualizan las variables primal y el dual así como una variable auxiliar $\hat{u}$ la cual introduce el paso de relajación. Los parámetros τ y σ definen la escala del paso dado en la dirección positiva o negativa del gradiente en cada caso [6]. Más adelante se hablará de dichos parámetros.

2.2.1. Cálculo del GAP

Se parte de la ecuación 1.14 definida en el capítulo 1, la cual se trae aquí por conveniencia:

$$GAP = \min_{x\in X}\{F(Kx) + G(x)\} - \max_{y\in Y}\{-G^*(-K^*y) - F^*(y)\} \tag{2.13}$$

La minimización es relativamente sencilla, pues corresponde con la función de energía ya aportada. Es el la maximización del dual donde se debe operar. Se comienza con la simplificación de que $F^*(y) = 0$. la función $G(u)$ en el modelo TV-ROF se define como:

$$G(u) = \frac{\lambda}{2} \parallel u - g \parallel_2^2 \tag{2.14}$$

La dualización de dicha función corresponde a:

$$G^*(p) = \sup_u(u^T p - \frac{\lambda}{2} \parallel u - g \parallel_2^2) \tag{2.15}$$

Para calcular el supremo, se deriva respecto a la variable u y se iguala a 0. Se sustituye el valor de u que hace máxima la función y se obtiene el valor de $G^*(p)$:

$$\partial_u(u^T p - \frac{\lambda}{2} \parallel u - g \parallel_2^2) = p - \lambda(u - g) = 0 \tag{2.16}$$

$$G^*(p) = \frac{\parallel p \parallel^2}{2\lambda} + g^T p$$

Se sustituye el valor p dual por la variable dual que exige la formulación anterior $p = -k^* y$

$$G^*(-k^* y) = \frac{\parallel -k^* y \parallel^2}{2\lambda} + g^T - k^* y \tag{2.17}$$

Donde el valor de $k^* y$ es la divergencia de la variable dual del algoritmo. Al final, para calcular el GAP en cada iteración, la ecuación a aplicar es la siguiente:

$$GAP = |\nabla u| + \frac{\lambda}{2} \parallel u - g \parallel - \frac{\parallel divp \parallel^2}{2\lambda} - g^T divp \tag{2.18}$$

2.2.2. Los parámetros del algoritmo

En el algoritmo calculado en los apartados anteriores aparecen una serie de parámetros con un significado físico que requiere una explicación para comprender mejor el significado de las ecuaciones:

El parámetro L

La L es el límite superior de la norma del operador ∇. Se acota su valor a $\sqrt{8}$ para asegurar la convergencia del método [3]. Se supone constante a lo largo de todo el algoritmo debido a que el operador de gradiente no cambia.

El parámetro λ

λ es un parámetro que pondera el peso del término de datos respecto al término de regularización en la función de energía. De esta forma, un valor pequeño de λ tendrá como consecuencia en el algoritmo una mayor minimización en el término del regularizador que en el término de comparación con la imagen defectuosa. Adelantando lo que se expondrá en la sección de análisis de resultados, en la figura 2.1 se muestra un ejemplo de denoising para un valor de λ elevado y valor λ pequeño:

(a) Imagen con ruido $\sigma = 0,1$ (b) Imagen resultado $\lambda = 1$ (c) Imagen resultado $\lambda = 1000$

Figura 2.1: Resultados para valores extremos de λ

λ tan sólo varía el resultado final y no tiene un efecto directo sobre la convergencia del algoritmo. Para cuantificar la influencia del parámetro λ respecto a la precisión, se calcula la potencia de la razón Señal/Ruido (SnR), definida como:

$$P_{SnR} = 10 \log_{10}\left(\frac{\sum_{i,j}^{N,M} I_o(i,j)^2}{\sum_{i,j}^{N,M} (ui, j - I_o(i,j))^2}\right) \tag{2.19}$$

Este valor compara cómo de fiel es el resultado final respecto a la imagen verdadera, es decir, la imagen antes de ser degradada. Por este motivo, el cálculo del SnR sólo se puede realizar para obtener una idea cualitativa de qué rango de parámetros son correctos para determinados algoritmos. La señal se define como la suma del valor de todos los píxeles de la imagen verdadera I_o al cuadrado. El ruido se define como la diferencia entre la imagen solución u y la imagen original I_o, también al cuadrado. Si se lograse obtener la imagen perfectamente arreglada, el valor de SnR sería infinito, por lo que el valor óptimo de λ será aquel que tenga el mayor valor de SnR.

El parámetro τ

τ actualiza la variable primal u dentro de cada iteración del algoritmo de optimizaciones. Es uno de los llamados "parámetros de convergencia", ya que su valor aumenta o disminuye las iteraciones que necesita el método para converger. τ suele estar acotado para asegurar que el método converge, y su valor siempre se encuentra en el intervalo [0 1].

Para poder calcular qué valor de τ es el óptimo, se ha propuesto un análisis de error ejecutando el algoritmo primal dual para un número de iteraciones muy elevado tal que se asegure su convergencia. Se define un error máximo admisible y

en cada iteración del algoritmo se comprueba si el error entre la imagen obtenida en cada iteración u^{n+1} y la imagen solución u^* guardada previamente es menor que el error impuesto. El error se calcula según el siguiente algoritmo:

$$e = \sum_{i,j}^{N,M} u_{ij}^{n+1} - u_{ij}^* \tag{2.20}$$

El parámetro θ

θ es un parámetro de relajación y aceleración de convergencia. Se utiliza siempre para calcular el valor de $\hat{u}$, que es el empleado en la actualización del dual. Si el valor de θ es muy elevado, $\hat{u}$ varía mucho respecto a la imagen resultado de la iteración, si es pequeño serán muy similares.

Su evaluación es similar a la del parámetro τ, de nuevo requiriendo una sobreiteración del algoritmo hasta convergencia. Tras comparar los diferentes algoritmos se puede observar que las iteraciones requeridas para obtener la solución final (cuando la energía alcanza su valor óptimo) varían mucho más con τ que con θ.

2.3. El modelo Huber-ROF

En el caso anterior, el regularizador se basaba en el uso de la norma L_1. Ahora se sustituye esta norma por la llamada "norma de Huber". El proceso es similar al anterior. Se parte del siguiente modelo de energía

$$\min_{u \in U} \| \nabla u \|_\alpha^h + \frac{\lambda}{2} \| u - g \|_2^2 \tag{2.21}$$

La norma de Huber se define como

$$\| x \|_\alpha = \begin{cases} \frac{|x|^2}{2\alpha} & \text{si} \quad |x| \leq \alpha \\ \| x \| = |x| - \frac{\alpha}{2} & \text{si} \quad |x| > \alpha \end{cases} \tag{2.22}$$

Al aplicar la definición de dualización a través de la transformada LF, se obtiene el siguiente resultado para el convexo conjugado de 2.22:

$$f^*(p) = \begin{cases} \frac{|x|^2}{2} & \text{si} \quad |x| \leq \alpha \\ \infty & \text{en cualquier otro caso} \end{cases} \tag{2.23}$$

Por tanto, el problema primal dual se define como:

$$\min_{u \in U} \max_{p \in P} (\langle p, \nabla u \rangle - \delta_p(p) - \frac{\alpha}{2} \| p \|^2 + \frac{\lambda}{2} \| u - g \|_2^2) \tag{2.24}$$

Tal como se ha realizado anteriormente, se debe minimizar el primal y maximizar el dual. Para ello hay que derivar la función término a término y calcular el resultado final.

En el Apéndice B se deducen los cálculos de las derivadas respecto a las variables u y p. A partir de estos resultados, se calculan las actualizaciones en el primal y en el dual. Nótese que el resultado obtenido para la variable primal es el mismo que en el TV-ROF ya que el nuevo término añadido no depende de u.

$$\partial_p E(u,p) = \nabla u - \alpha p^{n+1} = \frac{p^{n+1} - p^n}{\sigma} \tag{2.25}$$

$$\partial_u E(u,p) = -(-divp + \lambda(u^{n+1} - g)) = \frac{u^{n+1} - u^n}{\tau} \tag{2.26}$$

Observando detenidamente las ecuaciones, se puede apreciar que el primal se actualiza de la misma forma que en el anterior modelo. En el caso concreto de $\alpha = 0$ la variable dual se actualiza como en el modelo ROF. Como la norma de Huber es convexa, es posible emplear el siguiente algoritmo de convergencia acelerada [3]:

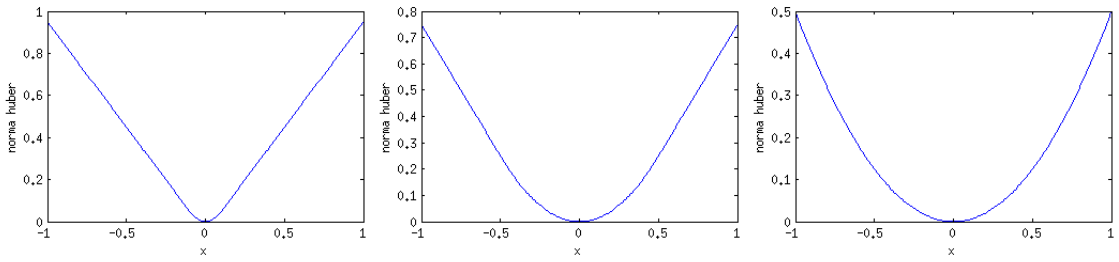
$$\begin{cases} p^{n+1} &= \dfrac{\frac{p^n + \sigma \nabla \hat{u}^n}{1+\sigma\alpha}}{\max(1, |\frac{p^n + \sigma \nabla \hat{u}^n}{1+\sigma\alpha}|)} \\ u^{n+1} &= \dfrac{u^n + \tau divp^{n+1} + \tau\lambda g}{1+\tau\lambda} \\ \hat{u}^{n+1} &= u^{n+1} + \theta(u^{n+1} - u^n) \end{cases} \tag{2.27}$$

El cálculo del GAP es muy similar al caso anterior. Esta vez el valor de $F^*(y)$ no es nulo y por tanto se debe añadir al resultado del caso anterior:

$$GAP = |\nabla u|^h + \frac{\lambda}{2} \parallel u - g \parallel - \frac{\parallel divp \parallel^2}{2\lambda} - g^T divp - \frac{\alpha}{2} \parallel p \parallel^2 \tag{2.28}$$

2.3.1. El parámetro α

Sólo tiene sentido hablar de α en el algoritmo de Huber. En este algoritmo, el término regularizador contiene la norma de Huber. Esta norma es diferente a la norma L_1, que es un valor absoluto, y a la norma ROF, que es la norma cuadrática. Huber interactúa entre estas dos normas, siendo equivalente a la primera cuando α=0 y siendo equivalente a la segunda cuando α=1.



(a) α=0.1 (b) α=0.5 (b) α=1

Figura 2.2: Variación del parámetro α de la norma de Huber

α es un parámetro que optimiza el resultado, no la convergencia. Por este motivo, al igual que λ, se debe evaluar el SnR para determinar su valor óptimo.

2.4. El modelo TVL_1

Al igual que en el modelo ROF, se usa la variación total para el término de regularización pero esta vez se introduce la norma L_1 para el término de datos.

$$\min_{u \in U} \| \nabla u \|_1 + \| \lambda(u - g) \|_1 \tag{2.29}$$

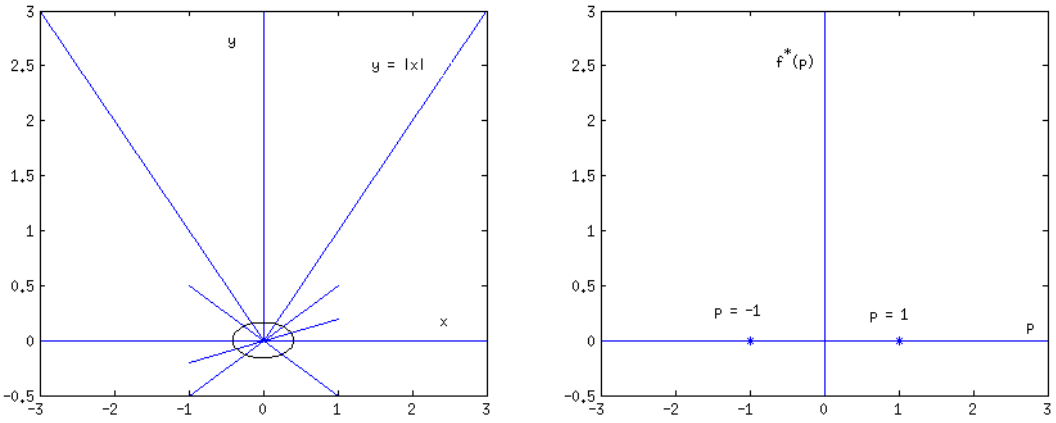
Se dualiza de nuevo el regulador, obteniendo las mismas expresiones que en el caso del modelo ROF. El problema que surge ahora es que la norma L_1 del término de comparación con la imagen original implica una nueva dualización.

$$\| \lambda(u - g) \|_1 = \max_{q \in Q}(\langle q, \lambda(u - g) \rangle - \delta_Q(q)) \tag{2.30}$$

Se tendrá que definir un nuevo convexo conjugado, realizar una ∂_q de la función y además añadir nuevos términos. La derivación es un poco más complicada y requiere del uso del concepto de subgradiente para derivar el término de datos respecto a la variable dual.

La norma L_1 puede ser definida como una función a trozos, donde:

$$\partial_u |f(u)| \begin{cases} f'(u) & si \quad u > 0 \\ -f'(u) & si \quad u < 0 \\ \text{[-f'(u), f'(u)]} & si \quad u = 0 \end{cases} \tag{2.31}$$



(a) Función valor absoluto

(b) Función derivada del valor absoluto

Figura 2.3: Ilustración del concepto de subgradiente para la función valor absoluto

Para el punto $x = 0$, puede haber un número infinito de rectas tangentes a la función en el intervalo $p \in [-f'(x), f'(x)]$. Por este motivo aunque la función es convexa, no es diferenciable. En nuestro caso concreto, se puede escribir lo siguiente

$$\partial_u \tau\lambda|u-g|_1 \begin{cases} \tau\lambda & si \quad u-g > \tau\lambda \\ -\tau\lambda & si \quad u-g < -\tau\lambda \\ indef & si \quad |u-g| \leq \tau\lambda \end{cases} \tag{2.32}$$

Este resultado se utiliza para resolver el algoritmo a partir del cálculo de las derivadas sobre la siguiente función de energía dualizada:

$$\min_{u\in U} \max_{p\in P}(\langle p, \nabla u\rangle - \delta_p(p) + \lambda \parallel u-g \parallel_1) \tag{2.33}$$

Donde:

$$\partial_p E(u,p) = \nabla u - \alpha p^{n+1} = \frac{p^{n+1}-p^n}{\sigma} \tag{2.34}$$

$$\partial_u E(u,p) = -(-divp + \partial_u \lambda \parallel u-g \parallel_1 = \frac{u^{n+1}-u^n}{\tau}) \tag{2.35}$$

Tras este desarrollo matemático, se obtiene el valor de u^{n+1}. Gracias a la ecuación 2.35, la actualización del primal se ejecuta de la siguiente manera:

$$u^{n+1} \begin{cases} u^n + \tau divp^{n+1} - \tau\lambda & si \quad u-g > \tau\lambda \\ u^n + \tau divp^{n+1} + \tau\lambda & si \quad u-g < -\tau\lambda \\ g & si \quad |u-g| \leq \tau\lambda \end{cases} \tag{2.36}$$

El algoritmo empleado es el mismo que en el caso del ROF.

Los resultados y conclusiones de los diferentes métodos del algoritmo se encuentran en el Apéndice F.1

Capítulo 3

Zooming

3.1. Motivación

Cuántas veces al reducir una fotografía y volver a ampliarla ha aparecido pixelada. Habitualmente, al expandir una fotografía cada píxel se repite por un factor de ampliación, lo que al final resulta una imagen muy poco realista, donde se aprecia con demasiada claridad que ha sido ampliada y no es una imagen original.

La solución más habitual consiste en la aplicación de una simple interpolación lineal. Si se amplía cuatro veces una imagen (el doble de alto y el doble de ancho), los píxeles generados no tienen el valor constante de su predecesor, sino una interpolación de sus vecinos. Así se consigue una sensación más homogénea en la imagen. Sin embargo, una interpolación lineal podría producir un efecto de difuminación de la imagen, eliminando los detalles.

La aplicación de un algoritmo primal dual para la minimización de una función de energía puede ser una solución al problema. Este apartado trata de su desarrollo matemático, implementación y conclusiones.

3.2. El modelo de energía

La ecuación 3.1 define el problema zooming como la minimización de una energía.

$$\min_{u \in U} \parallel \nabla u \parallel_1 + \frac{\lambda}{2} \parallel (Au - g) \parallel_2^2 \tag{3.1}$$

Una diferencia básica respecto al problema de denoising es el uso del operador lineal representado por la matriz A. En el caso general del denoising esta matriz puede considerarse como la matriz identidad. Sin embargo, en el problema del zooming es de vital importancia, pues la matriz A influye directamente en término de datos de la energía para transformar las dimensiones de la imagen ampliada u

en las dimensiones de la imagen de partida g. Consideremos el siguiente ejemplo, en el que la imagen de entrada es:

$$g = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}$$

Y, con un factor de ampliación de 2 (2 de ancho y 2 de alto), la salida es de la forma:

$$u = \begin{pmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{pmatrix}$$

Donde los valores interiores de la matriz hacen referencia a la posición de sus términos en un vector de la matriz organizado por columnas.

La matriz A que transforma dimensiones de u es por tanto de la siguiente forma:

$$A = \begin{pmatrix} X & X & 0 & 0 & X & X & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & X & X & 0 & 0 & X & X & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & 0 & 0 & X & X & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & 0 & 0 & X & X \end{pmatrix}$$

La primera fila hace referencia al primer término del vector g, donde las posiciones 1, 2, 5 y 6 apuntan a la matriz u. Físicamente se puede interpretar como que el píxel 1 de la matriz g se expande hacia los píxeles 1, 2, 5 y 6 de la matriz u. Lo siguiente a analizar es cuánto vale la X. Hay tantas X en una fila como factor de ampliación elevado al cuadrado. Si cada X valiese 1, el producto de Au sería de aproximadamente, factor al cuadrado veces el valor de g. El término de datos valora la variación entre la imagen de salida y la de entrada, por lo que para que sean del mismo orden se requiere que $X = \frac{1}{s^2}$, donde s es el factor de ampliación.

$$\min_{u \in U} \max_{p \in P} \langle p, \nabla u \rangle + \frac{\lambda}{2} \parallel Au - g \parallel_2^2 - \delta_p(p) \tag{3.2}$$

El problema dual será similar al de denoising ROF, pues el único cambio respecto a ese problema está en un término que no depende de p. El procedimiento para derivar el paso de actualización de la variable primal se describe en el Apendice B.4

$$u^{n+1} = (I + \tau\lambda A^T A)^{-1}(u^n + \tau div p^{n+1} + \tau\lambda A^T g) \tag{3.3}$$

La expresión 3.3 requiere de la solución de un sistema lineal de ecuaciones de la forma $Mx = b$ que en muchos casos puede llevar al desarrollo de un algoritmo muy

lento con dificultad en su paralelización mediante GPU. Para evitar este problema se utiliza el método de solución de ecuaciones de Jacobi. [1]. Si se sustituye $\mu = \frac{1}{\tau}$, se obtiene el siguiente resultado:

$$u^{n+1}(\tau I + \lambda A^T A) = \mu u^n + divp^{n+1} + \lambda A^T g \tag{3.4}$$

Para solucionar el sistema de ecuaciones se toma $M = \mu I + \lambda A^T A$ y $b = \mu u^n + divp^{n+1} + \lambda A^T g$. La matriz M de Jacobi se debe descomponer en D y R:

$D = (\mu + \frac{\lambda}{s^4})I$, ya que los elementos de A están divididos por s^2 y los valores de la diagonal de A^A estarán divididos por s^4. Nótese que D es un valor constante.
$R = \lambda A^T A - \frac{\lambda}{s^4} I$
$b = \mu u^n + divp^{n+1} + \lambda A^T g$

Por tanto, el resultado de la actualización primal es el siguiente:

$$u^{n+1} = \frac{\{\lambda A^T A - \frac{\lambda}{s^4} I\} u^n + \mu u^n + divp^{n+1} + \lambda A^T g}{\mu + \frac{\lambda}{s^4}} \tag{3.5}$$

Es importante decir que el método de Jacobi es iterativo, por lo que esta actualización debería realizarse varias veces en cada actualización del primal. Es decir, que el orden de las iteraciones se elevaría al cuadrado. Dada la rápida convergencia del algoritmo primal dual, tan sólo se realizará una iteración de Jacobi.

Se puede observar que la matriz $A^T A$ promedia los píxeles de la imagen grande que hacen referencia a la imagen pequeña mediante el factor $\frac{1}{s^4}$. Ese valor medio es el mismo para todos los píxeles de la imagen grande que tienen en común un mismo píxel de la pequeña. Este resultado permite la implementación sencilla del algoritmo.

Los resultados se encuentran en el Apéndice F.2.

Capítulo 4

Image Deconvolution

4.1. Motivación

La calidad de las imágenes capturadas mediante cámaras digitales dependen del tiempo de exposición del sensor. Si aparecen objetos móviles en la escena o la cámara se encuentra en movimiento, un tiempo de exposición excesivo puede afectar las imágenes ya que en cada instante de tiempo el sensor recibe y promedia diferente información de intensidad. Este fenómeno es también conocido como motion blurring o convolución de movimiento.

A partir de una imagen que ha sufrido un proceso de convolución se puede recuperar la imagen original en unas condiciones muy aceptables. Sólo se necesitan otros dos parámetros además de la imagen: la longitud en píxeles que se ha movido la imagen y la dirección en que ésto ha sucedido. La ecuación que relaciona la velocidad del objeto respecto a la cámara y la velocidad expresada en píxeles, se define como

$$e = \frac{vT}{d} \tag{4.1}$$

Donde v es la velocidad en el plano de la imagen, fácil de obtener a partir de la velocidad real del objeto y la distancia al mismo, T es el tiempo de exposición y d el tamaño de un píxel. La velocidad en el plano de la imagen se puede obtener gracias a una sencilla relación trigonométrica representada en la figura 4.1.

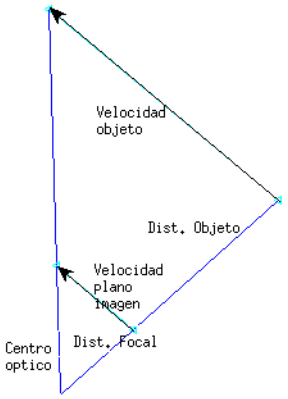


Figura 4.1: Transformación entre el vector de velocidad del objeto y el vector de velocidad en el plano de imagen

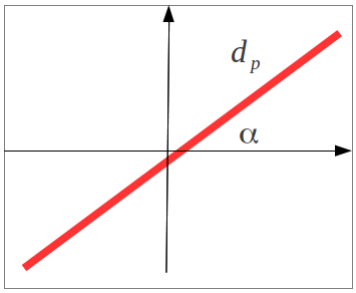
Suponiendo la velocidad como un vector, hay que calcular el vector proporcional en el plano imagen. Denotando v como la velocidad en el plano imagen, V como la velocidad real, d como la distancia focal y D como la distancia desde la lente hasta el objeto en movimiento, se deduce que:

$$v = d\frac{V}{D+d} \tag{4.2}$$

Colocando una cámara de bajas prestaciones en una línea de manufactura, sabiendo a la velocidad y dirección en que se mueve, y la distancia de la cámara a la línea, se puede predecir la convolución que sufrirá la imagen obtenida y se podrá arreglar con gran exactitud.

4.2. Modelo de energía

El caso de la deconvolución es análogo al del zooming, con la salvedad de que la matriz A representa el movimiento en píxeles aplicado a la imagen. Esta matriz se puede modelar mediante un kernel local o matriz de convolución que representa el movimiento lineal de los píxeles. Dicha matriz es fácil de calcular requiriendo sólo como parámetros la longitud en píxeles y su ángulo de inclinación de la recta. La figura 4.2 muestra un ejemplo de una máscara calculada para una longitud de 7 píxeles y un ángulo de 45º de inclinación.



$$A = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0{,}0144942 & 0 \\ 0 & 0 & 0 & 0 & 0{,}0375741 & 0{,}128286 & 0{,}0144944 \\ 0 & 0 & 0 & 0{,}0375741 & 0{,}128286 & 0{,}0375742 & 0 \\ 0 & 0 & 0{,}0375741 & 0{,}128286 & 0{,}0375741 & 0 & 0 \\ 0 & 0{,}0375742 & 0{,}128286 & 0{,}0375741 & 0 & 0 & 0 \\ 0{,}0144944 & 0{,}128286 & 0{,}0375741 & 0 & 0 & 0 & 0 \\ 0 & 0{,}0144942 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Figura 4.2: Cálculo de una máscara de motion blurring. Los valores obtenidos están normalizados, de manera que la suma global de todos los elementos es 1.

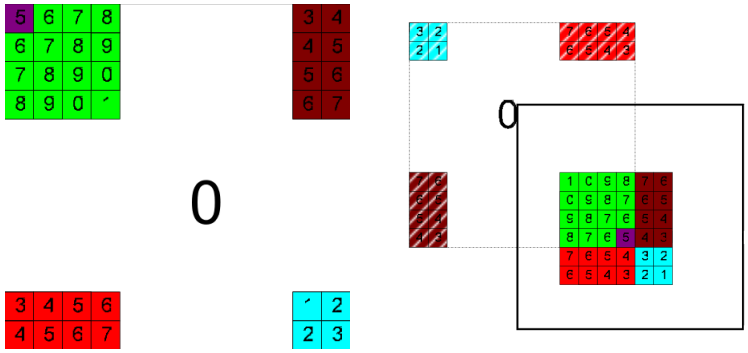


Figura 4.3: Representación del proceso de convolución sobre la imagen de entrada (derecha), dado un operador de motion blurring desplazado circularmente (izquierda).

Para modelar el problema de la deconvolución, se parte de la ecuación 4.3 que describe el modelo de minimización de energía:

$$\min_{u \in U} \| \nabla u \|_1 + \frac{\lambda}{2} \| (Au - g) \|_2^2 \tag{4.3}$$

El paso más importante es la derivación de la actualización de la variable pri-

mal u. Con el fin de evitar el uso de método de Jacobi para resolver el sistema lineal planteado en el problema del zooming, es posible describir las operaciones a partir de la definición de convolución en el dominio frecuencial. Ésto implica el uso del concepto de transformada de Fourier [5]. En nuestra aplicación, la matriz A se sustituye por k, el cual representa el kernel espacial de convolución. Una vez calculada la máscara lineal de movimiento se puede definir el operador lineal k que actuará sobre la imagen, donde el patrón lineal de movimiento aparece desplazado circularmente para realizar una convolución centrada. Al aplicar la transformada de Fourier sobre el kernel y la imagen degradada, la convolución se convierte en un producto de espectros. La figura 4.3 ilustra el patrón de la matriz k y el proceso de convolución sobre la imagen sin degradar en el dominio espacial.

El proceso para la definición de problema primal dual es similar al desarrollado en el caso del zooming. Una vez derivada la función de energía dualizada respecto a la variable primal u, se obtiene el siguiente resultado para su actualización

$$u = \mathcal{F}^{-1}\left(\frac{\mathcal{F}(\hat{u}) + \tau\lambda\mathcal{F}(k)^*\mathcal{F}(g)}{(1 + \tau\lambda\mathcal{F}(k)^2)}\right) \tag{4.4}$$

Donde $*$ es el operador de convolución y $\mathcal{F}$ representa la transformada de Fourier o espectro de una señal y $\mathcal{F}^{-1}$ la respectiva transformada inversa. Para realizar los cálculos píxel a píxel, es necesario que k del tamaño de la imagen para poder realizar la transformada de Fourier y que su producto tenga sentido. Así mismo, se debe hacer uso del algoritmo de Transformada de Fourier Rápida (FFT o Fast Furier Transform). De esta manera, siguiendo el algoritmo de actualización primal dual habitual, se puede realizar la deconvolución eficientemente.

Los resultados obtenidos se encuentran en el Apéndice F.3.

Capítulo 5

Image Inpainting

5.1. Motivación

Durante el proceso de captura de fotografías, es inevitable que en ocasiones aparezcan cuerpos extraños o defectos de la cámara que provoquen que provoquen la pérdida de información. Gracias a métodos de restauración es posible localizar los píxeles defectuosos, a partir de los píxeles correctos, permitiendo la recuperación casi total de la información en la imagen.

Se debe puntualizar que, pese a que este algoritmo que se expone en este capítulo obtiene unos resultados satisfactorios, su implementación requiere del previo conocimiento de la región deteriorada en la imagen. No obstante, es posible emplear algoritmos de detección de regiones como paso previo a la restauración. El resultado dependerá en gran medida de la precisión de dicho algoritmo. Esta sección únicamente se centra en el algoritmo de restauración o *Inpainting*, a partir de información previa de la región deteriorada para la recuperación de imágenes con alto porcentaje de pérdida de información.

5.2. El algoritmo

La formulación del problema general es la siguiente:

$$\min_{u \in U} \| \Phi u \|_1 + \frac{\lambda}{2} \| (u - g) \|_2^2 \tag{5.1}$$

Observando detenidamente la ecuación, se aprecian dos cambios fundamentalmente respecto a los dos casos anteriores:

- La sustitución de ∇ por Φ. En casos específicos el regularizador no tiene por qué afectar a toda la imagen. Más tarde se comentará cuál es su función.

- El segundo término de la ecuación debe ser también modificado, pues mide la variación respecto a la imagen original, y cuando los píxeles deteriorados toman un valor correcto, su variación incrementa.

La ecuación 5.2 puede redefinirse de la siguiente manera:

$$\min_{u \in U} \parallel \Phi u \parallel_1 + \frac{\lambda}{2} \sum_{i,j \in D \setminus I} (u_{i,j} - g_{i,j})^2 \tag{5.2}$$

Donde la I se refiere tan sólo al conjunto de píxeles afectados. Por tanto, el sumatorio de la variación se aplica tan sólo a los píxeles no afectados, obviando así la variación en los afectados.

Dualizando el problema de la misma forma que se ha hecho hasta ahora se obtiene el siguiente resultado:

$$\min_{u \in U} \max_{c \in C} \langle \Phi u, c \rangle + \frac{\lambda}{2} \sum_{i,j \in D \setminus I} (u_{i,j} - g_{i,j})^2 - \delta_c(c) \tag{5.3}$$

Para simplificar el problema dual, se asume $\Phi = \nabla$ y por tanto tiene la misma forma que en apartados anteriores. En el problema primal se debe diferenciar entre los píxeles afectados y los píxeles correctos, de manera que la variable dual se actualiza como:

$$u_{i,j}^{n+1} \begin{cases} u^n + \tau divc & si (i,j) \in I \\ \frac{u^n + \tau divc + \tau \lambda g_{i,j}}{1 + \tau \lambda} & en\ cualquier\ otro\ caso \end{cases} \tag{5.4}$$

Así, el primer caso corresponde a el arreglo de los píxeles corruptos y el segundo término corresponde al desarrollo del término como ya se ha hecho en anteriores apartados. Los resultados obtenidos en el algoritmo se encuentran en el Apéndice F.4

Capítulo 6

Conclusiones

En este proyecto se aborda el problema del procesamiento de imágenes dentro del contexto de la visión por computador para aplicaciones reales. Los problemas de adición de ruido (denoising), de ampliación (zooming), deconvolución o restauración (inpainting) son muy habituales en el trabajo diario con imágenes, especialmente en entornos de fabricación donde las cámaras son usadas como sensores de supervisión.

Debido al tratamiento de información densa de las imágenes, se requiere la formulación de métodos eficientes capaces de contrarrestar en poco tiempo los efectos mencionados. Los métodos continuos variacionales representan una alternativa atractiva no sólo para el modelado de los distintos problemas sino también por la facilidad y robustez que traen consigo en relación a los algoritmos diseñados para encontrar soluciones de alta precisión. El uso de métodos variacionales no sólo cubre el campo del procesamiento de imágenes. Su aplicación también se ha extendido a campos de gran interés como la reconstrucción de entornos en 3D.

A partir de una imagen deteriorada, estos algoritmos contabilizan la variación de cada solución parcial obtenida durante la iteración. En la mayoría de los casos, no se requiere del uso de información previa para la recuperación de la imagen. En este sentido, los algoritmos denoising y zooming trabajan únicamente sobre la imagen dañada, sin necesitar ningún otro dato. No obstante, el modelo empleado puede llegar a ser sensible respecto al parámetro de ruido.

Si bien denoising y zooming no requieren información previa, no sucede lo mismo con los algoritmos de deconvolución y de inpainting los cuales requieren mas información y tratamiento previo. En el caso de la deconvolución se necesita la máscara con la que la imagen ha sido convolucionada. El cálculo de máscaras de convolución conllevan a un camino de investigación interesante para complementar la aplicabilidad de los algoritmos en entornos reales.

Por otro lado, el algoritmo de inpainting recibe como información concretamente el conjunto de píxeles afectados. Esta información sólo se puede aportar en el

laboratorio, a partir de detectores de regiones no siempre disponibles o aplicables en la realidad. Se necesita un algoritmo que reconozca puntos espurios con total precisión para que funcione y, pese a que en los últimos años, se ha avanzado mucho en esta materia, se considera un tema aún en investigación.

A pesar de que es posible mejorar los algoritmos propuestos, se considera un campo de interés el mejorar la forma de obtener los datos que se requieren para que dichos algoritmos puedan funcionar con total autonomía.

Como valoración personal, este proyecto ha representado un enorme reto desde el punto de vista teórico y de desarrollo. El proyecto comprende la programación y desarrollo en lenguajes interpretados como Matlab usado principalmente como herramienta de prototipado y visualización (por ejemplo, para poder obtener las diferentes gráficas y comprender en un inicio el algoritmo); C++, un lenguaje mucho más cercano a la máquina que Pascal o Ada que se utilizaba en la especialidad de Automática y Robótica. Adicionalmente, se trabajó en programación en lenguajes orientados al cálculo masivo paralelo como CUDA. Este último punto requiere la comprensión de las capacidades del hardware diferenciando claramente el trabajo en CPU, GPU y su comunicación a nivel de manejo de memoria.

La dificultad del proyecto también ha residido en la utilización de un sistema operativo de libre distribución, en este caso Ubuntu, menos familiar dentro del ámbito de desarrollo y aplicación de la titulación de Ingeniería Industrial. Así mismo, la memoria se ha escrito en lenguaje Latex, poco habitual dentro de los editores de texto utilizados hasta la fecha.

Este proyecto ha permitido afianzar conceptos de informática dentro del contexto de la especialidad de Automática. De la misma manera, los conceptos matemáticos de análisis numérico y transformadas de Fourier han sido reforzados gracias a su aplicabilidad en el campo del procesamiento de imágenes.

Apéndice A

Gestión del proyecto

La realización del proyecto se ha divido en varias etapas claramente diferenciadas entre sí. Al tratarse de algoritmos cuya implementación es bastante compleja, se comenzó trabajando en el software matemático Matlab, donde se implementó el primer algoritmo: TV-ROF. Se hizo así para asentar los conceptos de Primal y de Dual, cuándo hay que actualizarlos, qué significan los parámetros... Matlab resulta muy cómodo ya que las ecuaciones de este algoritmo contienen una matriz A que multiplica al vector imagen, obteniendo los resultados. Una vez calculada esa matriz, la implementación del algoritmo era relativamente sencilla. El problema surge cuando A está compuesta de una gran cantidad de valores nulos en su interior. Estos valores multiplican también a la imagen obteniendo un valor nulo, como era de esperar. Toda esta cantidad de productos consumen mucho tiempo y no producen nada. Para la fotografía Lena.jpg, que contiene 512x512 píxeles, su ejecución duraba en torno a 45 minutos, para optimizar una sola fotografía. El objetivo es disminuir los tiempos hasta el rango de los milisegundos, se descarta la opción de generar una matriz A y realizar el producto.

Por este motivo, ese mismo algoritmo fue implementado en C++, evitando la generación de la matriz y tratando operar tan sólo con aquellos valores que efectivamente aportan información. Se instaló el sistema operativo Ubuntu (Linux), con los programas Texmaker (para la realización de la memoria), qtcreator (el entorno de programación utilizado) y otros útiles tales como LibreOffice o Matlab.

QtCreator es un entorno de programación que utiliza las bibliotecas Qt. Estas bibliotecas han sido ideadas con el fin de generar interfaces gráficas de usuario. Para trabajar con Qt lo primero que se realiza es un CMakelist.txt, un fichero donde se encuentra la información necesaria para ejecutar una futura compilación evitando la adición de librerías en todos los ficheros o los linkados manualmente.

Así, para solucionar el problema se ejecutaba un bucle "for"que recorría toda la imagen y realizaba las operaciones pertinentes. Se logró reducir así el tiempo de ejecución hasta aproximadamente 10 segundos. De aquí surge la necesidad de

implementar el algoritmo en la tarjeta gráfica. Como cada thread de la tarjeta es capaz de realizar una operación sencilla, se puede sustituir ese bucle "for"por una paralelización completa, y así optimizar el tiempo.

La solución es programar en CUDA, un conjunto de herramientas desarrollado por nVidia para codificar algoritmos en GPU nVidia. Como no se disponía de un ordenador personal con GPU nVidia, se solicitó una cuenta en el servidor "Hermes"de unizar donde se ejecutarían los algoritmos. Así, se procedió a la realización de los diferentes algoritmos en Hermes. Se implementaron TV-ROF, Huber-ROF, TV-L1 y Zooming. Una vez que se quería realizar el algoritmo de Image deconvolution, era necesaria una librería que calculase la FFT (Fast Fourier Transform). Al no encontrarse instalada en el servidor y, sabiendo que éste iba a estar apagado durante el mes de Agosto, se generó una cuenta en el ordenador de la universidad de mi tutora, Lina María Paz, con el fin de que pudiese seguir trabajando ahí.

Cuando se implantaron los diferentes algoritmos, surgió la duda de cuáles eran los parámetros óptimos que se debían colocar en cada uno de dichos algoritmos. Una vez entendido el significado de cada parámetro y qué optimiza, la manera de actuar era relativamente sencilla: recorrer un rango de valores con un paso relativamente pequeño y mostrar por pantalla el resultado de cada caso (ratio de señal/ruido o iteraciones hasta convergencia). Este hecho provocaba la necesidad de realizar un nuevo programa, de forma que para ordenar todos los algoritmos, se decidió realizar una interfaz de usuario donde se podía seleccionar el tratamiento al que se quería someter a la imagen y poder observar la imagen modificada, la arreglada, cómo evoluciona el GAP o la función de energía y los la evolución de los diferentes parámetros.

Apéndice B

Deducciones matemáticas

B.1. TV-ROF

B.1.1. Cálculo de $\partial_p E(u,p)$

$$\partial_p E(u,p) = \partial_p(\langle p, \nabla u\rangle + \frac{\lambda}{2} \parallel u - g \parallel_2^2 - \delta_p(p)) \tag{B.1}$$

Se simplifica término a término:

$\partial_p(\langle p, \bigtriangledown u\rangle) = \bigtriangledown u$, ya que se trata de un producto escalar.

$\partial_p(\frac{\lambda}{2} \parallel u - g \parallel_2^2) = 0$, ya que no depende de p.

$\partial_p(\delta_p(p)) = 0$, Se coloca como 0 para permitir su cálculo. Sin embargo, este término impone la restricción de que la variable dual en el resultado final no sea mayor que 1.

Por tanto se concluye que:

$$\partial_p E(u,p) = \nabla u \tag{B.2}$$

B.1.2. Cálculo de $\partial_u E(u,p)$

$$\partial_u E(u,p) = \partial_u(\langle p, \nabla u\rangle + \frac{\lambda}{2} \parallel u - g \parallel_2^2 - \delta_p(p)) \tag{B.3}$$

Se simplifica término a término

$\partial_u(\langle p, \bigtriangledown u\rangle) = \partial_u(-\langle u, divp\rangle) = -divp$. En el primer paso se convoluciona el producto de vector por gradiente en un producto negado de vector por divergencia. Se trata de una propiedad matemática aplicable al operador ∇. Una vez que es un producto derivable, se obtiene el resultado.

$\partial_u(\frac{\lambda}{2} \parallel u - g \parallel_2^2) = \lambda(u - g)$, al ser una norma cuadrática simple.

$\partial_u(\delta_p(p)) = 0$, ya que no depende de u.
Por tanto se concluye que

$$\partial_u E(u,p) = -divp + \lambda(u-g) \tag{B.4}$$

B.2. HUBER-ROF

B.2.1. Cálculo de $\partial_p E(u,p)$

$$\partial_p E(u,p) = \partial_p(\langle p, \nabla u\rangle - \delta_p(p) - \frac{\alpha}{2} \parallel p \parallel^2 + \frac{\lambda}{2} \parallel u-g \parallel_2^2) \tag{B.5}$$

$\partial_p(\langle p, \nabla u\rangle) = \nabla u$, ya que se trata de un producto escalar.
$\partial_p(\delta_p(p)) = 0$, Es el mismo caso que el anterior. Se coloca como 0 pero su efecto se añadirá al final del desarrollo.
$\partial_p(\frac{\alpha}{2} \parallel p \parallel^2) = \alpha p$ Una derivada común de un valor al cuadrado.
$\partial_p(\frac{\lambda}{2} \parallel u-g \parallel_2^2) = 0$, ya que no depende de p.
Por tanto, obtenemos como resultado

$$\partial_p E(u,p) = \bigtriangledown u - \alpha p \tag{B.6}$$

B.3. TV-L1

B.3.1. Cálculo de $\partial_u E(u,p)$

$$\partial_u E(u,p) = \partial_u(\langle p, \nabla u\rangle - \delta_p(p) + \lambda \parallel u-g \parallel_1) \tag{B.7}$$

$\partial_u(\langle p, \nabla u\rangle) = \partial_u(-\langle u, divp\rangle) = -divp$,
$\partial_u(\delta_p(p)) = 0$
$\partial_u(\frac{\lambda}{2} \parallel u-g \parallel_2^2) = \lambda(u-g)$

$$\partial_u E(u,p) = -divp + \partial_u\lambda \parallel u-g \parallel_1 \tag{B.8}$$

B.4. Derivación de la energía para el problema del zooming

Partiendo del modelo de energía, se desea calcular su derivada respecto a la variable primal:

$$\partial_u E(u,p) = \partial_u(\langle p, \nabla u\rangle + \frac{\lambda}{2} \parallel Au - g \parallel_2^2 - \delta_p(p)) \tag{B.9}$$

En primera instancia, se obtiene que:
$\partial_u(\langle p, \nabla u\rangle) = \partial_u(-\langle u, divp\rangle) = -divp$
$\partial_u(\delta_p(p)) = 0$
$\partial_u(\frac{\lambda}{2} \parallel Au - g \parallel_2^2))$, requiere un desarrollo más amplio:

$$\partial_u \parallel Au - g \parallel_2^2 = \partial_u(Au-g)^T(Au-g)$$
$$(Au-g)^T(Au-g) = ((Au)^T - g^T(Au-g))$$
$$((Au)^T - g^T(Au-g)) = (u^TA^T - g^T)(Au-g)$$
$$(u^TA^T - g^T)(Au-g) = u^TA^TAu - u^TA^Tg - g^TAu + g^Tg$$

se realiza el cambio $B = A^TA$ y se calcula así el primer término:

$$\partial_u(u^TA^TAu) = \partial_u u^TBu$$
$$\partial_u(u^TBu) = (B + B^T)u$$

Se invierte el cambio:

$$\partial_u(u^TA^TAu) = (A^TA + (A^TA)^T u$$
$$\partial_u(u^TA^TAu) = (A^TA + A^TA)u$$
$$\partial_u(u^TA^TAu) = 2A^TAu$$

Se calcula el resto de términos:

$$\partial_u(-u^TA^Tg - g^TAu) = \partial_u(-V^Tg - g^TV) = \partial_u(-2g^TV)$$
$$\partial_u(-2g^TV) = \partial_u(-2g^TAu) = 2A^Tg$$
$$g^Tg = 0$$

Al final por tanto:

$$\frac{\lambda}{2} \parallel Au - g \parallel_2^2) = \lambda(A^TAu - A^Tg)$$

$$\partial_u E(u,p) = -divp + \lambda(A^TAu - A^Tg) \tag{B.10}$$

Se sustituye la derivada por su definición:

$$\frac{u^n - u^{n+1}}{\tau} = -divp + \lambda(A^TAu^{n+1} - A^Tg) \tag{B.11}$$

$$(u^n - u^{n+1}) = -\tau divp^{n+1} + \tau\lambda(A^TAu^{n+1} - A^Tg) \tag{B.12}$$

Finalmente, la actualización puede derivarse de la siguiente expresión:

$$u^{n+1}(I + \tau\lambda A^TA) = u^n + \tau divp^{n+1} + \tau\lambda A^Tg \tag{B.13}$$

B.5. Derivación de la función de energía para el problema de la deconvolución

Partiendo de la definición de función de energía, y considerando que la $\partial_u E(u,p) = 0$, se obtiene que:

$$\partial_u E(u,p) = \partial_u(-\langle p, \nabla u\rangle + \frac{\lambda}{2} \parallel Au - g \parallel_2^2 - \delta_p(p)) \tag{B.14}$$

Sustituyendo términos por su valor:

$$\partial_u E(u,p) = divp + \lambda A^T(Au - g) \tag{B.15}$$

Si se coloca $\hat{u} = u - \tau divp$ en el algoritmo, se puede escribir que $divp = \frac{u-\hat{u}}{\tau}$

$$\partial_u E(u,p) = \frac{u-\hat{u}}{\tau} + \lambda A^T(Au - g) = 0 \tag{B.16}$$

Se realiza el cambio de la matriz A que como ya se ha comentado es muy complicada de implementar y se sustituye por $k*$, que es la convolución. Tiene el significado físico de pasar la máscara por la imagen.

$$\frac{u-\hat{u}}{\tau} + \lambda k^* * (k * u - g) = 0 \tag{B.17}$$

Donde $k*$ es la convolución y k^** es el conjugado que convoluciona, es la equivalencia a la A^T anterior. En este momento se aplica la Transformada de Fourier a la izquierda y derecha de la ecuación.

$$\frac{\mathcal{F}(u) - \mathcal{F}(\hat{u})}{\tau} + \lambda \mathcal{F}(k)^*(\mathcal{F}(k)\mathcal{F}(u) - \mathcal{F}(g)) = \mathcal{F}(0) \tag{B.18}$$

Al aplicar Fourier, las convoluciones se convierten en productos. Se sigue operando hasta dejar a un lado de la ecuación $\mathcal{F}(u)$:

$$\mathcal{F}(u) - \mathcal{F}(\hat{u}) + \tau\lambda\mathcal{F}(k)^2\mathcal{F}(u) - \tau\lambda\mathcal{F}(k)^*\mathcal{F}(g)) = 0 \tag{B.19}$$

$$\mathcal{F}(u)(1 + \tau\lambda\mathcal{F}(k)^2) = \mathcal{F}(\hat{u}) + \tau\lambda\mathcal{F}(k)^*\mathcal{F}(g) \tag{B.20}$$

$$\mathcal{F}(u) = \frac{\mathcal{F}(\hat{u}) + \tau\lambda\mathcal{F}(k)^*\mathcal{F}(g)}{(1 + \tau\lambda\mathcal{F}(k)^2)} \tag{B.21}$$

Se realiza la transformada inversa y así se obtiene la sución final:

$$u = \mathcal{F}^{-1}\left(\frac{\mathcal{F}(\hat{u}) + \tau\lambda\mathcal{F}(k)^*\mathcal{F}(g)}{(1 + \tau\lambda\mathcal{F}(k)^2)}\right) \tag{B.22}$$

Es necesaria una k del tamaño de la imagen para poder realizar la transformada de Fourier y que su producto tenga sentido. De esta manera, siguiendo el algoritmo de actualización primal dual habitual, y mediante el cálculo de la FFT en CUDA, se puede realizar la deconvolución de una forma muy rápida.

Apéndice C

Tipos de ruido

C.1. El ruido Gaussiano

El ruido Gaussiano asigna a cada píxel de la imagen una función gaussiana centrada en el valor de dicho píxel.

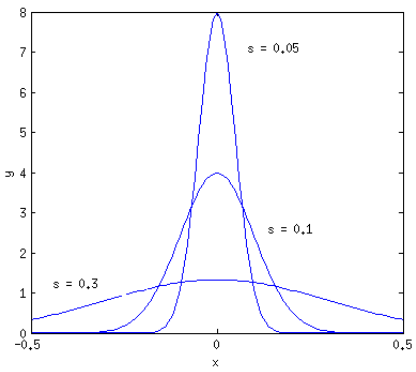


Figura C.1: Comparación de la función gaussiana para diferentes valores de σ

De esta forma, para valores de σ pequeños, el valor del píxel cambiará muy poco el la mayoría de los casos. Para valores de σ grandes, los píxeles cambiarán mucho su valor en media. El código que crea el ruido gaussiano es el siguiente:

```
1 Mat addNoise(Mat &I, float mean, float std)
  {
3     Mat Inoisy;

5     Mat Inoise(I.rows, I.cols, CV_32FC1);
      randn(Inoise, mean, std);
7
```

```
    Inoisy = I+Inoise;
    cv::max(Inoisy, 0.0, Inoisy);
    cv::min(Inoisy, 1.0, Inoisy);


    return Inoisy;
}
```

La función randn genera un valor aleatorio de media *mean* y de σ *std*. Se impone *mean* igual a cero, de forma que la matriz Inoise creada es una función centrada en cero y de la desviación estándar asignada. Luego se suma Inoise a la matriz original para crear el ruido, acotando los valores máximos y mínimos que pueda tener.

C.2. El ruido de sal y pimienta

Recibe este nombre por su similitud a la sal y la pimienta al colocar los píxeles de color blanco y negro. Es una forma de generar espurios y comprobar lo robusto del algoritmo ante su aparición. El parámetro que recibe como entrada es el porcentaje relativo de píxeles espurios que se desea aparezcan en la imagen. Se supone misma cantidad de píxeles generados blancos que negros. El código para generarlo es el siguiente:

```
Mat Add_sal_y_pimienta(Mat original, float porcentaje)
{
    int cols = original.cols;
    int rows = original.rows;
    bool rotar = false;
    cv::Mat I = original;
    for (int i = 0; i<cols; i++)
        for(int j=0; j<rows; j++)
        {
            int valor = rand()%100;
            if ((float)valor <= porcentaje)
            {
                if (rotar == false)
                {
                    I.at<float>(i,j) = 0;
                    rotar = true;
                }
                else
                {
                    I.at<float>(i,j) = 1;
                    rotar = false;
```

```
22                      }
                    }
24              }
        return I;
26  }
```

Para cada píxel se genera un valor aleatorio que al hacer %100 se calcula el resto de ese valor con 100. Así, el número resultante es un valor entre 00 y 99. Si el número aleatorio generado es inferior al porcentaje, significa que ese píxel va a ser un espurio. La variable booleana *rotar* es quien se encarga de decidir si el espurio será blanco o negro, cambiando su valor para el siguiente espurio.

Apéndice D

Software

Hay varios apartados interesantes dentro del entorno Qtcreator que deben ser explicados.

D.1. El CMakelist.txt

Como ya se ha mencionado, el CMakelist.txt es un archivo donde se coloca información previa a la compilación con el fin de simplificar los archivos de código, evitando la inclusión de rutas de librerías o procedimientos de linkado en dichos archivos.

Se comienza colocando la versión del CMake, que es quien compila el programa de Qtcreator. También se coloca el título del proyecto.

```
CMAKE_MINIMUM_REQUIRED( VERSION 2.8.0 )
2
PROJECT( visualization )
```

A continuación se deben colocar las distintas fuentes del programa. Los archivos .cpp contienen funciones que se ejecutan en CPU. Los archivos .h contienen la cabecera de esas funciones. Los archivos .cu contienen funciones ejecutadas en GPU y las funciones que, ejecutándose en CPU, llaman las que se ejecutan en GPU. Como en el caso anterior, los archivos .cuh contienen las declaraciones de dichas funciones. Finalmente, los archivos .ui son aquellos que contienen la información de la interfaz. Estos archivos los genera automáticamente Qtcreator a través de un entorno gráfico muy sencillo de utilizar donde el usuario puede colocar botones, widgets y otras útiles con mucha facilidad. En el CMakelist.txt se añade de la siguiente forma (ejemplo de interfaz individual de denoising):

```
SET(SOURCES main.cpp denoising.cu denoising.cpp viewerwidget.cpp)
SET(HEADERS denoising.h denoising.cuh viewerwidget.h)
SET(FORMS denoising.ui)
```

Siempre debe haber un archivo llamado main.cpp, que es el principal. Él es quien se encarga de llamar al resto de archivos. A continuación se deben añadir los paquetes necesarios en la aplicación. En caso de que no se encuentren esos paquetes, se debe añadir su ruta manualmente.

```
FIND_PACKAGE( CUDA REQUIRED )
FIND_PACKAGE( Qt4 REQUIRED )
FIND_PACKAGE(GLEW REQUIRED)
FIND_PACKAGE( OpenGL REQUIRED )
FIND_PACKAGE( OpenCV REQUIRED )
```

Para hacer una precompilación, se añade el siguiente código, donde se generan los .moc y .ui, no ejecutables con un editor de texto y que permiten la generación de la interfaz. Las dos últimas líneas especifican dónde se colocará el ejecutable.

```
QT4_WRAP_CPP(HEADERS_MOC ${HEADERS})
QT4_WRAP_UI(FORMS_HEADERS ${FORMS})
QT4_ADD_RESOURCES(RESOURCES_RCC ${RESRC})

INCLUDE_DIRECTORIES(${CMAKE_CURRENT_BINARY_DIR})
INCLUDE_DIRECTORIES(${CMAKE_CURRENT_SOURCE_DIR})
```

Finalmente, se añade la información de compilación. El apartado ADD DEFINITIONS es el que se encarga de escribir la orden de compilación, aquella que el usuario debería escribir por consola en caso de hacerlo manualmente. Las últimas dos líneas generan el ejecutable a partir de los archivos fuente y las librerías necesarias.

```
ADD_DEFINITIONS( -O2 -march=core2 -msse3 -Wall )

CUDA_ADD_EXECUTABLE(${CMAKE_PROJECT_NAME} ${SOURCES} ${HEADERS_MOC} ${FORMS_HEADERS} ${RESOURCES_RCC})
TARGET_LINK_LIBRARIES(${CMAKE_PROJECT_NAME} ${SDK_LIBS} ${OpenCV_LIBS} ${OPENGL_LIBRARIES} ${GLEW_LIBRARIES} ${QT_LIBRARIES} ${QWT_LIBS})
```

D.2. Archivos de interfaz

Una interfaz tiene siempre asignados tres archivos, siguiendo con el ejemplo anterior, serían: denoising.cpp, denoising.h y denoising.ui. El archivo .ui tiene poco interés porque contiene el diseño gráfico. Se comienza por la explicación del archivo .h.

Dentro del archivo .h se debe comenzar por definir una clase, que contendrá todas las funciones y variables de esa interfaz. La definición de la clase tiene la siguiente estructura:

```
namespace Ui {
class Denoising;
}

class Denoising : public QMainWindow
{
    Q_OBJECT

public:
    explicit Denoising(QWidget *parent = 0);
    ~Denoising();

private slots:
  ...

private:
  ...

};
```

Los slots son las funciones que tienen que ver con la interfaz, como podrían ser *botón pulsado* o *editado de variable finalizado*. En el apartado de private slots es en el que se declaran. En el apartado private, se incluyen todas las variables y funciones privadas que no son slots, como podría ser la función que genera ruido. En el archivo .cpp es donde se implementan todos los slots.

D.3. CUDA

CUDA [4] trabaja en GPU, ejecutando los algoritmos más rápido al poder paralelizarlos. Sin embargo, la información que está en CPU hay que enviársela a GPU y ésta a su vez devolverla a GPU, algo que cuesta tiempo. Si este procedimiento se realizase en cada iteración para cada variable, el ahorro en tiempo no sería muy grande. Por este motivo surgen las texturas. Las variables en GPU pueden ser asignadas a texturas donde son guardadas y no deben ser enviadas de CPU para ser actualizadas, acelerando el proceso. Al principio del programa se deben declarar las texturas y luego asignarles las variables que se introducirán en ellas.

```
texture<float, 2> tex_u;
texture<float, 2> tex_g;
texture<float, 2> tex_u_hat;
texture<float2, 2> tex_p;
texture<float, 2> tex_u_comp;

  ...

cudaBindTexture2D( NULL, tex_u    , u_dev    , chd_float, cols, rows,
    BYTES_PER_ROW);
cudaBindTexture2D( NULL, tex_g    , g_dev    , chd_float, cols, rows,
    BYTES_PER_ROW);
cudaBindTexture2D( NULL, tex_u_hat    , u_hat_dev    , chd_float,
    cols, rows, BYTES_PER_ROW);
cudaBindTexture2D( NULL, tex_p    , p_dev    , chd_float2, cols, rows,
    BYTES_PER_ROW2);

  (en el .cuh:)

static const cudaChannelFormatDesc chd_float = cudaCreateChannelDesc<
    float>();
static const cudaChannelFormatDesc chd_float2 = cudaCreateChannelDesc
    <float2>();
```

En toda declaración y asignación de textura debe especificarse el tipo de dato que se va a guardar en ella y la dimensión, en este caso dos: ancho y alto de la imagen.

A toda función que se ejecuta en GPU deben aportársele dos parámetros concretos: la cantidad de bloques y la cantidad de threads por bloque. Los threads son los pequeños componentes de la tarjeta gráfica que ejecutan las operaciones sencillas que se les envían con CUDA. Los bloques son una agrupación ficticia realiza el programador para tratar de optimizar el código. Ambos valores deben ser enteros, por lo que una vez definidos los threads por bloque, la cantidad de

bloques se calcula dividiendo filas y columnas por los threads por bloque. Para asegurar que se recorre toda la imagen, se hace una división redondeando hacia arriba.

```
#define NTHREADS_BLOCK 16

dim3 nThreads(NTHREADS_BLOCK, NTHREADS_BLOCK);
dim3 nBlocks(divUp(cols, nThreads.x),divUp(rows, nThreads.y));
```

Para mejorar la organización y claridad del código, a todas las funciones ejecutadas en GPU se les nombra con el prefijo kernel de forma que así es más sencillo entender qué se está ejecutando y dónde. Un ejemplo de algoritmo primal-dual es el siguiente:

```
for (int iter = 0; iter < maxIter; ++iter)
        {
        kernel_update_dual_TV_ROF<<<nBlocks, nThreads>>>(p_dev,
    u_hat_dev, cols, rows, sigma, gx_dev, gy_dev);
        cudaThreadSynchronize();
        theta = 1.0f/sqrtf(1.0f+2*gamma*tau);
        kernel_update_Primal_TV_ROF<<<nBlocks, nThreads>>>(u_dev,
    u_hat_dev, g_dev, p_dev, tau, lambda, theta, cols, rows, div_p_dev
    );
        cudaThreadSynchronize();

        tau = theta*tau;
        sigma = sigma/theta;
     }
```

Las variables con el sufijo dev son todas aquellas que están en la GPU y las host son aquellas que reciben los datos de dev en la CPU. Así, se recorre un bucle "for"durante las iteraciones que se desee, actualizando los valores u dev (primal) y p dev (dual) en cada una de las iteraciones.

Apéndice E

Programación

La estructura básica de el archivo donde se ejecutan los algoritmos matemáticos deducidos en los apartados 2,3,4 y 5 sigue una secuencia como la expuesta en el apartado anterior. En este apartado se explica cómo se realizan las actualizaciones del Primal y del Dual. Se comienza con un ejemplo:

```
__global__ void kernel_update_dual(float2 *p_dev, float *u_hat_dev,
    int cols, int rows, float sigma)
{
    int x = blockDim.x*blockIdx.x + threadIdx.x;
    int y = blockDim.y*blockIdx.y + threadIdx.y;

    if(x < cols && y < rows)
    {
        float tx = x + 0.5f;
        float ty = y + 0.5f;
        int offset = y*cols+x;

        float gx = tex2D(tex_u_hat, tx+1,ty)-tex2D(tex_u_hat, tx,ty);
        float gy = tex2D(tex_u_hat, tx,ty+1)-tex2D(tex_u_hat, tx,ty);

        if (x == (cols-1))
        {
            gx = 0.0f;
        }

        if (y == (rows-1))
        {
            gy = 0.0f;
        }

        float2 p = tex2D(tex_p,tx,ty);
        p.x = p.x + sigma*gx;
```

```
27          p.y = p.y + sigma*gy;

29          float norma = sqrtf(p.x*p.x+p.y*p.y);
            float den = fmaxf(1.0f, norma);
31          p.x = p.x/den;
            p.y = p.y/den;
33          p_dev[offset] = p;

35      }
    }
```

El primer paso es la definición de los valores x e y en función del bloque y el Thread concreto de ese bloque en el que se vaya a ejecutar el código. Esta función actualiza el dual, quien, según el algoritmo expuesto, requiere del gradiente de $\hat{u}$, que ha sido guardado en textura. Como la textura había sido definida previamente con dos dimensiones, es muy sencillo tomar los valores del gradiente ya que se hace como se haría en una matriz de píxeles: El píxel de la derecha menos el píxel en el que se ejecuta la función es el gradiente en x y el píxel de abajo menos el píxel actual es el gradiente en y. p también se encuentra en textura, lo que hace más rápido el algoritmo de actualización. Una vez calculada la norma, la $p.x$ y la $p.y$, se debe enviar mediante la instrucción $p_{dev}[offset] = p$; a la variable p_{dev}, para que su nuevo valor se almacene en textura y en la siguiente iteración se pueda proceder de la misma manera.

```
__global__ void kernel_update_Primal(float *u_dev, float *u_hat_dev,
    float *g_dev, float2 *p_dev, float tau, float lambda, float theta,
     int cols, int rows)
2
{
4
    int x = blockDim.x*blockIdx.x + threadIdx.x;
6   int y = blockDim.y*blockIdx.y + threadIdx.y;

8   float uhat;
    float u;
10
    if(x < cols && y < rows)
12  {
        float tx = x + 0.5f;
14      float ty = y + 0.5f;
        int offset = y*cols+x;
16
        float kijx = 0, kijy = 0, ki_1jx = 0, kij_1y = 0;
18      float2 py_ij_1 = tex2D(tex_p,tx,ty-1);
```

```
        float2 p_ij = tex2D(tex_p,tx,ty);
20      float2 px_i_1j = tex2D(tex_p,tx-1,ty);

22      if (y > 0)
        {
24          kij_1y = py_ij_1.y;
        }
26
        if (y < rows-1)
28      {
            kijy = p_ij.y;
30      }

32      if (x > 0)
        {
34          ki_1jx = px_i_1j.x;
        }
36
        if (x < cols-1)
38      {
            kijx = p_ij.x;
40      }

42      float u_old = tex2D(tex_u, tx,ty);
        float g = tex2D(tex_g, tx,ty);
44      float dip = kijx-ki_1jx+kijy-kij_1y;
        uhat = u_old +tau*dip;
46      u = (uhat+tau*lambda*g)/(1+tau*lambda);
        uhat = u+theta*(u-u_old);
48      u_dev[offset] = u;
        u_hat_dev[offset] = uhat;
50  }
}
```

La manera de actualizar el Primal cambia sustancialmente en función del método. Este ejemplo corresponde al TV-ROF. El primer paso común a todos los métodos es la realización de la divergencia de p. Para ello se definen las variables $kijx$, $kijy$, ki_1jx y kij_1y, que se refieren al valor de p en el píxel actual en $p.x$ y en $p.y$, al píxel de la derecha en el caso de $p.x$ y al píxel de abajo en el caso de $p.y$. La divergencia se calcula como se muestra en el código. A continuación se debe guardar el valor de u_{old} para la futura actualización de $\hat{u}$. A partir de ahí, se debe seguir el algoritmo. Finalmente, tanto $\hat{u}$ como u deben enviarse a la variable que está en textura para que la actualización sea efectiva.

Apéndice F

Resultados

Se exponen a continuación los resultados y conclusiones obtenidas a partir de los diferentes algoritmos.

F.1. Resultados Denoising

F.1.1. Evaluación del modelo TV-ROF

El primer experimento se realiza sobre una imagen a la que se le ha adicionado ruido gaussiano con desviación estándar σ_r. El resultado de la figura F.1 se ha obtenido al ejecutar el algoritmo TV-ROF.

(a) Imagen original (b) Imagen ruidosa $\sigma_r = 0{,}1$ (c) Imagen resultado $\lambda = 8{,}0$ TV-ROF

Figura F.1: Primer experimento del modelo TV-ROF

Como es de esperar, el uso de la norma del gradiente no penaliza las discontinuidades en la imagen pudiéndose mantener los bordes. Así mismo, las regiones con intensidades cercanas se suavizan. Durante la ejecución del algoritmo se ha

llevado a cabo un análisis de verificación del GAP definido para cada modelo. La figura F.2 muestra el resultado para un numero fijo de iteraciones. Se puede notar que el GAP tiende a cero con apenas 30 iteraciones.

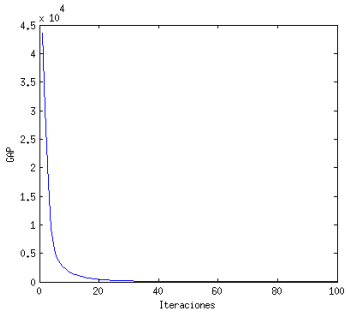
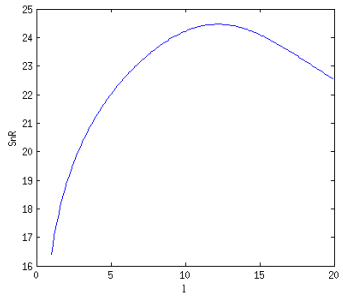


Figura F.2: Evolución del GAP respecto a las iteraciones

Figura F.3: Imagen ruidosa $\sigma = 0{,}3$

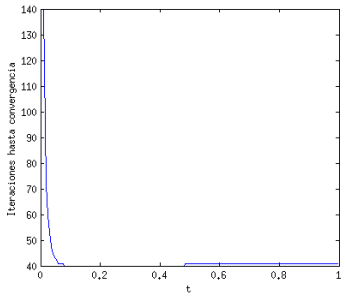
Figura F.4: Imagen resultado $\lambda = 8{,}0$ TV-ROF

Se puede observar que al aumentar drásticamente el porcentaje de ruido en la imagen el algoritmo no puede recuperar totalmente la imagen original. No obstante, ésto ocurrirá para cualquier algoritmo de filtrado. Los parámetros óptimos del método se pueden obtener mediante una simulación. Así, se obtienen las gráficas:



(a)

Dependencia SnR respecto a λ en TV-ROF



(b)

Dependencia iteraciones hasta convergencia respecto a τ en TV-ROF

Figura F.5: Evaluación de parámetros λ y τ

De acuerdo a los resultados, el valor óptimo para λ sería de 12 tal como se observa en F.5 a) y un valor óptimo de τ sería de 0.25, de acuerdo a la gráfica F.5 b).

Los valores óptimos también dependerán de la cantidad de ruido introducido, siendo λ especialmente sensible a esta variación. La figura F.4 se ha calculado para $\sigma_r = 0.3$. Si se realiza la evaluación de SnR de los parámetros para este ruido, el valor de λ varía como se muestra en la figura .

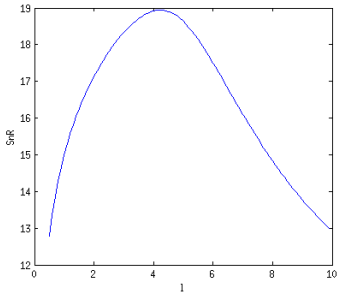


Figura F.6: dependencia SnR respecto a λ en TV-ROF para un ruido de $\sigma = 0.3$.

El óptimo de λ ha variado a 4.2. Con ese nuevo valor se obtiene una resultado distinto. La figura F.7 muestra una comparación entre el resultado previamente obtenido con $\lambda = 8$ y el valor óptimo:

(a) Imagen resultado $\lambda = 8{,}0$ TV-ROF

(b) Imagen resultado $\lambda = 4{,}2$ TV-ROF

Figura F.7: Comparación de resultados cuando se incrementa el ruido en la imagen y se aplican valores diferentes de λ.

Cuanto menor es el error, mayor es la fiabilidad de la imagen sobre la que el algoritmo actúa, por lo que el resultado será muy similar a dicha imagen de forma que λ será muy elevada. Cuanto mayor es el error, más importante será el término de gradiente, así que el λ óptimo deberá ser menor.

F.1.2. Evaluación del modelo Huber-ROF

En el caso del modelo de Huber-ROF, observar que el algoritmo primal dual converge mucho más rápido a la solución en comparación con el modelo TV-ROF. La figura F.8 ilustra el resultado obtenido. Las figuras F.9 y F.10 representan cómo varía el GAP y la relación SnR respecto a un parámetro. Se puede concluir por tanto que este parámetro siempre converge en las mismas iteraciones. El valor óptimo de λ es de 7.5 y el de α es de 0.025.

(a) Imagen ruidosa $\sigma_r = 0{,}1$

(b) Imagen resultado $\lambda = 5{,}0$ Huber-ROF

Figura F.8: Resultado por el algoritmo primal dual para el modelo Huber-ROF.

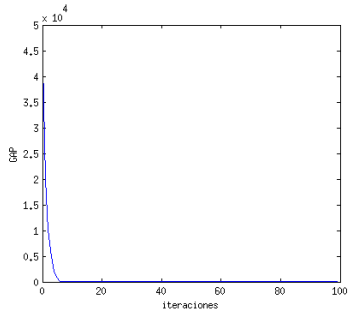
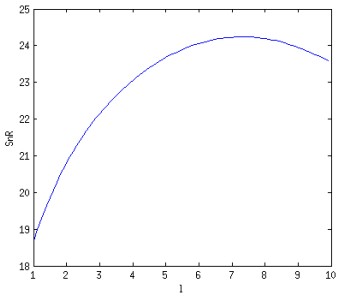
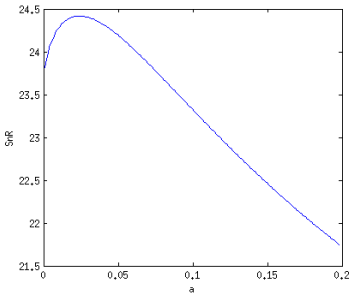


Figura F.9: Evolución del GAP por iteración para el modelo Huber-ROF.



(a) Dependencia SnR respecto a λ



(b) Dependencia SnR respecto a α

Figura F.10: Evaluación de parámetros para el modelo Huber-ROF

F.1.3. Evaluación del modelo TV-L1

El algoritmo TV-L1 contiene una diferencia esencial respecto a los dos anteriores: La dificultad del cálculo del GAP. A diferencia de los otros modelos, se ha calculado el valor de la función de energía. Dicha función de energía debe alcanzar un valor mínimo una vez el algoritmo haya convergido a la solución óptima. La figura F.11 muestra la imagen de resultado obtenida mientras la figura F.12 muestra la evolución de la energía para el modelo TV-L1.

(a) Imagen ruidosa $\sigma_r = 0{,}1$

(b) Imagen resultado $\lambda = 1{,}5$ TV-L1

Figura F.11: Resultado obtenido para el modelo TV-L1

Se comprueba que efectivamente su función de energía disminuye hasta alcanzar un mínimo.

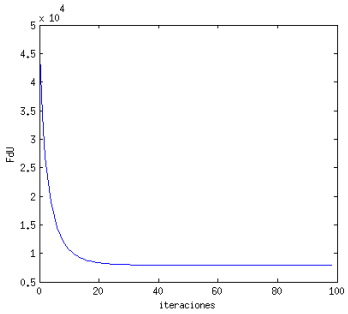
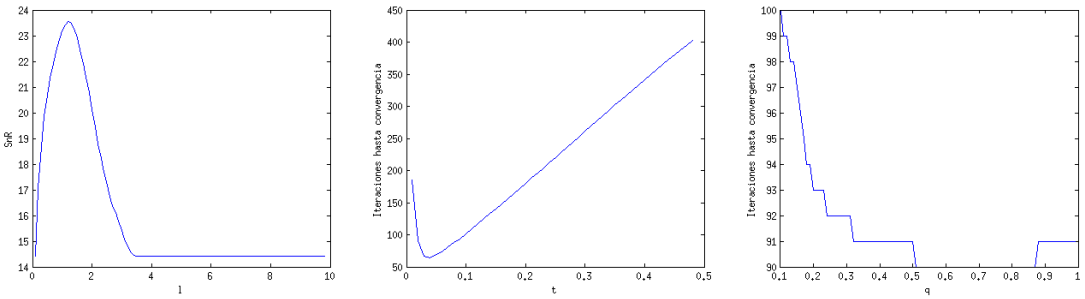


Figura F.12: Evolución de la función de energía por iteración para el modelo TV-L1.

El método TV-L1 es el que más grados de libertad tiene de los tres implementados. Por tanto, se realizará un análisis de λ, τ y θ representados gráficamente en la figura F.13.



(a) Dependencia SnR respecto a λ

(b) Dependencia iteraciones hasta convergencia respecto a τ .

(c) Dependencia iteraciones hasta convergencia respecto a θ.

Figura F.13: Evaluación de parámetros para el modelo TV-L1.

Observando los resultados se puede concluir que el valor óptimo de λ es de 1,25 aproximadamente y el de τ es de 0.025. θ afecta en menor medida que los otros parámetros pudiéndose asignar un valor de 0.5.

F.1.4. Comparación de modelos

La figura F.14 muestra los resultados visuales para los tres modelos. Así mismo, la tabla F.1 muestra el tiempo total de ejecución para cada solución.

(a) TV-ROF

(b) Huber

(b) TV-L1

Figura F.14: Comparación de modelos. Resultados obtenidos al aplicar el conjunto de parámetros óptimos en cada caso.

Para una misma cantidad de ruido ($\sigma = 0{,}1$), el resultado es muy similar. A simple vista, se observa que la imagen resultado del TV-L1 elimina algo peor el ruido en comparación a los modelos TV-ROF y HUBER-ROF. Entre el modelo TV-ROF y el HUBER-ROF se encuentran pocas diferencias apreciables.

Modelo	Tiempo 200 iteraciones	Tiempo hasta convergencia
TV-ROF	30 ms	5ms
HUBER-ROF	37 ms	1ms
TV-L1	28 ms	5ms

Tabla F.1: Tiempos de ejecución para los modelos evaluados

En todos los métodos se han aplicado 200 iteraciones para asegurar convergencia. Sin embargo, no todos los métodos convergen con las mismas. En el mejor de los casos, bastan 40 repeticiones de TV-ROF, 10 iteraciones de HUBER-ROF y 60 de TV-L1 para alcanzar el valor de convergencia. Los resultados de tiempos de ejecución se observan en la segunda columna de la tabla. De acuerdo a los resultados, el método óptimo sería el HUBER-ROF, ya que permite su aplicación en tiempo real y buena precisión. También podrían ser aplicados los métodos TV-ROF y TV-L1, ya que 5 ms es muy poco tiempo. Así, lo más importante es la calidad de la imagen obtenida. La de mayor calidad es la de TV-ROF, de forma que se escogerá TV-ROF si se prima la calidad y HUBER-ROF si se prima el tiempo de ejecución.

El análisis previo se ha considerado ruido gaussiano, donde cada píxel ha modificado su valor según el parámetro σ_r. Otra forma de considerar el efecto del ruido es a partir de la contabilización de los datos espurios. Si a una imagen se le aplica ruido de sal y pimienta, que coloca un porcentaje de los píxeles de color blanco y negro, se puede observar cómo reaccionan los diferentes métodos (ver figura F.15).

(a) Imagen ruido Sal Pimienta, 20 %

(b) Resultado TV-ROF

(c) Resultado Huber-ROF

(d) Resultado TV-L1

Figura F.15: Resultados para los modelos evaluados ante la adición de ruido de sal y pimienta.

Si bien ante un ruido Gaussiano, el algoritmo TV-L1 no era considerado como el más efectivo, ante un ruido de espurios resulta ser el más adecuado. El motivo es que la norma cuadrática L_2 suaviza los píxeles de variaciones pequeñas pero le da mucho peso a las variaciones grandes. Así, la norma cuadrática es mejor ante variaciones pequeñas mientras la norma L_1 es mejor ante variaciones grandes o espurios. La figura F.17 muestra la comparación entre normas $L2$ y L_1. La figura F.16 muestra la SnR del algoritmo TV-L1 ante ruido de Sal y Pimienta.

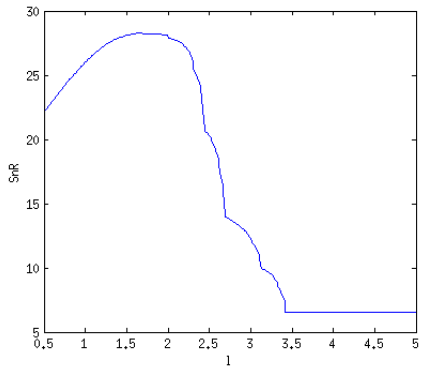


Figura F.16: Evolución de SnR frente a ruido de Sal y Pimienta en $TV - L1$

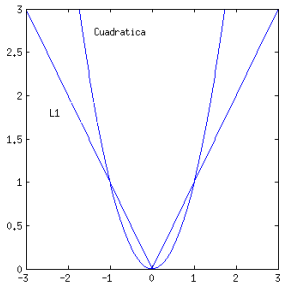
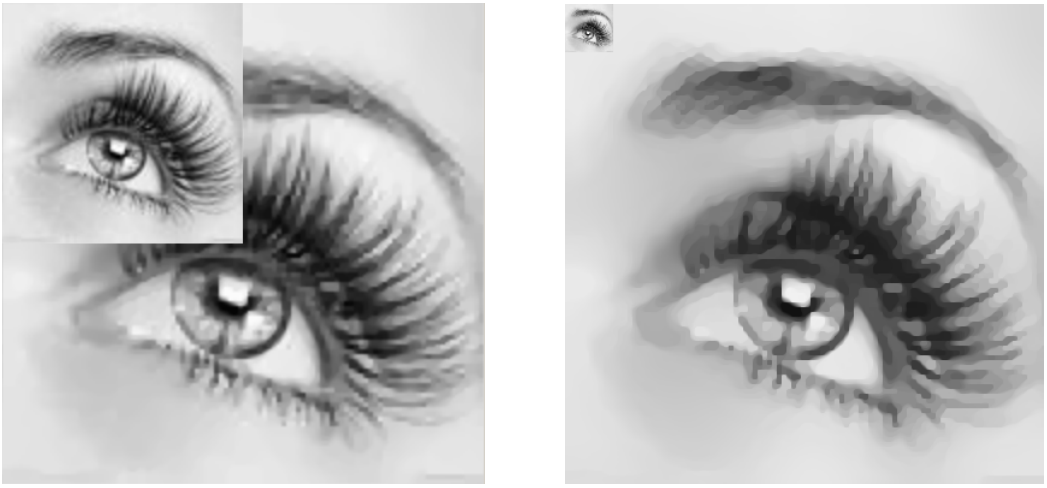


Figura F.17: Normas $L1$ y $L2$

Por este motivo, para ser capaces de tomar una decisión acerca del algoritmo a aplicar, se debe saber de antemano qué ruido va a aparecer en la imagen. De esta manera, si el ruido es Gaussiano, se aplicará un algoritmo TV-ROF o HUBER-ROF. En caso de que el ruido esté compuesto por espurios, se deberá aplicar el algoritmo TV-L1.

F.2. Resultados Zooming

La figura F.18 muestra cómo varía la imagen resultado en función del factor de ampliación s.



(a) s = 2, $\lambda = 300$ (b) s = 10, $\lambda = 300$

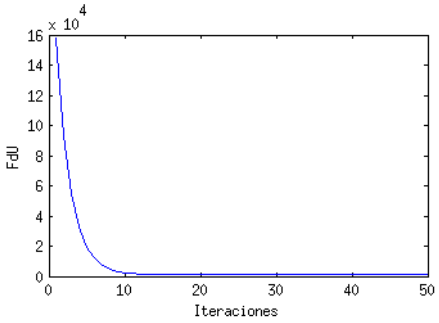
Figura F.18: Resultado del zooming para diferentes valores de escala deseados.

A diferencia de los otros problemas tratados en este proyecto, el parámetro λ puede dar lugar a la divergencia del algoritmo. Este problema se debe que el

método de Jacobi impone su propia restricción para asegurar la convergencia. Es posible probar que el algoritmo converge siempre y cuando $\lambda < \mu s^4/(s^2 - 2)$.

Es lógico pensar por tanto que el parámetro λ óptimo para cada factor de ampliación vaya a variar. Así, para un mismo λ, con un factor de ampliación de 2, la imagen aparece mucho más pixelada que en el caso de factor de ampliación de 10. Si se quiere un resultado similar, habría que reducir el parámetro λ en el caso de factor 2 o ampliarlo en el caso de factor 10. Reduciéndolo en el caso de factor 2, se obtiene un resultado muy similar (ver figura F.19, izquierda). La evolución de la energía se muestra en la figura F.19, derecha.

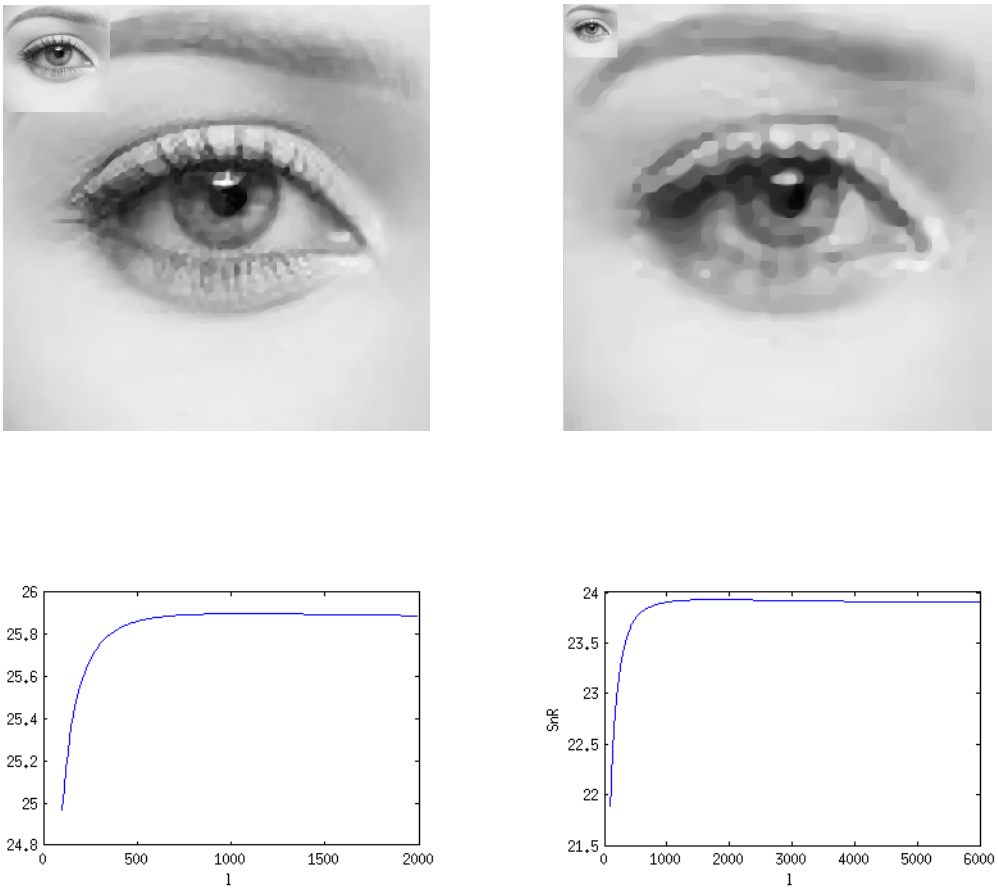
(a) Resultado óptimo



(b) SnR

Figura F.19: Resultado de zomming para s = 2, $\lambda = 100$

El método converge muy rápidamente y la función de energía decrementa mucho su valor, más que en los anteriores casos. El motivo es que anteriores apartados, la imagen resultado era inicializada al valor de la imagen dañada y en este caso la imagen resultado es inicializada con todos sus píxeles negros. El valor de λ en la optimización de parámetros sólo tendrá sentido para un factor de ampliación constante. Si se varía éste, λ variará en consecuencia. Para calcular el valor óptimo de λ hay que calcular se calcula la razón SnR. Ese parámetro requiere la imagen original sin dañar. La imagen original se puede interpretar como el resultado ideal al que convergería el método si toda la información pudiese recuperarse. Para simular este fenómeno, se toma una imagen, se reduce externamente al programa, y se aplica el algoritmo zooming con el mismo parámetro de reducción a la imagen reducida. De esta forma, se obtiene un resultado que es comparable a la imagen ideal. La figura F.20 ilustra los resultados obtenidos de evaluación de precisión.

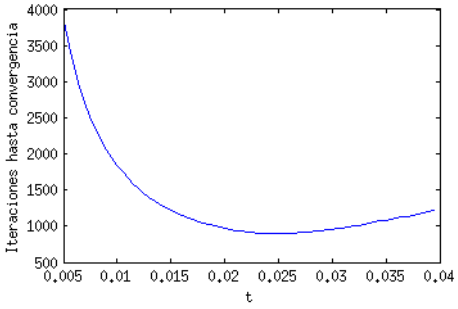


(a) SnR, s = 4 (b) SnR, s = 8

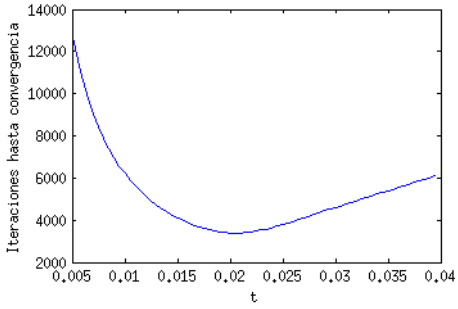
Figura F.20: Evaluación de precisión respecto a la variación del parametro λ, para s = 4 y s = 8.

En el experimento se ha tomado $\mu = 100$. Aplicando la ecuación anterior, se deduce que el valor máximo de λ que asegura convergencia es de 1828.57. En el experimento se ha superado ese valor y el método ha convergido, pero más lentamente. Observado la gráfica, el valor que se escogería es el óptimo más lejano de la no convergencia, es decir, $\lambda = 800$. En el caso de factor igual a 8, el valor máximo de λ es de 6606.45, por lo que el experimento se acota al valor 6000. Se encuentra el valor 1800 como óptimo teórico en este caso. En este algoritmo λ decrece muy lentamente una vez alcanzado el óptimo, sin embargo conviene no tomar λ muy alto, pues el método podría no converger.

A continuación, se comprueba el valor óptimo de τ. De nuevo se va realizar el mismo experimento para dos escalados diferentes y se interpretarán los resultados obtenidos:



(a) SnR, s = 4



(b) SnR, s = 8

Figura F.21: Evaluación de precisión para el parámetro τ, para s = 4 y s= 8

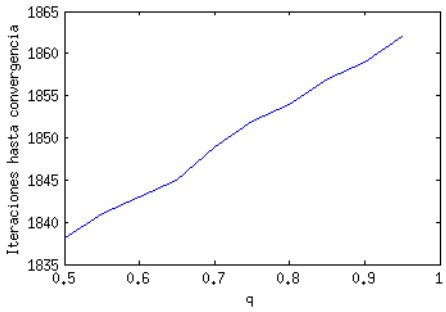
En el primer caso, el valor óptimo de τ está en torno a 0.025, donde el método converge en unas 800 iteraciones. Ésto se debe al error mínimo establecido para alcanzar la convergencia. Si es error admisible se aumenta, el método es muy rápido pues disminuye el numero de iteraciones. En el segundo caso, el valor óptimo de τ es similar, de 0.02, sin embargo las iteraciones hasta la convergencia aumentan considerablemente. Al tener un factor mayor, las iteraciones se multiplican.

θ es el último parámetro a analizar de acuerdo a su influencia en la aceleración de la convergencia. En la figura F.22 se comprueba cómo evoluciona su valor para s = 4.

Factor de ampliación	Escala	Tiempo mínimo de convergencia
4	100x100	243ms
4	50x50	90ms
8	50x50	1589ms

Tabla F.2: Tiempo de ejecución para el algoritmo de zooming

(a) Resultado óptimo



(b) SnR

Figura F.22: Evaluación de la precisión para el parámetro θ, para s = 4.

Se observa que θ debe ser lo más pequeña posible para que disminuya las iteraciones. El mínimo para asegurar la convergencia es de 0.5, por lo que se mantiene ese valor como óptimo. Adicionalmente se han realizado tres experimentos para obtener un valor aproximado del tiempo de cómputo del algoritmo. La tabla F.2 resume los resultados para diferentes valores de escala. Se puede por tanto comprobar que el tiempo de cómputo aumenta sustancialmente con el tamaño de la imagen, pero es más sensible ante variaciones en el factor de ampliación.

F.3. Resultados Deconvolution

Dada una imagen, se aplica una máscara de motion blurring de 10 píxeles de longitud de movimiento y 45º de inclinación. La figura F.23 expone la imagen original, la imagen degradada con blurring y la imagen de resultado obtenida después de la deconvolución.

(a) Imagen original

(b) Imagen con blurring, $d_p = 10$, $\alpha = 45^o$

(c) Imagen resultado $\lambda = 1000$

Figura F.23: Resultado obtenido tras la deconvolución aplicando el algoritmo primal dual.

Para evaluar la eficacia del algoritmo de deconvolución, se ha incrementado el efecto de blurring. La figura muestra el resultado para una máscara lineal de d_p=100

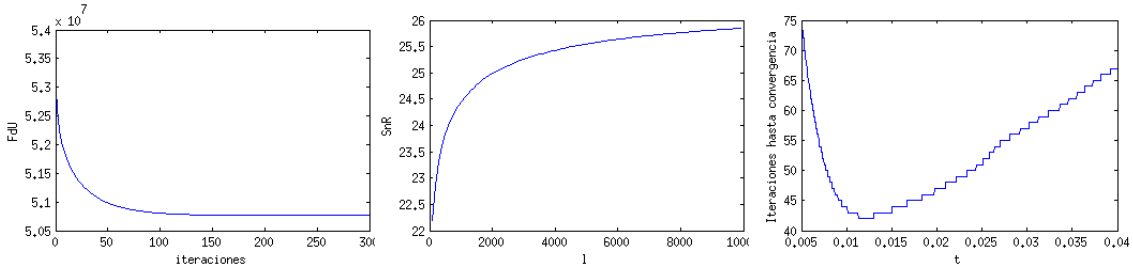
(a) Imagen original

(b) $d_p = 100$, $\alpha = 45^o$

(c) $\lambda = 1000$

Figura F.24: Resultado de deconvolución para una imagen degradada con d_p=100

Una primera conclusión que se puede obtener tras observar la evolución de la función de energía (ver figura F.25) es que ésta disminuye muy poco debido a la utilización de la transformada de Fourier que transforma el resultado del dominio espacial al de la frecuencia.



(a) Evolución de FdU, $d_p = 100$ (b) Evolución de λ (c) Evolución de τ

Figura F.25: Evaluación de parámetros

Si se observan las gráficas del parámetro λ se deduce que lo más correcto es tomar una λ infinita, pues ésta no deja de aumentar. Sin embargo, ¿qué sucede en el caso de que una imagen se vea afectada al mismo tiempo por blurring y ruido? Imaginemos un 5 % de la imagen con píxeles blancos y negros. Al transformar la imagen al dominio frecuencial y optimizar ahí el algoritmo, se intentará mantener a toda costa la cantidad de píxeles ruidosos. Además, al aplicar la máscara de forma inversa en la resolución del algoritmo, el ruido se expande. Como conclusión, se utilizará $\lambda = 1000$ como parámetro óptimo.

En cuanto al parámetro τ, se observa que para un valor de 0.012 es óptimo. El parámetro θ no se incluye pues no provoca variación en las iteraciones hasta convergencia en su rango de aplicación.

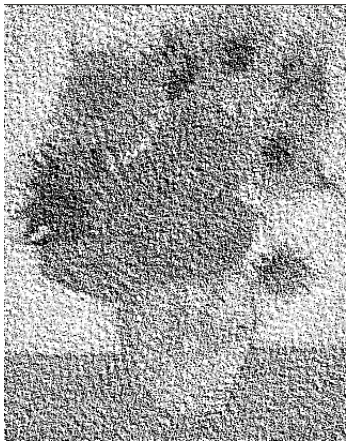
Para poder mostrar cómo evoluciona el algoritmo en caso de que haya ruido en la imagen, se ha habilitado la posibilidad de generarlo en la interfaz. De esta forma, se realiza un blurring sobre la imagen y además se añade un ruido gaussiano de $\sigma = 0.1$. La imagen dañada es la siguiente:

Figura F.26: Imagen con blurring y ruido de $\sigma = 0.1$

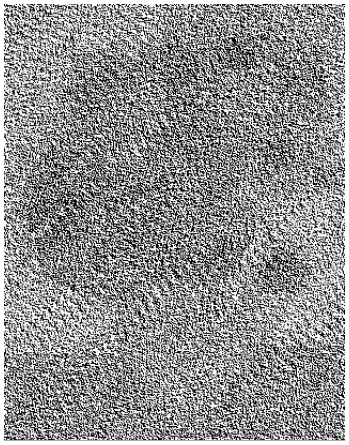
El óptimo teórico de la deconvolución es $\lambda = \infty$, se asigna $\lambda = 10000$. El óptimo teórico para solucionar el ruido está en torno a 10. De esta forma, para comprender mejor el algoritmo, se hacen cuatro experimentos: $\lambda = 10$, $\lambda = 100$, $\lambda = 1000$ y $\lambda = 10000$.

(a) $\lambda = 10$

(b) $\lambda = 100$



(c) $\lambda = 1000$



(d) $\lambda = 10000$

Figura F.27: Resultado de deconvolución para diferentes valores de λ al adicionar ruido gaussiano.

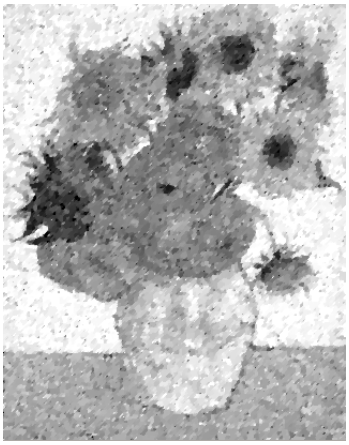
Si se observa la imagen F.27 a), no ha reparado su blurring. Sin embargo, ya no aparece nada de ruido y las superficies han sido homogeneizadas. En la imagen F.27 b) se ha removido la mayor parte del ruido y la degradación por movimiento ha sido casi reparada. La imagen F.27 c) muestra lo que sucede si λ toma un valor demasiado alto, el ruido no sólo no desaparece, sino que es deconvolucionado con

el método y se ha expandido. La imagen F.27 d) muestra el resultado para un $\lambda = 10000$ para el cual el blurring ha sido eliminado, pero el ruido persiste.

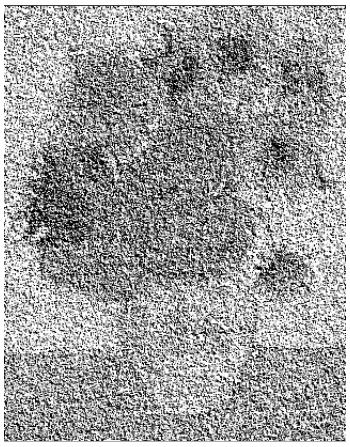
El problema que se trata de resolver en este apartado es el del blurring, no el ruido, por lo que una $\sigma = 0.1$ es muy grande para este punto. Se añade un ruido de $\sigma = 0.01$, inapreciable para el ser humano y se realiza el mismo experimento. Simulando una situación real, se empieza con una $\lambda = 1000$ que había sido sugerida en el análisis de parámetros, ya que no se sabe que existe ruido.

Figura F.28: Imagen con gran blurring y ruido de $\sigma = 0.01$

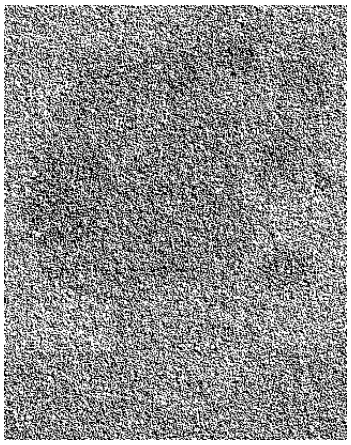
(a) $\lambda = 1000$



(b) $\lambda = 10000$



(c) $\lambda = 100000$



(d) $\lambda = 1000000$

Figura F.29: Resultado de deconvolución para differentes valores de λ al adicionar ruido gaussiano.

Como el resultado de $\lambda = 1000$ no repara el blurring completamente, se piensa en aumentar el valor de λ. Conforme se aumenta en un factor de 10, el resultado cada vez es peor. Este último análisis demuestra que pese a que un análisis teórico muestre que el λ óptimo es ∞, conviene no aumentarlo por encima de cierto valor, ya que puede haber algo de ruido inapreciable que puede empeorar drásticamente el resultado.

En cuanto a la eficiencia del método, se ha evaluado también el tiempo de ejecución. Nótese que los tiempos no varían demasiado en función de la longitud de movimiento en píxeles (Ver tabla F.3).

Longitud de movimiento	Tiempo mínimo de convergencia
10	17 ms
100	55 ms

Tabla F.3: Tiempo de ejecución para el método de convolución.

F.4. Resultados Image Inpainting

La imagen utilizada en este apartado vuelve a ser Lena.jpg, la misma que se utilizó en el apartado de denoising. Como primera simulación, se daña el 50 % de la imagen y se recupera según el algoritmo de inpainting.

Figura F.30: Imagen dañada

Figura F.31: Imagen arreglada $\lambda = 640$

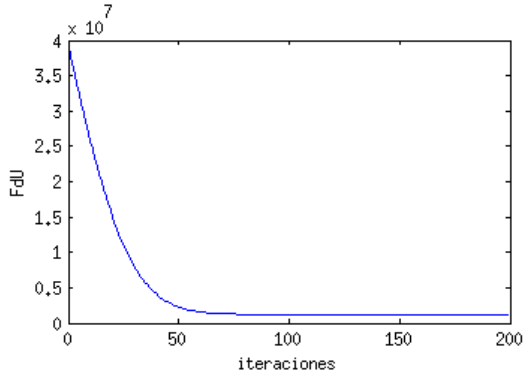


Figura F.32: Evolución de FdU

El algoritmo es muy potente, ya que es capaz de recuperar la imagen prácticamente en su totalidad. Hay que decir que en la realidad no se encontraría un resultado tan fiel a la realidad, puesto que en la función que repara la imagen se debe introducir qué píxeles son espurios. En esta aplicación se maneja la información perfecta de qué píxeles son defectuosos, lo que en la realidad no es posible. Para que este algoritmo tenga una aplicación más realista, se debe crear un algoritmo que sea capaz de calcular puntos espurios con precisión. Como ejemplo de la potencia del algoritmo si maneja información perfecta, se introduce una imagen dañada al 85 %.

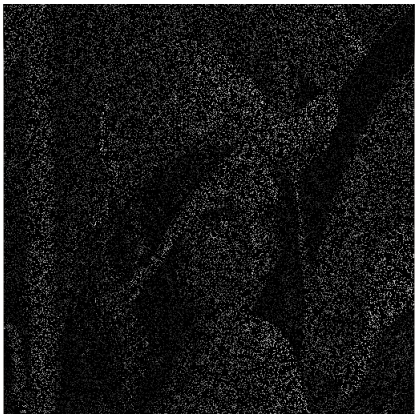


Figura F.33: Imagen dañada

Figura F.34: Imagen arreglada $\lambda = 640$

Una vez se han mostrado los resultados del algoritmo, se expone la optimización de parámetros:

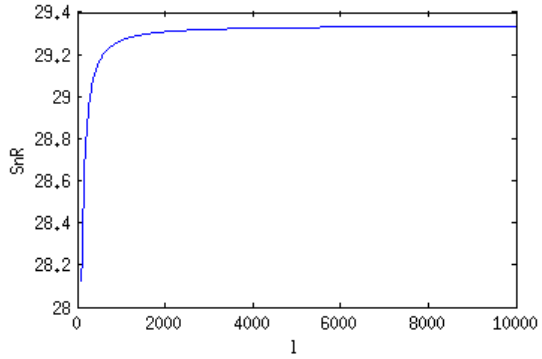


Figura F.35: Evolución de λ

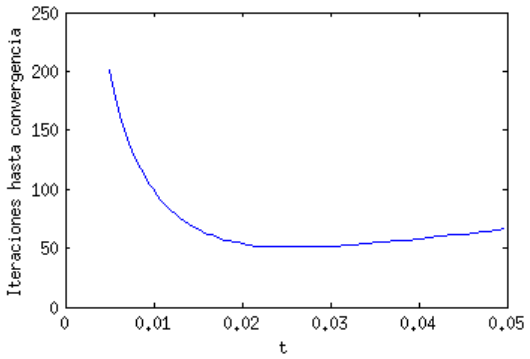


Figura F.36: Evolución de τ

Al igual que en el caso anterior, vuelve a aparecer una λ infinita como óptima. De nuevo, hay que tomar este resultado con precaución, porque en esta aplicación se maneja información perfecta. Si no se detectan todos los píxeles espurios. Para demostrarlo, se presenta a continuación una imagen dañada al 50 %, donde tan sólo se ha detectado la mitad de dañados y se aplica una λ muy elevada:

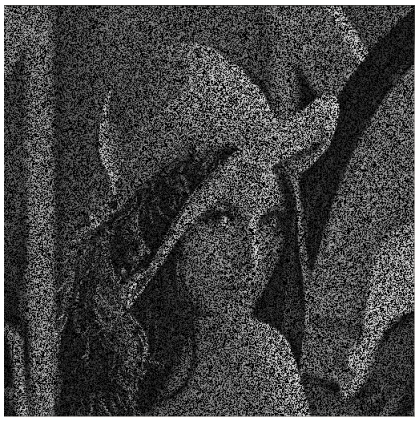


Figura F.37: Imagen dañada 50 %

Figura F.38: Imagen resultado sólo detectando 50 % espurios

Por tanto, una λ muy elevada sólo es útil ante información perfecta, de modo que es mejor no tomarla demasiado elevada. De esta manera, el valor de λ óptimo se estima en 640 y el valor óptimo de τ se estima en 0.025.

Inpainting no sólo es capaz de recuperar una imagen dañada por píxeles. Es posible que algunas filas o columnas enteras sean dañadas en algún instante del tratamiento de imagen. El algoritmo inpainting es capaz de recuperar la imagen original razonablemente bien:

Figura F.39: Imagen dañada 10 % con líneas

Figura F.40: Imagen resultado del 10 % de líneas

Figura F.41: Imagen dañada 50 % con líneas

Figura F.42: Imagen resultado del 50 % de líneas

El el caso del 10 % no se aprecia diferencia entre la imagen obtenida y la imagen real. En el caso del 50 % hay zonas de la imagen dañada donde hay hasta 10 líneas seguidas dañadas. Este hecho provoca en el algoritmo zonas en las que no es del todo preciso, pero el resultado sigue siendo muy satisfactorio.

No sólo es capaz de recuperar la imagen en caso de que haya líneas o píxeles dañados, también se puede recuperar una zona relativamente grande que ha sido perdida, de esta manera:

Figura F.43: Imagen dañada al 5 %

Figura F.44: Imagen arreglada tras 200 iteraciones

Figura F.45: Imagen arreglada tras 20000 iteraciones

El algoritmo Image Inpainting repara la zona dañada, pero cuanto mayores son las áreas a reparar, más costoso es el método. Así, con 200 iteraciones bastaba para reparar el punteado completamente, aunque éste fuera del 50 % de la imagen, ya que se tiene mucha información alrededor de cada espurio para que en pocas repeticiones se alcance el resultado óptimo. En las líneas el problema es algo mayor, aunque también se soluciona con relativamente pocas iteraciones. Sin embargo, cuando son superficies compactas las que hay que reparar, se requiere un número muy elevado de repeticiones. Así, con 200, el resultado es insatisfactorio. Sin embargo, aumentando a 20000 repeticiones, se obtiene un resultado que incluso trata de adoptar los colores del sombrero en la zona más clara y en la zona más oscura.

Figura F.46: Imagen dañada al 20 %

Figura F.47: Imagen arreglada tras 10000 iteraciones

Figura F.48: Imagen arreglada tras 200000 iteraciones

Cuanto mayor es la superficie, mayor es la cantidad de píxeles a estimar. De esta manera, si se daña la imagen con un cuadrado que ocupa el 20 % de la imagen, el resultado tras 10000 iteraciones sigue siendo insatisfactorio, ya que el algoritmo ha rellenado el área dañada con la tonalidad del fondo de la imagen. Si se aumenta la cantidad de iteraciones hasta 200000 se observa que el algoritmo ya ha convergido, no aportará solución mejor que la obtenida, de modo que está acotado.

Figura F.49: Imagen dañada en el hombro

Figura F.50: Imagen con el hombro arreglado

En la figura F.50 se puede comprender cómo evoluciona el algoritmo inpainting. Las esquinas del cuadrado dañado toman los valores próximos a ellas de forma que se prediga el interior desconociendo la profundidad de las tonalidades.

En cuanto a los tiempos, el algoritmo se comporta de la siguiente manera:

Tipo de dañado	% de daño	Tiempo mínimo de convergencia
píxeles	50 %	20 ms
píxeles	20 %	14 ms
líneas	50 %	63 ms
líneas	20 %	31 ms
área	20 %	169 ms

De esta forma, se puede observar que la variación de tiempo en función de la cantidad del % de píxeles dañados es ínfima. No es así sin embargo la variación del tiempo respecto al tipo de dañado, siendo el dañado por píxeles el más rápido, el dañado por líneas el medio y el dañado por área el más lento.

Apéndice G

Manual de usuario

G.1. Ventana Principal

Una vez ejecutado el programa, aparece la siguiente interfaz:

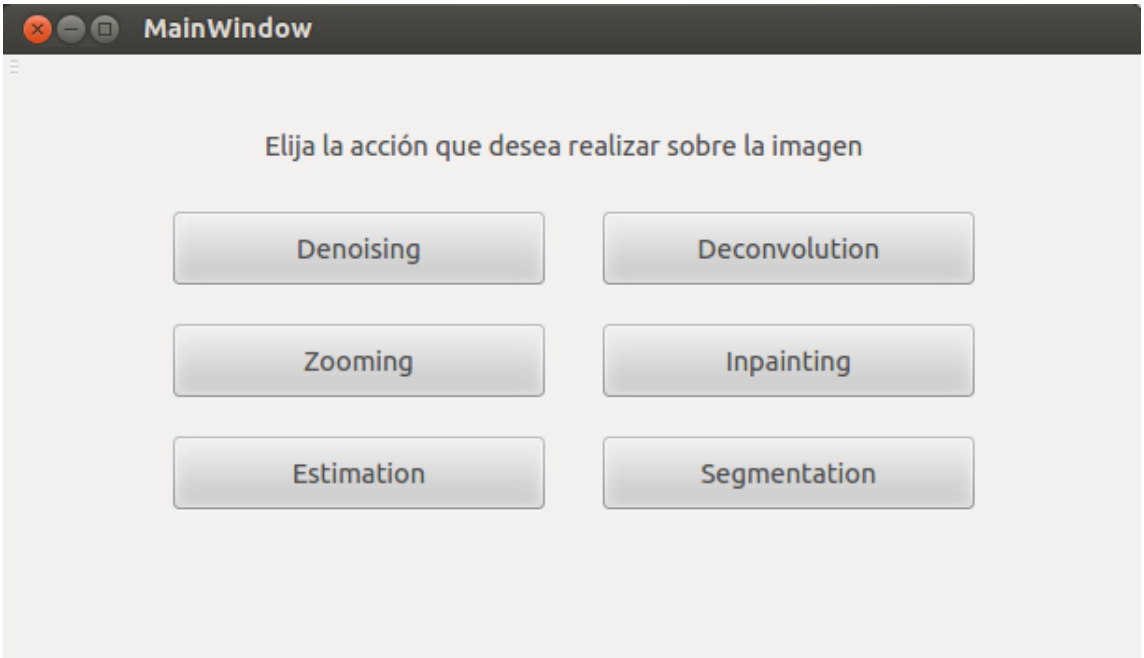


Figura G.1: Ventana Principal

Donde se puede escoger entre los cuatro programas realizados, habilitando la posible implementación futura de la Estimation y Segmentation. Una vez se pulsan los botenes correspondientes, aparecen las nuevas interfaces descritas a continuación:

G.2. Ventana Denoising

Cuando se escoge la opción Denoising se abre la siguiente interfaz de usuario:

Figura G.2: Interfaz denoising al inicio de ejecución

Punto 1. El botón lleva asignada la elección de una imagen. Al pulsarlo, se abre una ventana que permite navegar por las carpetas del sistema y escoger la imagen deseada. Automáticamente esta imagen se guarda en la memoria del programa y se muestra en el hueco habilitado para Imagen original. Siempre que se introduce una imagen, se daña o se arregla, cuando se muestra en su lugar de la interfaz, se modifica su tamaño para que se ajuste al hueco que le corresponde.

Punto 2. En el desplegable inferior al botón Adicionar ruido se escoge el tipo de ruido que se desea: un ruido gaussiano de valor σ el que se introduce en el espacio habilitado para ello, o el ruido de sal y pimienta, al que se le introduce como parámetro el porcentaje de imagen dañada que se desea. Este porcentaje se introduce en el hueco donde ahora se ve la palabra sigma. Al cambiar el tipo de ruido, se cambia el nombre sigma por % para hacerlo más intuitivo para el usuario. La imagen dañada se guarda en memoria y se muestra en el widget Imagen ruidosa.

Punto 3. La ejecución del programa depende del algoritmo utilizado. El desplegable que se encuentra en esta zona permite elegir entre TV-ROF, Huber-ROF y TV-L1. Una vez que se escoge un método se habilitan espacios para introducir los parámetros en función del método (por ejemplo, en TV-ROF, θ depende de τ y λ, por lo que sólo se habilitan estos dos últimos) con unos parámetros que convergen cargados por defecto. Una vez se pulsa el botón, se ejecuta el algoritmo en la tarjeta gráfica y muestra por pantalla la imagen resultado.

Punto 4. La gráfica muestra en escala logarítmica o decimal en función del desplegable la evolución del GAP o función de energía según el caso de los algoritmos. Así, se puede observar que en cada iteración el GAP decrece hasta 0, valor que toma cuando el método ha convergido. Por su parte, la función de energía nunca valdrá 0, alcanzará un mínimo en el que se mantendrá cuando haya convergido.

Punto 5. Tras la ejecución del algoritmo se muestra por pantalla la imagen resultado, la evolución del GAP o función de energía y el tiempo de cómputo. Este tiempo de cómputo es el que invierte el algoritmo en realizar los cálculos propios del algoritmo, descontando el tiempo invertido en calcular el GAP, ya que éste es más de cien veces mayor y es absurdo calcular el GAP en cada iteración para saber si el método ha convergido. Si no se sabe con exactitud las iteraciones necesarias para converger, es computacionalmente más barato aumentar en uno el orden de las iteraciones previstas y no calcular el GAP, que es muy costoso.

Punto 6. Las pestañas de la intefaz. Aquí se puede cambiar entre las diferentes opciones.

A continuación se muestra la interfaz una vez se ha reparado una imagen. El ruido generado es de sal y pimienta y se soluciona mediante el único algoritmo que lo soluciona: El TV-L1. Se puede observar cómo σ se ha cambiado por % al cambiar el tipo de ruido y el resultado en logarítmico. En TV-L1 se calcula la función de energía y no el GAP así que también ha cambiado en los ejes de la gráfica. El TV-L1 es un método muy rápido y con tan sólo 100 iteraciones y ante tan poco ruido, tarda muy poco tiempo en finalizar la ejecución.

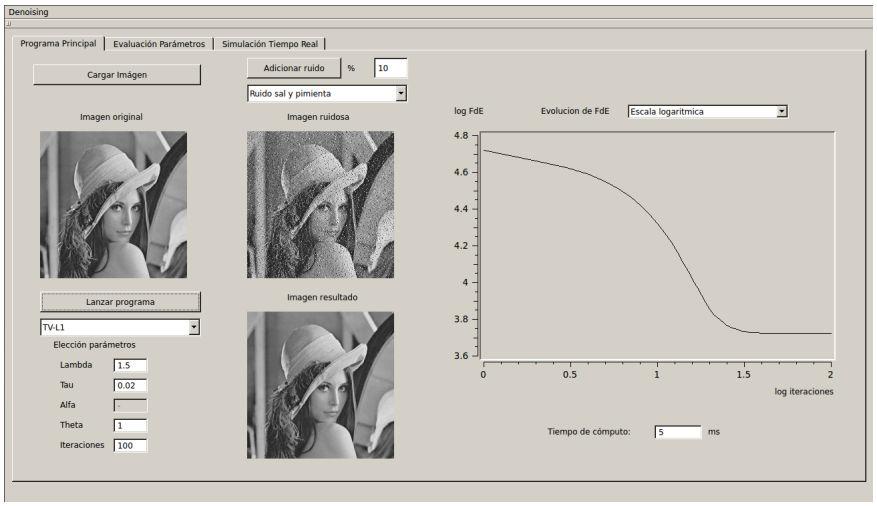


Figura G.3: Interfaz denoising tras simulación

Se muestra ahora la pestaña de Evaluación de Parámetros vacía, para explicar su funcionamiento.

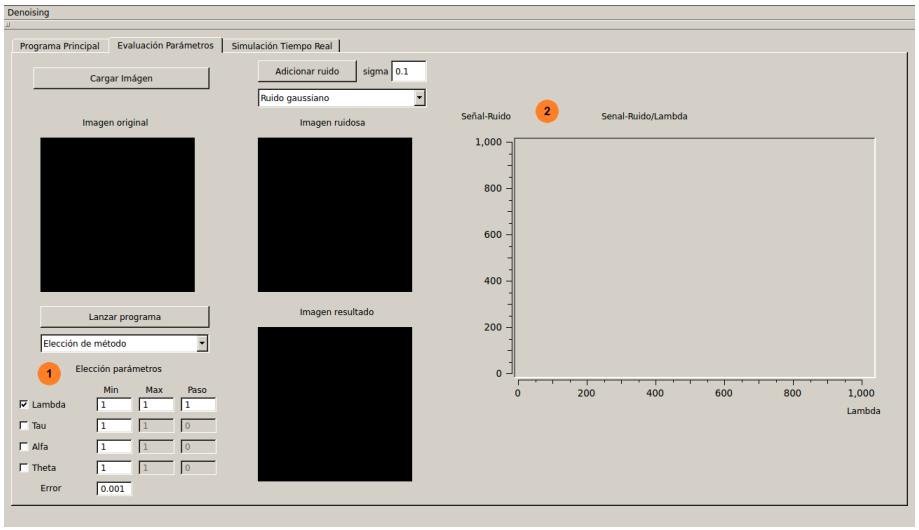


Figura G.4: Interfaz denoising parámetros al inicio de ejecución

Punto 1. De nuevo se puede escoger el método que se desee. Además, se debe elegir qué parámetro se va a evaluar colocando un tick donde se desee. Una vez se elige el parámetro, se asignan unos parámetros por defecto en sí mismo y los demás. En el escogido se habilitan tanto el mín, máx y el paso. En los demás, sólo se habilita el mín, donde se coloca el valor asignado al otro parámetro. Se evalúa el algoritmo para el valor de cada parámetro seleccionado entre el mín y el máx, iterando según el paso. La cantidad de veces que se ejecuta el algoritmo es por tanto $\frac{p_{max}-p_{min}}{p_{paso}}$ siempre redondeando hacia arriba para asegurar que se evalúan los dos extremos. Así, conforme mayor sea la diferencia entre el min y el max y menor sea el paso, más tiempo tardará en ejecutarse la evaluación de parámetros. También se debe considerar que en algunos algoritmos hay límites de convergencia en algunos de ellos. Si los límites son demasiado amplios podrían originarse problemas de no convergencia en algún método.

Punto 2. Tras la evaluación, aparecerá el ratio Señal-Ruido si se escoge λ o α y las iteraciones hasta convergencia para τ y θ. La gráfica representará el valor evaluado en el rango del parámetro.

Se muestra un ejemplo solucionado. Se trata de la evaluación de α en el algoritmo de Huber ante un ruido gaussiano.

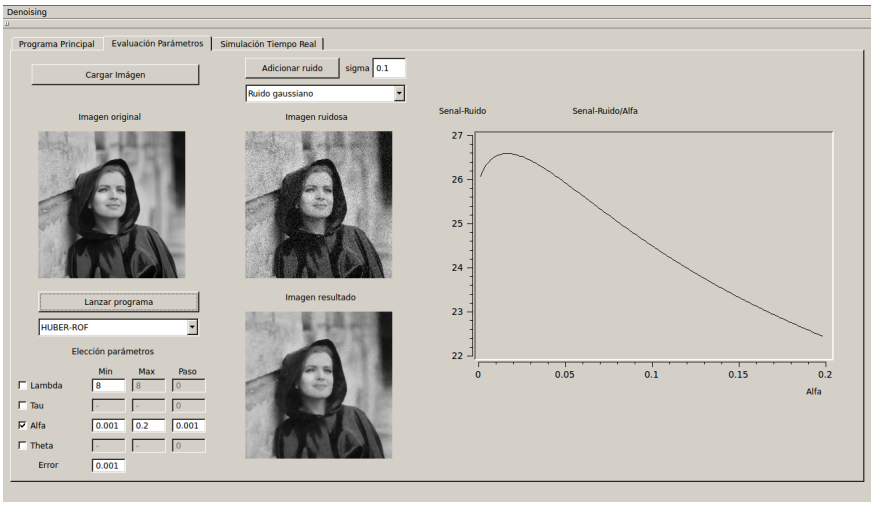


Figura G.5: Interfaz denoising parámetros al final de ejecución

G.3. Ventana Zooming

Una vez pulsada la opción de Zooming en la ventana principal, aparece la siguiente interfaz de usuario.

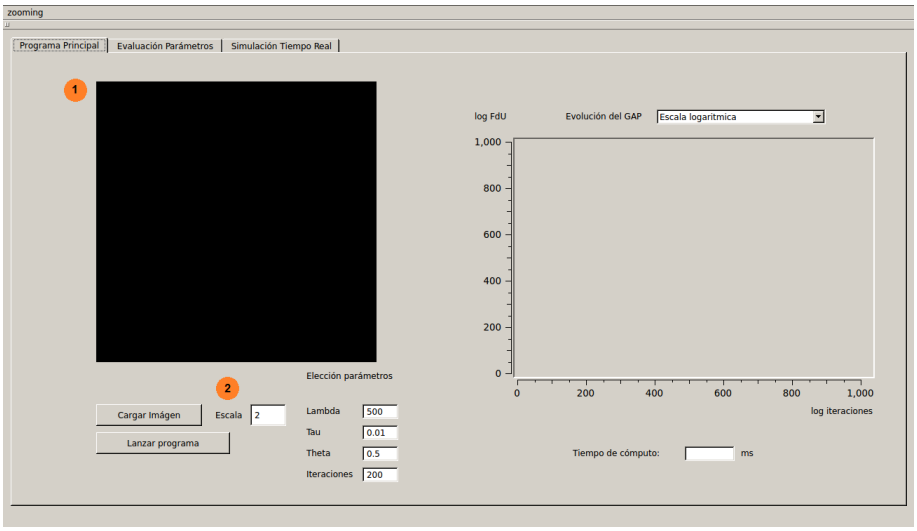


Figura G.6: Interfaz zooming al inicio de la ejecución

Punto 1. La organización de la interfaz ha cambiado un poco. En este caso, aparece un hueco grande donde aparecerá la imagen ampliada. Al ser negro en el inicio no se puede ver la zona donde aparecerá la imagen pequeña, que es la

esquina superior izquierda de la zona negra. Zooming consiste en una ampliación de la imagen de partida. Si esta imagen es de 100x100 y se amplia con un factor de 5, la imagen resultante es de 500x500. En una interfaz compacta es imposible de modelar, pues la imagen resultante se podría solapar con la gráfica o los parámetros si hubiera un factor de escalado muy grande. La decisión que se tomó para representar de una forma gráfica la ampliación es fijar el tamaño de la grande en la interfaz, variando en tamaño de la pequeña. El algoritmo trabaja con los tamaños reales pero a la hora de mostrar por pantalla se ajusta a lo dicho anteriormente. Así se puede apreciar cómo varía la relación de tamaños. Se debe introducir un factor de ampliación lo suficientemente grande respecto a la imagen original para que supere el tamaño de 400x400 que tiene el hueco para la imagen grande. Si es más pequeña el algoritmo funciona igual y la imagen queda representada de la misma manera, pero para ajustarse al tamaño se realiza una interpolación lineal que provocará que el resultado final sea una mezcla del primal dual (lo que se realiza en el algoritmo) y una interpolación lineal (lo que se trata de evitar al realizar el primal dual).

Punto 2. El factor de escalado introducido por defecto es 2. Una vez que se edita, el tamaño asignado a la imagen pequeña varía, aunque si no se ha ejecutado el programa no se aprecia al ser la zona de color negro. Se expone a continuación la interfaz resultado con dos diferentes escalados: 2 y 10.

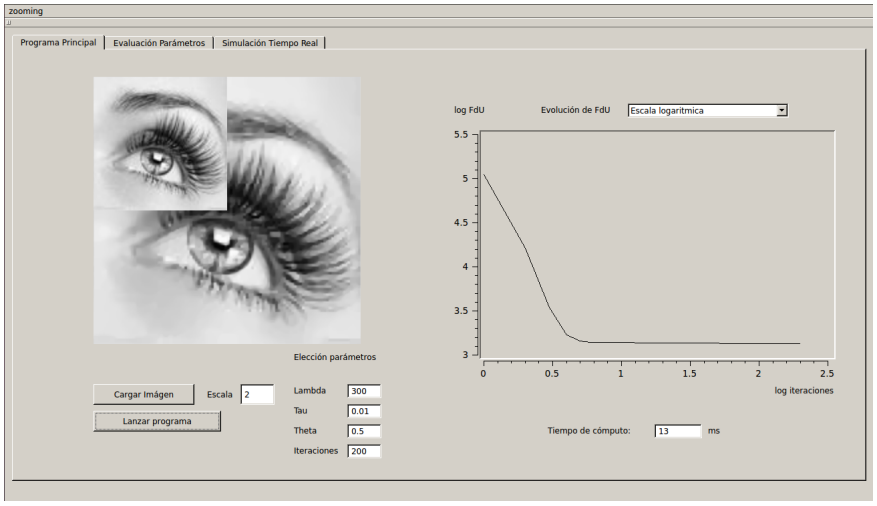


Figura G.7: Interfaz zooming al final de la ejecución, s = 2

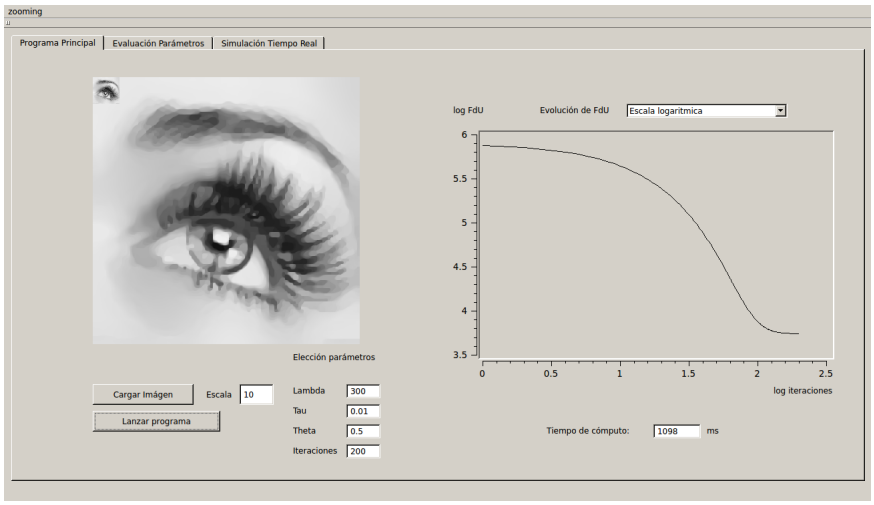


Figura G.8: Interfaz zooming al final de la ejecución, s = 10

Con los ejemplos se entiende algo mejor cómo se debe interpretar la interfaz. Para distintos escalados, se observa la diferencia existente entre la imagen original y la imagen ampliada. Además, se pueden hacer pruebas con el parámetro λ, observando que aunque en la imagen de escalado 2 parece óptimo, en la imagen de escala 10 el resultado está más difuminado. También a igual valor de τ y θ, se puede observar en la gráfica y tiempo de cómputo la diferencia en iteraciones hasta convergencia y en tiempo de cómputo existente en función del escalado.

Se muestra a continuación la interfaz de la evaluación de parámetros antes y después de la ejecución:

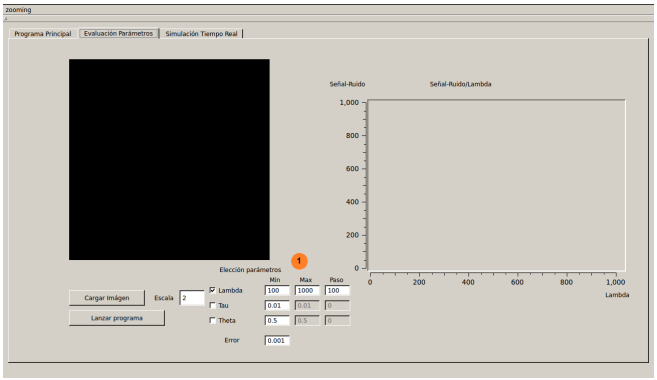


Figura G.9: Interfaz zooming parámetros al inicio de la ejecución

Punto 1. De nuevo se puede escoger qué parámetros evaluar. En este caso el resultado no depende de α de forma que el parámetro ha desaparecido. Los diferentes valores del resto son inicializados con parámetros que convergen y, al igual

que en la interfaz denoising, pueden variar como desee el usuario. Es crítico tener cuidado con la introducción del parámetro λ, que mediante una dependencia con τ y s puede provocar la no convergencia del método y podría dar como resultado una imagen completamente blanca. Es habitual en evaluación de parámetros colocar límites de evaluación muy separados con un paso muy pequeño con el fin de obtener una gráfica precisa. Estos límites deben introducirse con cuidado o la gráfica de SnR podría carecer de sentido. Si se observa en dicha gráfica algo ilógico tan como una caída o una subida muy brusca en el valor de SnR, probablemente sea porque el método no ha convergido y se puede saber a partir de qué valor de λ sucede este fenómeno. Se muestra a continuación el resultado del método convergiendo y no convergiendo:

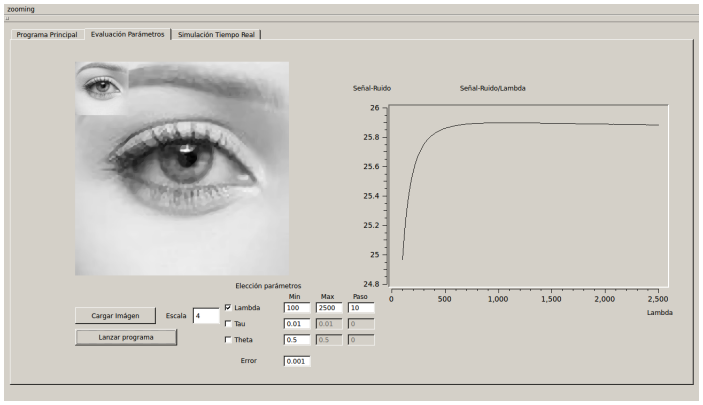


Figura G.10: Interfaz zooming parámetros al final de la ejecución que converge

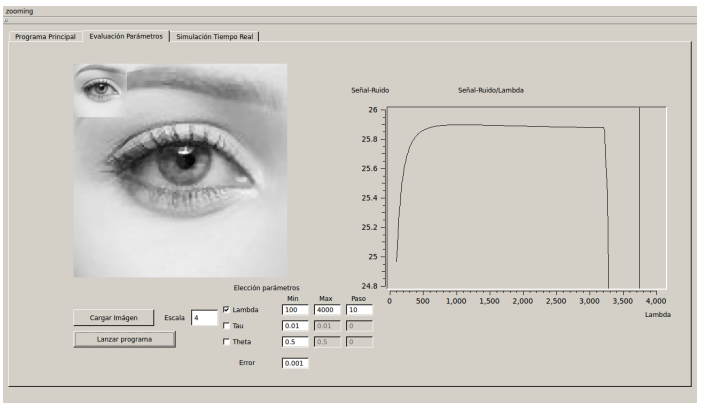


Figura G.11: Interfaz zooming parámetros al final de la ejecución que no converge

Se observa para este segundo caso que el método a partir de $\lambda = 3200$ no converge. Según la ecuación:

$$\lambda < \mu s^4/(s^2 - 2) \tag{G.1}$$

Sabiendo que $\mu = \frac{1}{\tau}$, que $\tau = 0.01$ y que $s = 4$, sólo se asegura convergencia para valores de λ menores a 1828,57. Este valor es aproximadamente el punto en el que en el primer experimento la SnR empieza a decrecer. Se debe calcular este valor antes de lanzar ninguna simulación, porque sino podría aparecer un resultado como el de la figura G.11.

G.4. Ventana Deconvolution

Una vez pulsada la opción de Deconvolution en la ventana principal, aparece la siguiente interfaz de usuario.

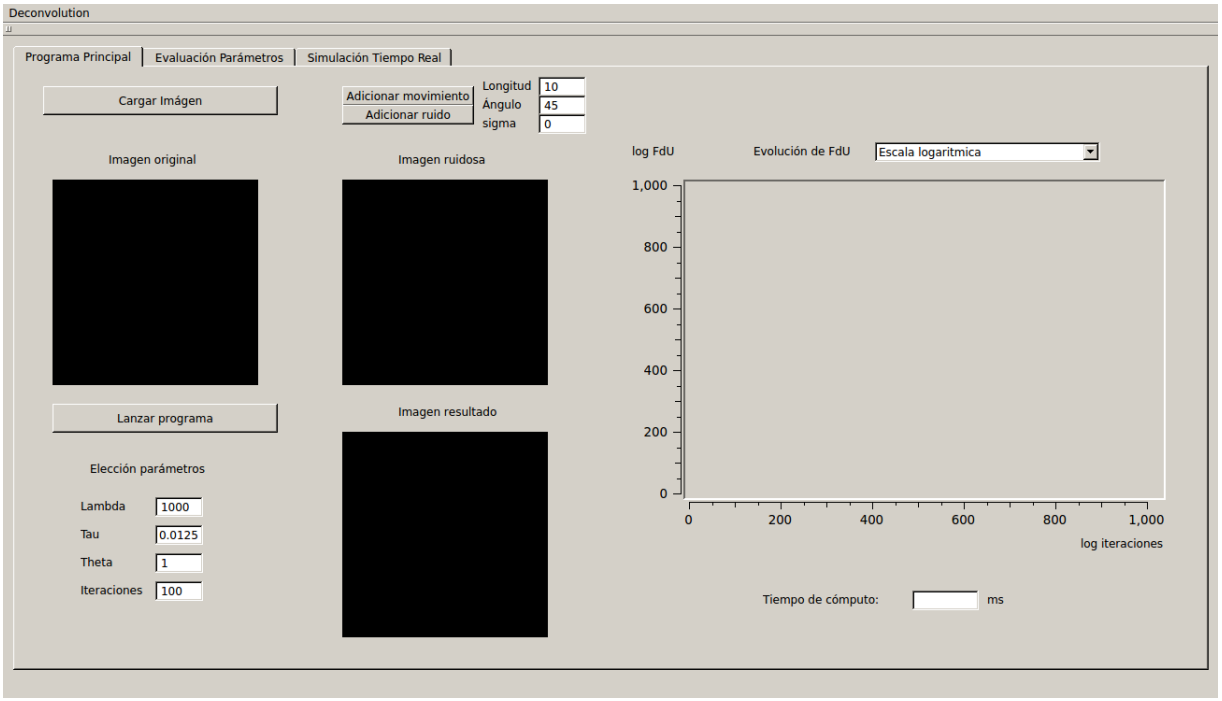


Figura G.12: Interfaz deconvolution al inicio de la ejecución

La interfaz es calcada a la de denoising. La diferencia reside en que donde antes se generaba ruido, ahora se genera movimiento. El valor de la longitud en píxeles de movimiento está acotado, ya que la imagen tiene una dimensión y en caso de exigir un movimiento demasiado grande, el algoritmo toma píxeles exteriores a la imagen, volviéndola completamente negra. Se sugiere no introducir un valor superior a 300. El ángulo no tiene limitaciones.

Una vez ejecutado el programa se obtiene lo siguiente:

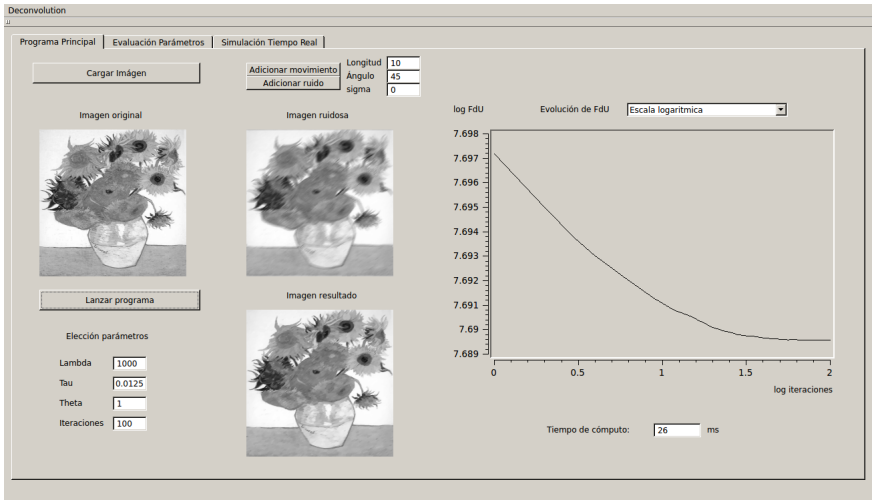


Figura G.13: Interfaz deconvolution al final de la ejecución

La pestaña de los parámetros sin ejecución toma la siguiente forma:

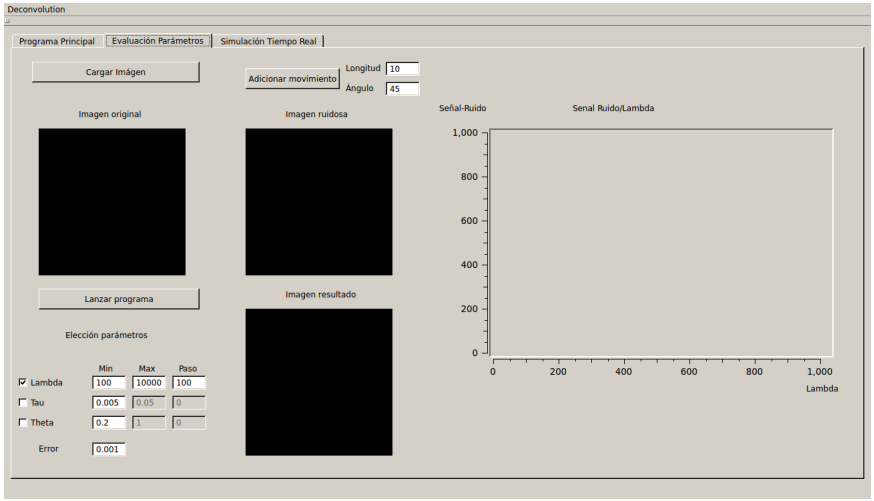


Figura G.14: Interfaz deconvolution parámetros al inicio de la ejecución

Esta interfaz no introduce novedad, de nuevo no se deben introducir valores elevados de longitud. Si se ejecuta el algoritmo aparece un resultado de la siguiente manera:

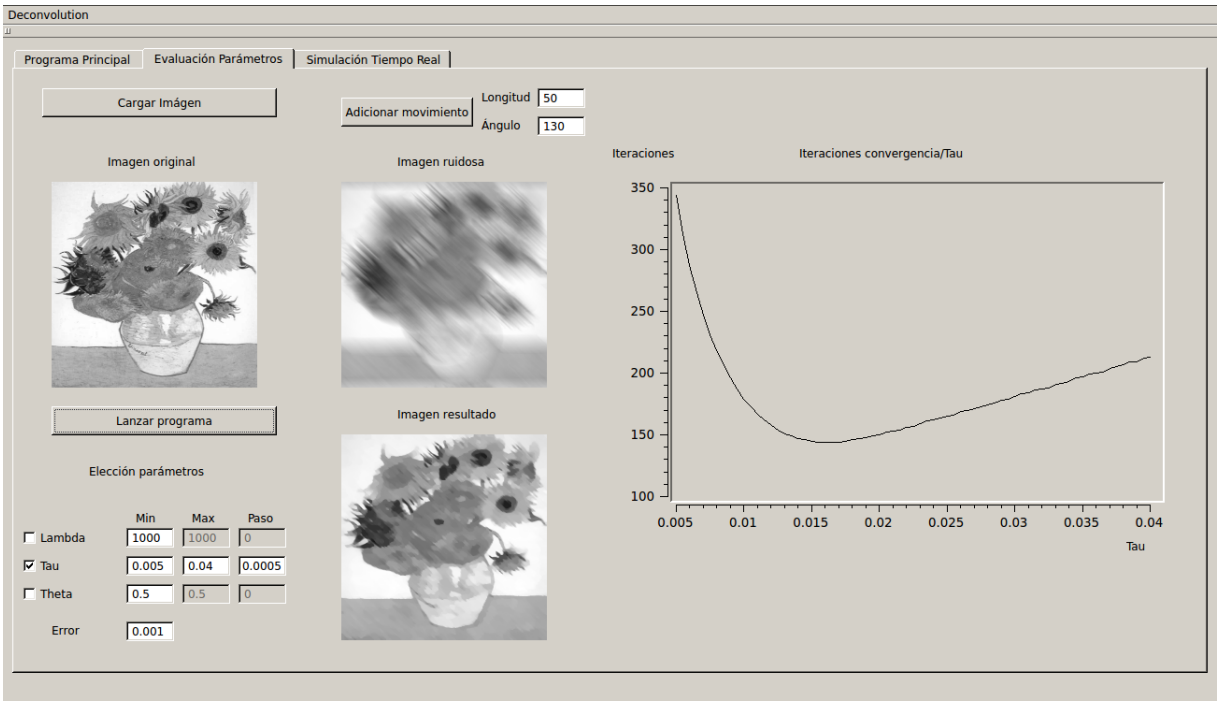


Figura G.15: Interfaz deconvolution parámetros al final de la ejecución

G.5. Ventana Inpainting

Una vez pulsada la opción de Inpainting en la ventana principal, aparece la siguiente interfaz de usuario.

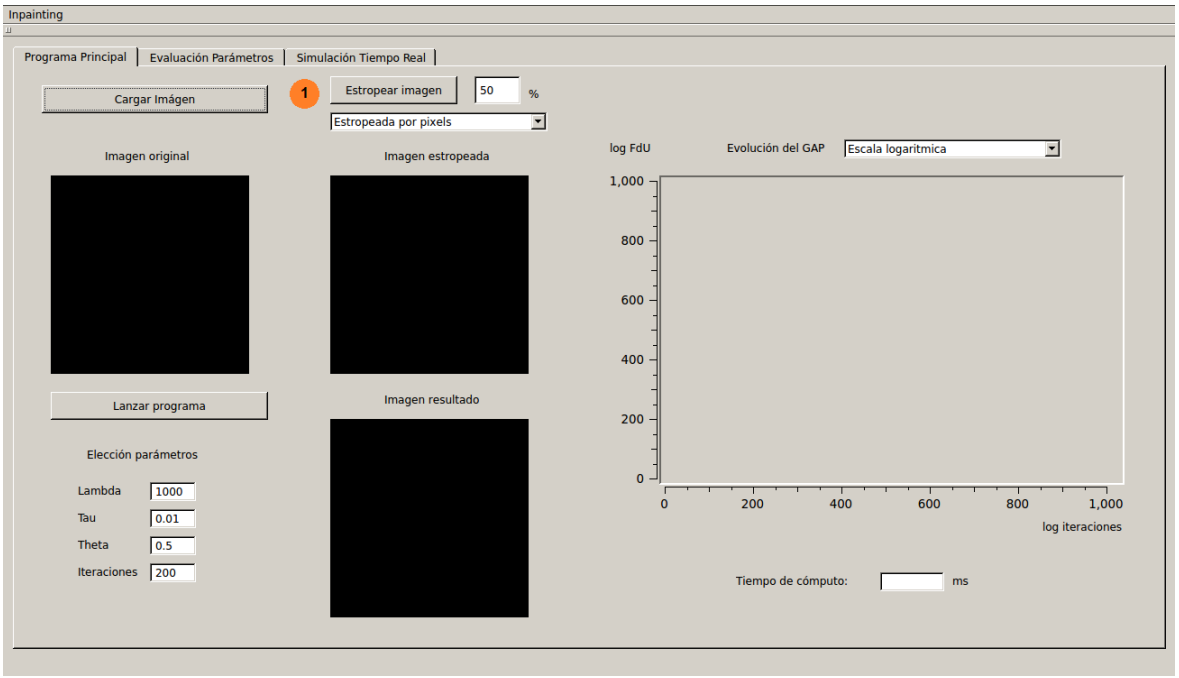


Figura G.16: Interfaz inpainting al inicio de la ejecución

Punto 1. La única diferencia que aparece en esta interfaz respecto a las anteriores es el dañado de la imagen. Se escoge el porcentaje de imagen dañada y el tipo de daño que se quiere aplicar. Se puede escoger entre dañado por píxeles, líneas o área. Este dañado es aleatorio de forma que no se escogen los píxeles, líneas o área dañada, tan sólo su magnitud.

Se presenta el resultado ante líneas:

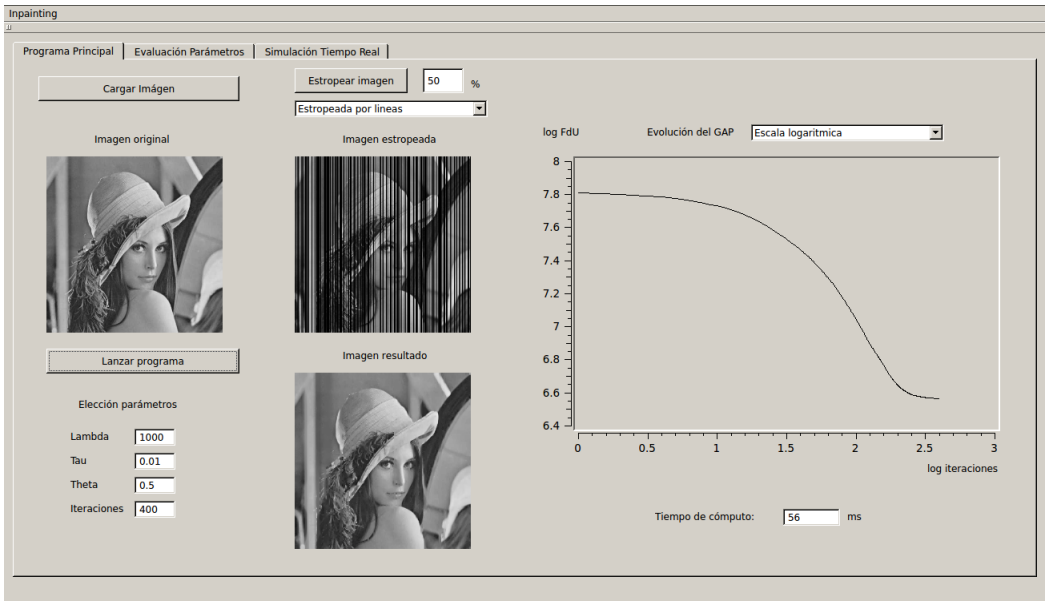


Figura G.17: Interfaz inpainting al final de la ejecución

La pestaña de parámetros toma la siguiente forma:

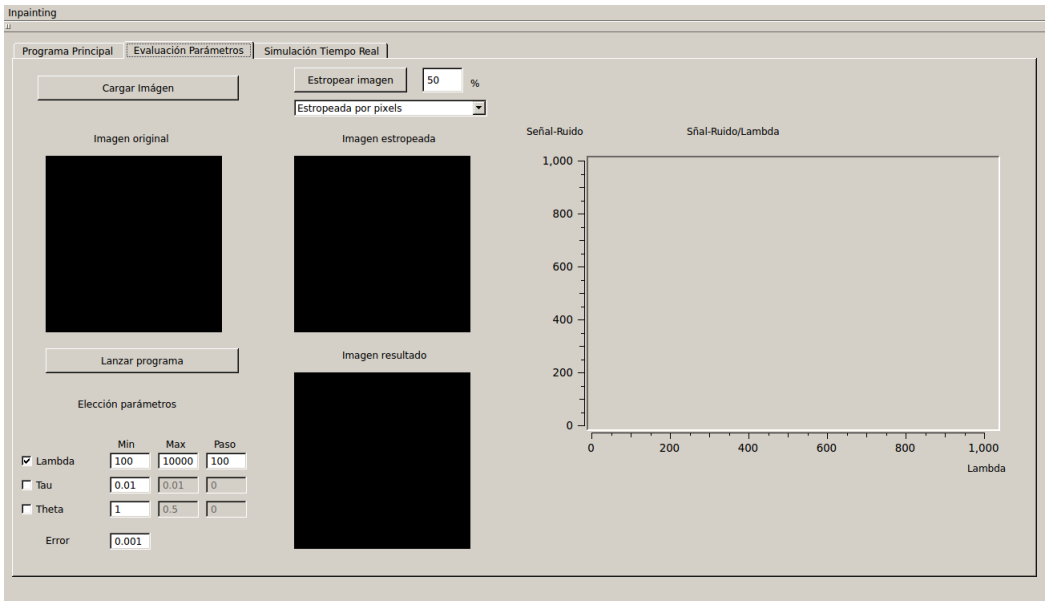


Figura G.18: Interfaz inpainting parámetros al inicio de la ejecución

Es muy similar a las anteriores, si se lanza el programa se obtiene lo siguiente:

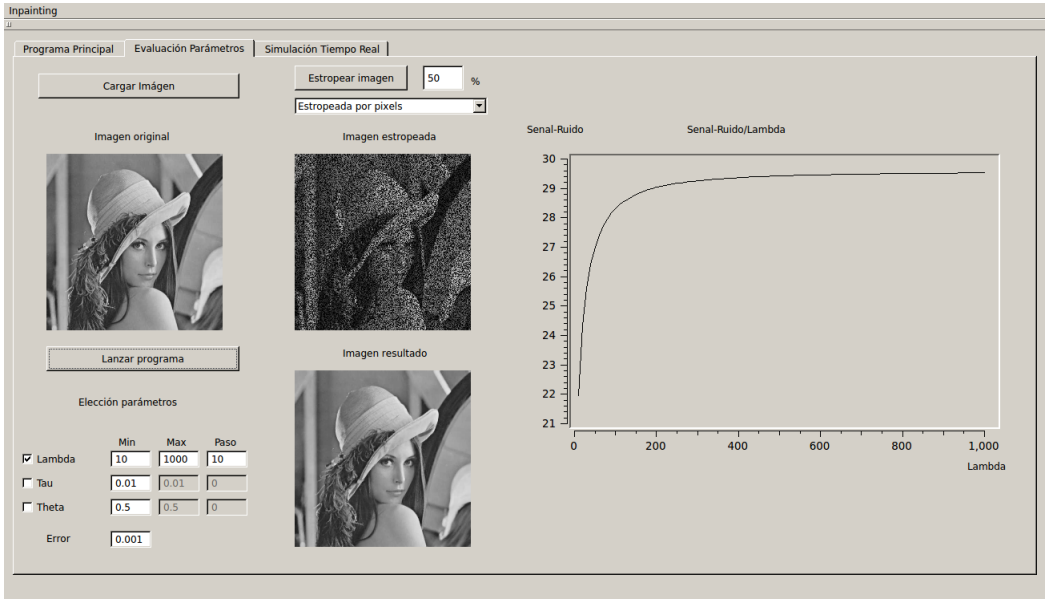


Figura G.19: Interfaz inpainting parámetros al final de la ejecución

Bibliografía

[1] A. Delshams A. Aubanell, A. Benseny. ÚTILES BÁSICOS DE CÁLCULO NUMÉRICO. 1993.

[2] Adrien Angeli Andrew J. Davison Ankur Handa, Richard A. Newcombe. Applications of Legendre-Fenchel transformation to computer vision problems. 2012.

[3] Antonin Chambolle and Thomas Pock. A First-Order Primal-Dual Algorithm for Convex Problems with Applications to Imaging. *J. Math. Imaging Vis.*, 40(1):120–145, May 2011.

[4] Edward Kandrot Jason Sanders. CUDA by Example. 2010.

[5] Raúl Cabanes Martínez. SERIES DE FOURIER. 2008.

[6] T. Pock and A. Chambolle. Diagonal preconditioning for first order primal-dual algorithms in convex optimization. In *IEEE Int. Conf. on Computer Vision (ICCV)*, pages 1762–1769, 2011.

[7] R. T. Rockafellar. *Convex Analysis*. Princeton University Press, Princeton, New Jersey, 1970.