Universidad CEU Cardenal Herrera

Departamento de Matemáticas, Física, y Ciencias Tecnológicas

Análisis computacional de mallados cuadrangulares en geometrías complejas para implementación en el método de los elementos finitos

TESIS DOCTORAL

Presentada por:
D. César Blecua Udías

Dirigida por:
Dr. D. Antonio Falcó Montesinos

VALENCIA
2017

CEU | Universidad Cardenal Herrera

TESIS DOCTORAL

Análisis computacional de mallados cuadrangulares en geometrías complejas para implementación en el método de los elementos finitos

El Doctor Don Antonio Falcó Montesinos, profesor de la Universidad CEU Cardenal Herrera informa que la Tesis, *Análisis computacional de mallados cuadrangulares en geometrías complejas para implementación en el método de los elementos finitos*, de la que es autor Don César Blecua Udías, ha sido realizada bajo su dirección y reúne todas las condiciones científicas y formales necesarias para su defensa.

Vº Bº del director:

DR. D. ANTONIO FALCÓ MONTESINOS

Valencia, 23 de enero de 2017

Para hacer las cosas bien es necesario:
primero el amor; segundo, la técnica.

A. Gaudí

Índice

Índice de figuras

Índice de tablas

Índice de algoritmos

1. Motivación y presentación

Las mallas empleadas en el método de los elementos finitos constituyen una descomposición del dominio objeto de análisis en subdominios, sin que se produzcan solapes ni zonas sin cubrir. Los subdominios pasan a denominarse elementos, que —en el caso de dominios planos— pueden ser triángulos o cuadriláteros.

En función de la topología de la malla, se puede establecer una clasificación de tipos de discretización y de algoritmos de mallado [30, 43]. En el contexto de la presente Tesis Doctoral no se han impuesto a priori limitaciones en la topología resultante, con las únicas salvedades de trabajar con dominios planos, elementos cuadrilátero, y de tratar de lograr que los elementos adquieran la máxima calidad (mínima distorsión) posible.

Los algoritmos de mallado existentes poseen en general un buen rendimiento al ejecutarse en ordenadores personales actuales, incluso en implementaciones secuenciales literales del algoritmo. Sin embargo, existen dos factores que invitan a una mejora sustancial del rendimiento:

- En la práctica profesional, el tamaño de los análisis por elementos finitos ha pasado hoy de miles a millones de nodos [30], y esto se traduce en un lógico incremento del tiempo de mallado.

- El *hardware* actual tiene un diseño de múltiples núcleos de proceso, incluso en los ordenadores más sencillos, por lo que un aprovechamiento eficiente de recursos sugiere una paralelización de los algoritmos.

El tiempo de mallado debería acercarse lo más posible a una respuesta interactiva. En problemas de tamaño pequeño no hay dificultad en lograrlo. Pero en cuanto se desea aumentar el número de nodos, el tiempo empleado tiende a crecer exponencialmente, alejándose mucho de la interactividad. Un usuario que está diseñando un modelo de cálculo, desea poder comprobar al instante los efectos de la edición que está realizando en el dominio.

Es significativo el cambio producido en la estrategia para la mejora de rendimiento en los ordenadores personales actuales. Si bien se ha seguido verificando la *Ley de Moore* [34] (duplicando el número de transistores aproximadamente cada dos años), no ha aumentado la potencia de cálculo de cada núcleo de proceso en la misma magnitud en que lo hacían en el pasado. El camino para lograr mayores rendimientos de *CPU* ha ido de la mano de incrementar el número de núcleos de proceso. Hoy en día todos los ordena-

dores personales tienen una *CPU* con dos, cuatro, o más núcleos de proceso (tabla 1).

Otra novedad es la disponibilidad y características de las *GPU* (*Graphics Processing Unit*). Con sus orígenes en los aceleradores gráficos[(1)] de los años 80 y 90, han evolucionado desde una funcionalidad inicialmente fija[(2)] hasta su actual diseño con capacidad de programación de propósito general y aplicación en cálculo científico (es un camino de desarrollo conocido como *GPGPU, "General-purpose computing on graphics processing units"*).

Siendo habitual que los ordenadores personales actuales dispongan de *GPU*, y considerando que es un componente con una elevada potencia de cálculo en coma flotante (tabla 2), y además programable para propósito general mediante *GPGPU*, surge de inmediato la idea de aprovecharlo para acelerar el mallado de elementos finitos.

El abanico de aplicaciones de *GPGPU* es tal, que algunos fabricantes tienen en el mercado *GPUs* montadas en tarjetas dirigidas exclusivamente al cálculo científico[(3)]. Estos productos son tan específicos que no disponen de salida de vídeo, y por tanto no pueden emplearse como tarjetas gráficas. Su potencia de cálculo en coma flotante de 32 bits no suele ser mayor que las de *GPUs* montadas en tarjetas gráficas, pero a cambio ofrecen otras características de interés para el sector científico al que van dirigidas (buen rendimiento en coma flotante de 64 bits, y mayores tamaños de memoria, entre otras).

En cualquier caso, en esta Tesis Doctoral se ha decidido trabajar con *GPUs* dirigidas a un público general, con la finalidad de que las conclusiones obtenidas sean aplicables a los ordenadores personales de uso cotidiano.

Antes de proseguir, conviene hacer una observación sobre la diferencia entre el diseño de una *CPU* y el de una *GPU*. Las *GPUs* cuentan con un número de núcleos mucho mayor que una *CPU*, pero se trata de máquinas de tipo *SIMD* (*Single Instruction, Multiple Data*). Su elevado rendimiento en coma flotante es inseparable de este diseño *SIMD*, y es imprescindible te-

(1) Componente habitual en las *estaciones de trabajo* empleadas en ingeniería, ciencia, y diseño gráfico en la época. Un claro precursor de las *GPU* lo constituyen los subsistemas gráficos de las estaciones y supercomputadores *Silicon Graphics* y, en concreto, el concepto de *Geometry Engine* de Clark [9].

(2) Estaban diseñados para acelerar el dibujo de gráficos exclusivamente. En algunos casos hubiese sido técnicamente posible reprogramarlos para aplicaciones no gráficas, pero los fabricantes no solían facilitar las herramientas de programación de microcódigo requeridas.

(3) Estamos hablando de productos como *NVIDIA Tesla* y algunos modelos de la gama *AMD FirePro S*.

nerlo en cuenta al implementar algoritmos en *GPU*. No todos los algoritmos son trasladables de manera óptima a una máquina *SIMD*. En particular, algoritmos que requieran ramificaciones en el código (fruto de sentencias condicionales) requieren de un estudio especial para lograr un rendimiento eficiente en máquinas *SIMD*.

Todo ello nos conduce a que sea también objeto de estudio la viabilidad del empleo de *GPUs* para acelerar el mallado de elementos finitos, dada su amplia disponibilidad.

De hecho, en la última década ha surgido un interés por el desarrollo de nuevos algoritmos de mallado acelerados en *GPU*. Sin embargo, la investigación realizada al respecto se centra en acelerar en *GPU* algoritmos de mallado triangular (véanse las referencias [4, 14, 16, 25, 35, 36, 45, 46, 47, 48]), sin entrar a considerar el caso del mallado con elementos cuadrangulares.

En el contexto del hardware actual que acabamos de describir, es decir, *CPUs* de múltiples núcleos, y utilización de *GPUs* para tareas de propósito general, es donde cabe plantearse la optimización de rendimiento de los algoritmos de mallado cuadrangular, al ser los recursos disponibles en los ordenadores personales habituales de hoy.

Otro aspecto de interés en el mallado cuadrangular es la de la mejora de la calidad de sus elementos mediante un postproceso de suavizado o relajación. En efecto, la calidad de la forma de un elemento es cuantificable [49] en cuanto a su virtud para modelizar de manera adecuada un problema para su análisis mediante método de elementos finitos. Al respecto, existen algoritmos aplicables a este postproceso y cabe utilizarlos, o bien reconsiderarlos o ampliarlos si se encuentra necesario.

	Año	Transistores[4]	Núcleos[5]
Intel 4004	1971	2300	1
Intel 8008	1972	3500	1
Intel 8080	1974	4400	1
Zilog Z80	1976	8500	1
Intel 8086	1978	29000	1
Motorola 68000	1979	68000	1
Intel 80286	1982	134000	1
Motorola 68020	1984	190000	1
Intel 80386	1985	275000	1
MIPS R2000	1986	110000	1
Motorola 68030	1987	273000	1
MIPS R3000	1988	150000	1
Intel 80486	1989	1180235	1
Motorola 68040	1990	1200000	1
MIPS R4000	1991	1200000	1
Intel Pentium	1993	3100000	1
Motorola 68060	1994	2500000	1
MIPS R8000 + R8010	1994	3430000	1
Intel Pentium Pro	1995	5500000	1
MIPS R10000	1996	6800000	1
PowerPC 7xx *(G3)*	1997	6350000	1
Intel Pentium II	1997	7500000	1
MIPS R12000	1998	7150000	1
Intel Pentium III	1999	9500000	1
PowerPC 74xx *(G4)*	1999	10.5 millones	1
Intel Pentium 4	2000	42 millones	1
PowerPC 970 *(G5)*	2002	58 millones	1
Intel Core 2 Duo	2006	291 millones	2
Intel Core i7 (quad)	2008	731 millones	4
Intel Core i7 (6 core)	2010	1170 millones	6
Intel Xeon Sandy Bridge-E5 (8 core)	2011	2270 millones	8
Intel Xeon Ivy Bridge-EX (15 core)	2014	4310 millones	15
Intel Xeon Haswell-E5 (18 core)	2014	5560 millones	18
Intel Xeon Broadwell-E5 (22 core)	2016	7200 millones	22

Tabla 1: Número de transistores y núcleos de una selección de CPUs (según [20] y especificaciones de fabricantes)

[4] Obsérvese que, en los diseños recientes, el aumento del número de transistores va a la par con un incremento del número de núcleos de proceso.

[5] En un mismo chip.

Tabla 2: Breve selección, comparativa y evolución de GPUs.

	Chip[6]	Año	gpgpu[7]	Transistores	GFLOPS[8]	Precio[9]	Sector[10]
Silicon Graphics Reality Engine	GE8/RM4/DG2	1993	No	No publicado	0.6-1.2[11]	≃$100000	Pro
Silicon Graphics Max. IMPACT	GE11/RE4	1995	No	No publicado	0.96	≃$30000	Pro
Silicon Graphics Infinite Reality	GE12/RM6/DG4	1996	No	260 mill.[12]	2.16[13]	≃$100000	Pro
NVIDIA GeForce 256	NV10	1999	No	22 millones	50	$300	Juegos
NVIDIA Quadro[14]	NV10GL	1999	No	22 millones	>50[15]	$900	Pro
NVIDIA GeForce 3	NV20	2001	No	57 millones	46-76	$500	Juegos
NVIDIA Quadro DCC[16]	NV20GL	2001	No	57 millones	46-76	$650	Pro
NVIDIA GeForce 6800 Ultra	NV40	2004	No	222 millones	200	$300-$500	Juegos

(Continúa)

[6] Reality Engine, IMPACT, e Infinite Reality no son propiamente GPUs, sino *sets* de varias placas que comprenden un elevado número de chips.

[7] Indica si es programable para propósito general. No se han considerado como programables aquellas GPU que ofrecen una programabilidad parcial, mediante *shaders* o soluciones similares.

[8] Referido a coma flotante de 32 bits.

[9] Precio de lanzamiento de las tarjetas gráficas en EEUU (sin incluir el ordenador o estación).

[10] El precio de tarjetas gráficas basadas en una misma GPU puede variar bastante en función del sector al que van dirigidas. Es una estrategia habitual entre los fabricantes de GPUs, introduciendo diferencias de funcionalidad o certificación en los drivers, o en ocasiones variando ligeramente la funcionalidad del chip, como por ejemplo reduciendo el rendimiento del cálculo en coma flotante de 64 bits (precisión *double*). Esto permite rentabilizar mejor cada nuevo chip, al poder comercializarlo a diferente precio en diferentes sectores.

[11] Admite 6, 8, ó 12 *Geometry Engines*, siendo cada uno de ellos un Intel i860XP con 100 MFLOPS.

[12] 260 millones por cada *pipe* en configuración máxima.

[13] Cada *pipe* tiene cuatro GE12 con 540 MFLOPS cada uno.

[14] Prácticamente idéntica a GeForce 256, pero con funcionalidad adicional dirigida al mercado profesional, como el *antialiasing* de puntos y líneas.

[15] NV10GL opera a 135MHz, ligeramente superior a los 120MHz de NV10.

[16] Mismo *hardware* que GeForce 3, aunque dirigida al sector profesional.

Tabla 2 *(Continuación)*

NVIDIA Quadro FX 4000[17]	NV40GL	2004	No	222 millones	200	$2200	Pro
AMD ATI Radeon HD 2900	R600	2007	No	700 millones	288-475[18]	$400	Juegos
AMD ATI FireGL V8600[19]	R600	2007	No	700 millones	440	$1600	Pro
NVIDIA GeForce 8800 GT	G92	2007	Sí	754 millones	504	$250-$300	Juegos
NVIDIA Quadro FX 3700[20]	G92	2008	Sí	754 millones	420	$1600	Pro
AMD ATI Radeon HD 5870	Cypress XT	2009	Sí	2154 millones	2720	$380	Juegos
AMD ATI FirePro V8800	Cypress XT	2010	Sí	2154 millones	2640	$1499	Pro
NVIDIA GeForce GTX 480	GF100	2010	Sí	3200 millones	1345	$500	Juegos
NVIDIA Quadro 6000[21]	GF100	2010	Sí	3200 millones	1028	$5000	Pro
NVIDIA Tesla C2050[22]	GF100	2010	Sí	3200 millones	1030	$2500	Pro
Intel HD Graphics 3000	Sandy Bridge	2011	No	(en CPU)[23]	130[24]	(en CPU)	Juegos
NVIDIA GeForce GTX 680	GK104	2012	Sí	3540 millones	3090	$500	Juegos
Intel HD Graphics 4000	Ivy Bridge	2012	Sí	(en CPU)	332	(en CPU)	Juegos

(Continúa)

[17] Análoga a GeForce 6800 Ultra, pero dirigida al sector profesional (es posible convertir una en otra modificando el *firmware*).

[18] Según modelo.

[19] Al igual que NVIDIA, ATI (adquirida por AMD en 2006) ofrece una línea profesional en paralelo a los productos de juegos, compartiendo diseños entre ambas líneas.

[20] Emplea el mismo chip que GeForce 8800 GT, pero corre a menor frecuencia de reloj, por lo que su rendimiento es algo menor a pesar de ir dirigida al sector profesional.

[21] Menor frecuencia de reloj que GeForce GTX 480, pero dirigida al sector profesional.

[22] La gama Tesla generalmente no dispone de salidas de vídeo, estando dirigida al cálculo con GPGPU exclusivamente. Proporciona además un rendimiento en precisión *double* superior al de GPUs enfocadas a juegos

[23] Las GPU de Intel se encuentran integradas en la CPU.

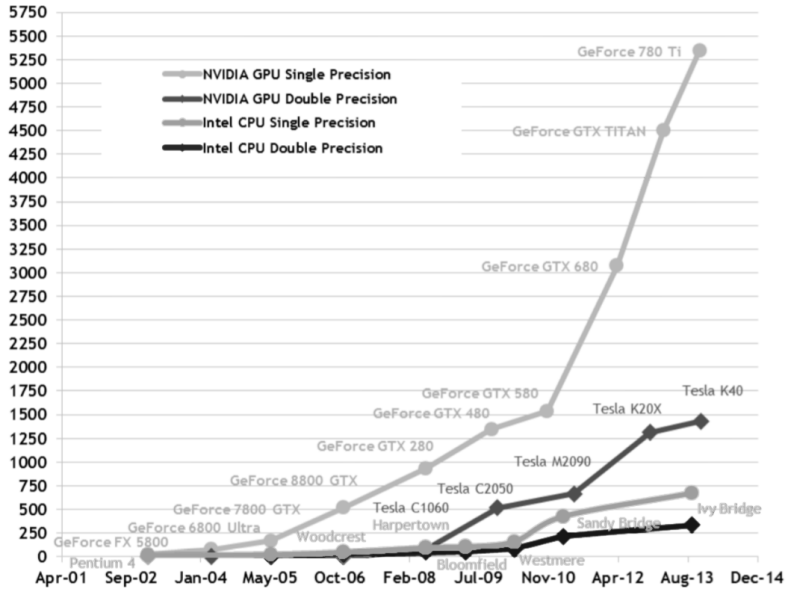
[24] Las GPU de Intel suelen ofrecer rendimientos inferiores a las GPU no integradas en CPU.

Tabla 2 *(Continuación)*

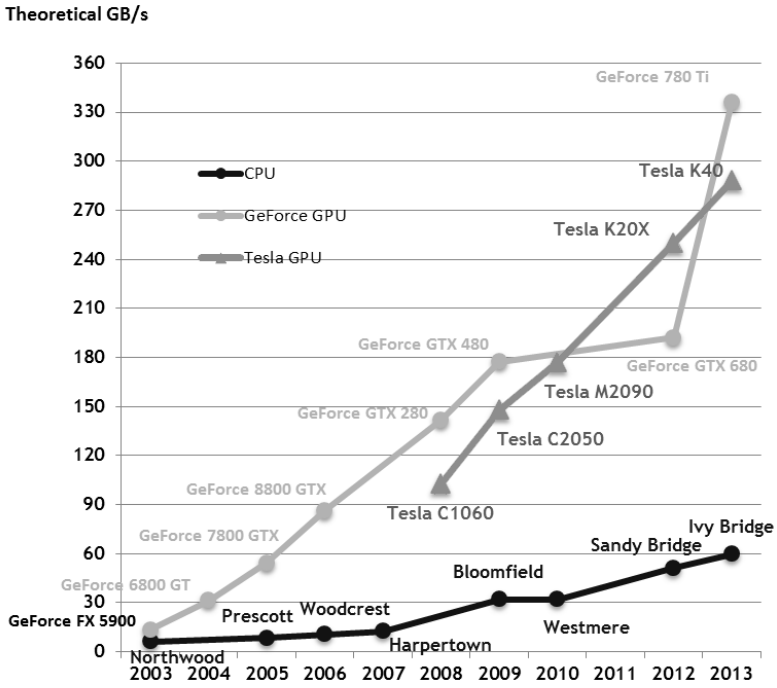
AMD Radeon HD 7970	Tahiti XT	2012	Sí	4313 millones	3788	$550	Juegos
AMD FirePro W9000	Tahiti XT	2012	Sí	4313 millones	3993	$3999	Pro
NVIDIA Tesla K20X	GK110	2012	Sí	7080 millones	3950	≃$3500	Pro
NVIDIA GeForce GTX 780	GK110	2013	Sí	7080 millones	3976	$500-$650	Juegos
NVIDIA GeForce GTX Titan	GK110	2013	Sí	7080 millones	4500	$999	Semiprof.
NVIDIA GeForce GTX 780 Ti	GK110	2013	Sí	7080 millones	5046	$699	Juegos
Intel Iris Pro Graphics 5200	Haswell	2013	Sí	(en CPU)	832	(en CPU)	Juegos
AMD FirePro W9100	Hawaii XT	2014	Sí	6200 millones	5237	$4000	Pro
Intel Iris Pro Graphics 6200	Broadwell	2015	Sí	(en CPU)	883	(en CPU)	Juegos
AMD Radeon R9 Fury X	Fiji XT	2015	Sí	8900 millones	8601	$649	Juegos
NVIDIA Tesla P100	GP100	2016	Sí	15300 millones	10600	≃$10000	Pro
Intel Iris Pro Graphics 580	Skylake	2016	Sí	(en CPU)	1152	(en CPU)	Juegos
NVIDIA GeForce GTX 1060	GP106	2016	Sí	4400 millones	4400	$250	Juegos
NVIDIA GeForce GTX 1080	GP104	2016	Sí	7200 millones	9000	$600	Juegos
NVIDIA Titan X (Pascal)[25]	GP102	2016	Sí	12000 millones	11000	$1200	Semiprof.
AMD Radeon RX 480	Polaris 10[26]	2016	Sí	8900 millones	5161	$199-$239	Juegos

[25] La gama Titan (cuyo primer modelo fue lanzado en 2013 y estaba basado en el chip GK110) va dirigida a un mercado semiprofesional, aunque también goza de mucha aceptación en el sector de los juegos.

[26] En el momento de realizar esta tabla, AMD ya ha anunciado GPUs de gama profesional basadas en *Polaris*, pero no están aún disponibles.



(a) Valores teóricos de operaciones de coma flotante por segundo *(FLOPS). Nota: Los valores de la gama Tesla se refieren a coma flotante de 64 bits, al contrario que en la tabla 2, en la cual se muestra el rendimiento de coma flotante de 32 bits para todas las GPU.*



(b) Valores teóricos de ancho de banda.

Figura 1: Comparativa de rendimiento de *CPUs* y *GPUs* según NVIDIA [40].

2. Introducción y antecedentes

2.1. Clasificación de algoritmos de mallado cuadrangular

Los algoritmos de mallado cuadrangular pueden clasificarse según varios criterios. Siguiendo los trabajos de S. H. Lo [30] y Owen [43], estableceremos la clasificación mostrada en los siguientes apartados.

2.1.1. Mallado estructurado y no estructurado

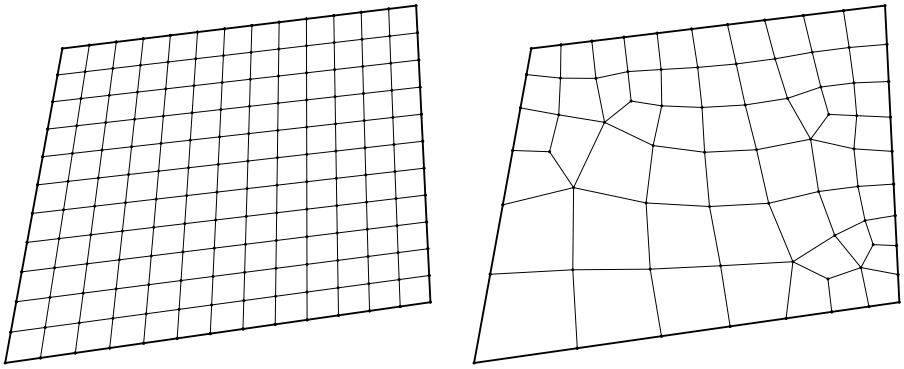
En el mallado **estructurado**, todos los nodos interiores están conectados al mismo número de elementos (es decir, en un mallado cuadrangular estructurado todos los nodos interiores estarían conectados a cuatro elementos). Este requisito da lugar a una topología con forma de rejilla o cuadrícula, lo que explica que en inglés se emplee la palabra *grid* para estas discretizaciones, y se etiquete a estos algoritmos como *grid generation* en lugar de *mesh generation*, término más general.

En el mallado **no estructurado** no existe dicho requisito, y cada nodo puede estar conectado a un número diferente de elementos.

En función del tipo de análisis que se vaya a realizar, puede ser preferible elegir uno de estos tipos, o bien resultar indiferente tomar uno u otro.

Por ejemplo, si se desea que la discretización tenga un alineamiento estricto de los elementos para la simulación de magnitudes físicas con una direccionalidad muy marcada, el mallado estructurado se hace necesario. Sin embargo, en general y salvo necesidades concretas, no es imprescindible el mallado estructurado.

El mallado no estructurado facilita la discretización con variaciones en el tamaño de los elementos (figura 2), además de adecuarse mejor a perímetros con irregularidades importantes. Por ello, posee una mayor generalidad en su aplicación, y centraremos el estudio en este tipo. Además, el mallado estructurado y el no estructurado son dos campos bastante diferenciados, cada uno de ellos con sus propios algoritmos.



(a) Estructurado. (b) No estructurado.

Figura 2: El mallado estructurado conduce a una topología muy rígida, que puede dar buenos resultados cuando el perímetro tiende a ser regular y no hay variaciones en el tamaño de los elementos. Por contra, el mallado no estructurado facilita adecuarse a contornos muy irregulares, o a variaciones en el tamaño de los elementos. La figura de la derecha tiene asignadas diferentes densidades de mallado en cada esquina del contorno.

2.1.2. Mallado cuadrangular indirecto y directo

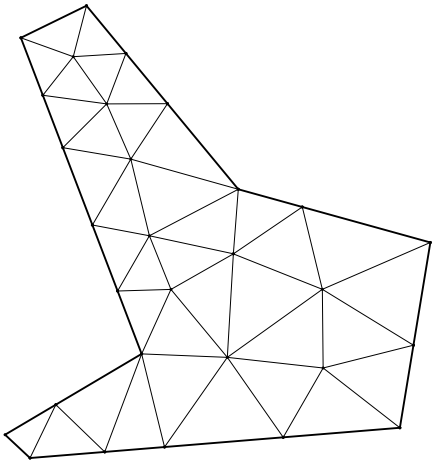
Si se dispone previamente de un algoritmo que genere mallados triangulares de elevada calidad, cabe plantearse su aprovechamiento para mallado cuadrangular, transformando los elementos triangulares en cuadrangulares.

Este posible camino abre otro criterio de clasificación: Denominamos métodos **indirectos** a aquellos que generan el mallado cuadrangular mediante la transformación de una triangulación previa. Por contra, cuando no se parte de ningún mallado previo, reciben el nombre de métodos **directos**.

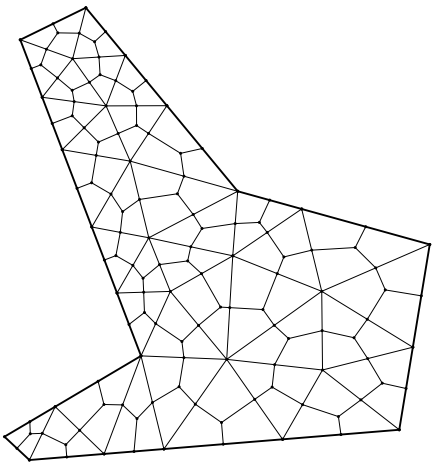
2.1.2.1. Métodos indirectos

Un camino trivial para convertir un mallado triangular en otro cuadrangular consiste en **subdividir** cada triángulo en tres cuadriláteros, mediante la adición de un nuevo nodo en el interior de cada triángulo y en el punto medio de cada lado del mallado (figura 3b).

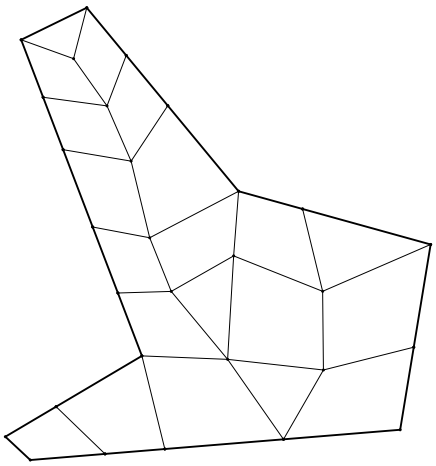
El método indirecto por subdivisión de triángulos tiene muy sencilla implementación, pero sin embargo posee importantes inconvenientes: Tiende a generar cuadriláteros de peor calidad que los triángulos originales, y además no respeta la densidad de mallado original (los cuadriláteros resultantes son más pequeños que los triángulos de partida y se incrementa el número de nodos tanto en el interior como en el perímetro).



(a) Mallado triangular original.



(b) Cuadriláteros por subdivisión.



(c) Cuadriláteros por agrupación.

Figura 3: Estrategias de mallado cuadrangular indirecto.

Otra modalidad de método indirecto es la **agrupación de triángulos** para constituir cuadriláteros. Este camino otorga mayores posibilidades de lograr elementos de mejor calidad que el método indirecto por subdivisión, pero a cambio tiene el inconveniente de no garantizar que el mallado resultante sea completamente cuadrangular. En general, al finalizar la agrupación, pueden quedar triángulos sobrantes (figura 3c).

Como esta última estrategia no garantiza que todos los elementos sean cuadriláteros, al resultado de estos algoritmos se le suele denominar "mallado cuadrangular-dominante" (*quad-dominant mesh* en inglés), dejando así constancia de que puede haber triángulos en el mallado.

Entre estos algoritmos cabe destacar los desarrollados por S. H. Lo [29], Johnston *et al* [24], así como Lee y Lo [28]. Todos ellos tratan de minimizar el número de triángulos sobrantes, ya sea mediante heurísticas o modificando localmente los triángulos originales en caso necesario. En [28] se garantiza que todos los elementos serán cuadriláteros si el número de lados del perímetro es par.

Posteriormente, Owen *et al* [44] propusieron otro algoritmo indirecto, conocido como *Q-Morph*. Consiste en volver a mallar localmente a lo largo de frentes de avance paralelos al perímetro, intercambiando aristas localmente y añadiendo nodos adicionales, con la finalidad de lograr que el mallado triangular original quede transformado en elementos cuadrangulares de la máxima calidad posible.

2.1.2.2. Métodos directos

Cuando no se dispone de un mallado triangular previo, el algoritmo debe generar los elementos cuadrangulares partiendo de cero. Los métodos que no parten de un mallado previo se denominan **directos**, y pueden a su vez clasificarse en los siguientes tipos:

2.1.2.2.1. Por descomposición

Los métodos de esta categoría operan mediante la descomposición del dominio en contornos más sencillos con el objetivo de generar finalmente elementos cuadrangulares.

Baehmann *et al* [1] proponen representar el dominio a mallar como una *quad-tree*, quedando descompuesto en cuadrantes de manera recursiva. La finura de la *quad-tree* depende de parámetros especificados por el usuario al definir el perímetro del dominio, o bien de la complejidad del mismo. Cada cuadrante final de este árbol puede ser interior al dominio, contener parte del perímetro, o bien ser exterior. Tras eliminar segmentos degenerados o conflictivos en la intersección del perímetro con los cuadrantes finales, se aplica a cada uno de ellos un patrón de mallado que transforma cada cuadrante en elementos triangulares o cuadrangulares (este algoritmo puede generar mallados triangulares o cuadrangulares, según la aplicación).

Talbert y Parkinson [55] descomponen el dominio recursivamente en regiones poligonales más sencillas. Aplicando a dichas regiones un catálogo de plantillas de mallado, se generan los elementos cuadrangulares deseados.

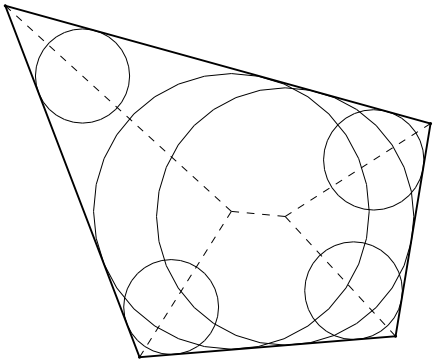


Figura 4: Método directo de descomposición por eje medio.

Autores como Chae [8] y Nowottny [39] han propuesto algunas mejoras a este algoritmo. Otra alternativa, generando la descomposición en polígonos convexos es la propuesta por Joe [23].

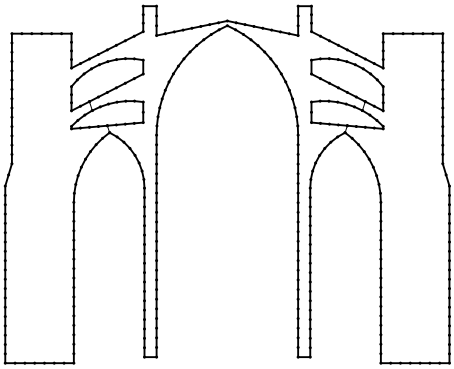
Tam y Armstrong [56] introdujeron la descomposición del dominio a través de su eje medio (figura 4). A las áreas resultantes tras esta descomposición, se les aplican de nuevo plantillas de mallado para generar la discretización.

Sarrate y Huerta [50] desarrollaron un algoritmo que cabe también clasificar como método directo por descomposición, si bien su planteamiento difiere de los anteriores en que el resultado de la descomposición es ya el mallado cuadrangular buscado. Los algoritmos que hemos mencionado en esta categoría aplican una descomposición en regiones sencillas, que posteriormente son malladas. Por contra, Sarrate y Huerta descomponen el dominio recursivamente hasta que todas las regiones resultantes son cuadriláteros (figura 5).

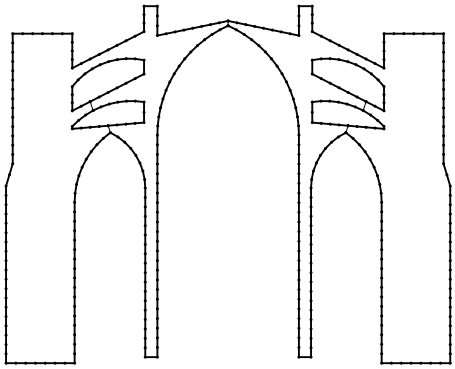
El algoritmo de Sarrate y Huerta realiza la descomposición de un contorno en dos mitades asignando una función de coste a todas las líneas candidatas a ser divisoras, y eligiendo aquella cuyo coste sea menor. Bastian y Li [2] proponen algunas modificaciones, si bien el algoritmo es el mismo.

2.1.2.2.2. Por frente de avance

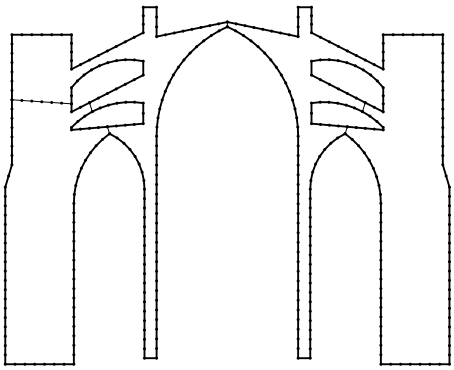
Otra categoría de métodos directos la constituyen los conocidos como frente de avance (*advancing front* en inglés). Consisten en ir mallando anillos paralelos al perímetro, hacia el interior del dominio. Cabe destacar el de Blacker y Stephenson [3], llamado *paving* por sus autores. El concepto es sencillo, pero la complejidad surge al tener que considerar muchos casos



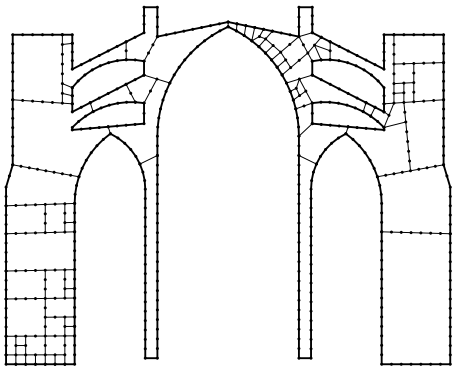
(a) Contorno inicial (agujeros ya unificados).



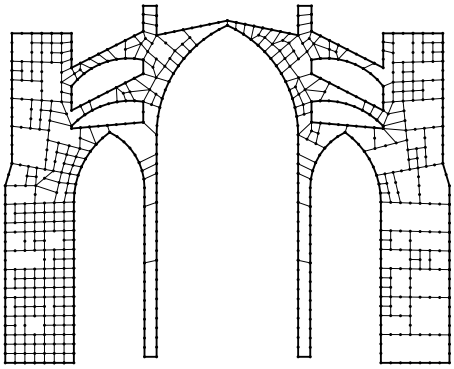
(b) Primera subdivisión (en la clave del arco central).



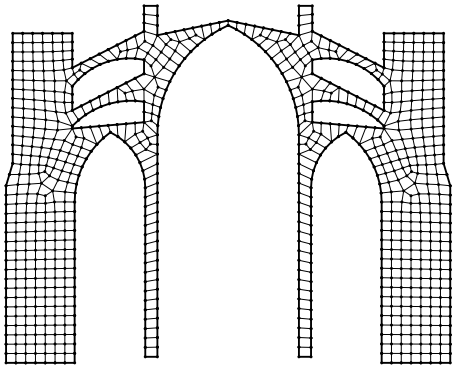
(c) Segunda subdivisión.



(d) Progreso de las subdivisiones.

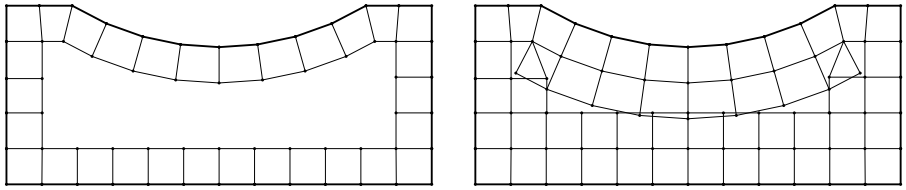


(e) Fase avanzada de subdivisión.



(f) Mallado finalizado y relajado (670 cuadriláteros).

Figura 5: Aplicación del algoritmo de Sarrate y Huerta [50] a la sección transversal de la Catedral de Palma de Mallorca por el segundo peripiaño (la geometría del contorno procede del estudio realizado por [32]).



(a) Mallado del primer anillo. (b) Mallado del segundo anillo.

Figura 6: El algoritmo *paving* de Blacker y Stephenson [3] requiere la consideración de un catálogo importante de casos particulares, como los producidos por las auto-intersecciones en el mallado del segundo anillo de este ejemplo.

particulares, debido a las auto-intersecciones producidas conforme los anillos avanzan hacia el interior (figura 6).

En cualquier caso, una vez solventadas con éxito dichas situaciones conflictivas, se trata de un algoritmo que logra de manera natural que los cuadriláteros estén orientados fielmente según las direcciones del perímetro.

2.2. Mallado en GPU

Las características y programabilidad de las *GPUs* actuales ha posibilitado su empleo para el mallado en aplicaciones de análisis de elementos finitos.

Los primeros avances al respecto trataron la generación de diagramas Voronoi en *GPU*. No eran malladores, pero aportaron una primera experiencia necesaria para los desarrollos posteriores.

Rong *et al* [48] fueron los primeros en implementar en *GPU* la triangulación Delaunay plana (no restringida), si bien se requería también la colaboración de la *CPU*. Plantean una construcción del diagrama Voronoi de manera discreta (en una imagen/textura con una resolución en *pixels* fijada por el usuario). Posteriormente se usa la *CPU* para ajustar las coordenadas desde el espacio discreto al espacio continuo. Los autores midieron una mejora de rendimiento de alrededor de un 53 % comparada con el software *Triangle* de Shewchuk [52], que opera exclusivamente en *CPU*.

Qi *et al* [45, 46] continuan el trabajo de [48], pero logran implementarlo completamente en *GPU*, y con la capacidad adicional de generar triangulaciones Delaunay restringidas[(27)]. Consiguen un incremento de rendimiento

[(27)] En las triangulaciones Delaunay restringidas se fuerza que algunas aristas formen parte del mallado.

de 10 a 30 veces el de *Triangle*. Otras implementaciones de la triangulación Delaunay en *GPU* son Navarro *et al* [36], y Kandasamy y König [25].

También se han desarrollado algoritmos para la triangulación 3D Delaunay en *GPU*, como Nanjappa [35] y Cao *et al* [6].

Todos estos trabajos están limitados a mallados triangulares, pero evidentemente el mallado cuadrangular debería de poder beneficiarse igualmente del aprovechamiento de *GPUs*. Para ello, comenzaremos estableciendo los criterios que lo faciliten.

2.3. Programación en GPUs

Tal y como se mencionaba en el apartado *Motivación y presentación*, los subsistemas precursores de las actuales *GPUs* eran placas con un elevado número de circuitos integrados.

Algunos de sus componentes se dedicaban a la gestión de *pixels* (dibujo de *pixels*, aceleración de operaciones *BITBLT*[28], gestion de texturas, *rasterización* de líneas y polígonos, etc...).

Otros componentes de estos aceleradores gráficos se dedicaban a proyectar coordenadas 3D en el espacio 2D de la pantalla, así como a calcular la iluminación de vértices según la normal a la superficie y el vector director de la fuente de luz. Problemas que se traducen en la múltiplicación de millones de vectores 3D ó 4D por matrices 4x4, es decir, operaciones aritméticas de coma flotante sobre grandes volúmenes de datos (figura 7).

Esta funcionalidad se programaba llamando a librerías gráficas como *OpenGL* [53], con una funcionalidad fija. Es posible emplear *OpenGL* para acelerar tareas no gráficas, utilizando por ejemplo las coordenadas X, Y, Z espaciales o los colores R, G, B de *pixels*, para almacenar otras magnitudes de diferente naturaleza, pero evidentemente no es trivial hacerlo, ni tampoco se dispone de la flexibilidad deseada.

[28] El acrónimo *BITBLT* procede del inglés *bit block transfer* y se refiere a operaciones sobre bloques de *pixels* en imágenes de mapa de bits, aplicando operadores de *Boole* (y operadores aritméticos en general en chips más potentes). En la década de 1980, la proliferación de interfaces gráficas de ventanas y los videojuegos 2D, propiciaron la necesidad de aliviar a la *CPU* del trabajo de mover rectángulos de *pixels* de una parte a otra de la pantalla, y se hicieron populares los chips aceleradores de operaciones *BITBLT*, tanto en ordenadores domésticos (VDP V9938 en ordenadores MSX2) como en estaciones gráficas de elevado coste (Silicon Graphics, entre otros).

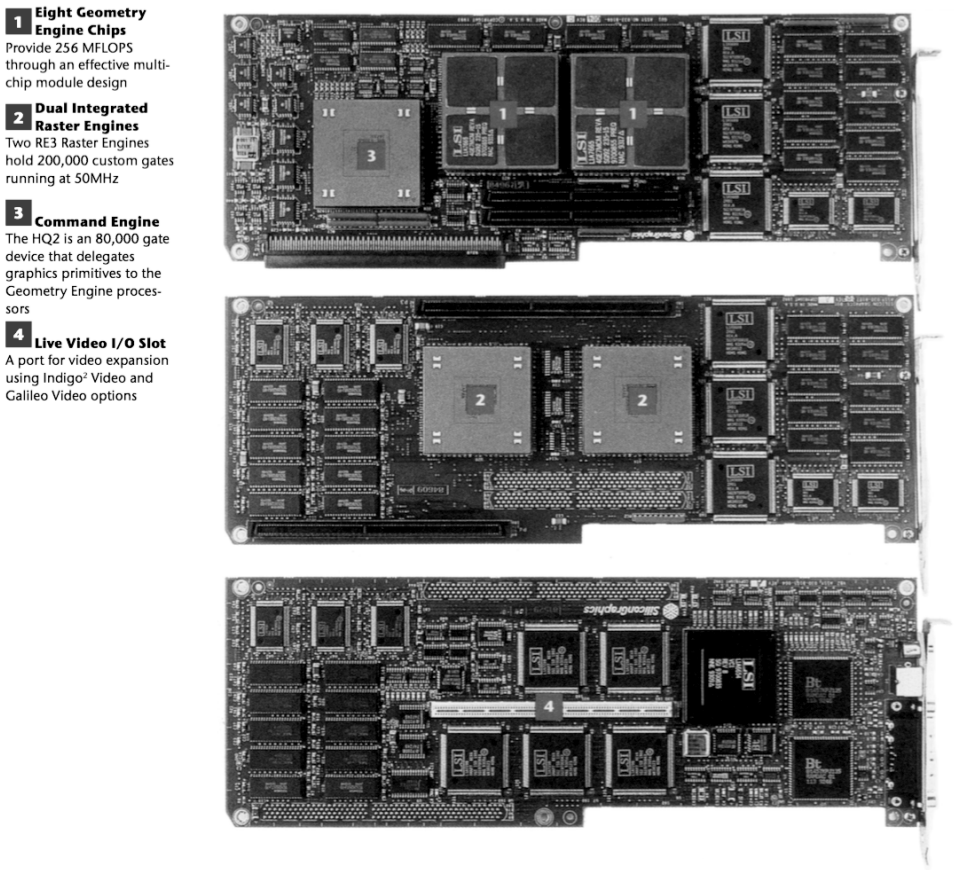


Figura 7: Subsistema gráfico *Extreme*, del año 1994, formado por tres placas para estaciones *Indigo2 (fuente: Silicon Graphics, Inc.). Nota: Los Geometry Engines (marcados con el número 1) son componentes especializados en cálculo masivo con coma flotante (rendimiento teórico total de 256 MFLOPS), mientras los Raster Engine (marcados con el número 2) están especializados en operaciones con pixels y números enteros.*

Algunos aceleradores se basaban en chips capaces en teoría de ser reprogramados para otras tareas (por ejemplo, los *Geometry Engine* del subsistema gráfico *Reality Engine* eran chips *Intel i860XP*, procesadores *RISC* que pueden operar como *CPU* de propósito general) pero los fabricantes de aceleradores gráficos no facilitaban las herramientas necesarias para tal reprogramación.

No obstante, algunas extensiones de *OpenGL* ayudaban a aprovechar la potencia de cálculo de los aceleradores gráficos para tareas no gráficas. Por ejemplo, las extensiones *Imaging* de *OpenGL* (capítulo 10, *"Imaging Extensions"*, de [26]) proporcionaban operaciones de convolución aceleradas en hardware (extensión *GL_EXT_convolution*), y también multiplicación masiva de vectores 3D y 4D por matrices 4x4 almacenando el resultado en la memoria gráfica (mediante la extensión *GL_SGI_color_matrix*), *mapeado* de valores numéricos mediante tablas (extensión *GL_SGI_color_table*), y obtención de histogramas (con la extensión *GL_EXT_histogram*).

Se trataba evidentemente de extensiones dirigidas inicialmente al proceso de imágenes, pero podían aplicarse a otras tareas de cálculo numérico, aprovechando así la potencia de cálculo de coma flotante de los subsistemas gráficos para aplicaciones no gráficas.

Posteriormente *OpenGL* incluyó la posibilidad de programar *shaders*[(29)] (pequeños programas escritos en lenguaje *GLSL*, de sintaxis casi análoga al lenguaje *C*, que permitían modificar los colores de los *pixels* y las coordenadas de vértices con una flexibilidad mucho mayor que la funcionalidad tradicional de *OpenGL*). Todavía no podía decirse que el hardware gráfico era completamente reprogramable a voluntad, pero claramente se marcaba el camino en esa dirección.

La evolución desde estos aceleradores gráficos hacia las *GPUs* ha producido notables cambios:

- La funcionalidad que antes se lograba mediante multitud de circuitos integrados, hoy se encuentra concentrada en un único chip, la *GPU*, conduciendo así a mayores rendimientos y mayor economía de producción.

- El deseo de programabilidad ha influido también en el diseño de *GPUs*,

[(29)] El concepto de *shader* proviene de *Pixar*. La especificación *Pixar RenderMan* [57], de 1989, define el empleo de *shaders* para dotar de una mayor flexibilidad al proceso de *renderizado*. Años más tarde, *OpenGL* reutilizó esta idea (hecha ya popular también en la mayoría de herramientas de diseño de gráficos 3D).

buscando que sea flexible y aprovechable en aplicaciones diversas.

- Las versiones recientes de la librería gráfica *OpenGL* poseen una programabilidad casi total, aunque sigue enfocada a gráficos.

- Se han desarrollado nuevas librerías (como *CUDA* y *OpenCL*) que permiten programar una *GPU* de manera completamente genérica, con lenguajes de alto nivel similares a *C* y *C++*.

Como ejemplo de diseño de una *GPU* actual, podemos observar la arquitectura *Pascal* de *NVIDIA* [41]. Su configuración máxima se encuentra en el chip *GP100*, destinado a tarjetas de cálculo científico de la gama *Tesla*.

Eliminando diversos grados de funcionalidad del chip *GP100*, se han obtenido diferentes *GPUs (GP102*, *GP104*, y *GP106)*, con un abanico de precios diverso, dirigidas a diferentes sectores, desde juegos a usuarios profesionales.

La arquitectura *Pascal* pone de manifiesto el deseo de flexibilidad y programabilidad que ha acompañado a la evolución de las *GPUs*, a través de las siguientes características:

- Núcleos de proceso de funcionalidad flexible: Las *GPUs* actuales se diseñan sobre un núcleo de proceso básico y programable que se repite de manera masiva en una arquitectura paralela. El chip *GP100* consta de 3584 núcleos *CUDA* de coma flotante de 32 bits, y 1792 de 64 bits.

- Aumento del tamaño de los cachés de memoria (ya introducido por *NVIDIA* en anteriores generaciones de *GPUs*)[30].

- Memoria unificada: La *GPU* soporta direccionamiento de 49 bits, de manera que puede direccionar toda la memoria del espacio de la *CPU*. Unido a un sistema de protección de páginas de memoria implementado en la *GPU*, permite gestionar la memoria de manera unificada entre *CPU* y *GPU*, haciendo no sólo la programación en *GPU* más sencilla,

[30] En las primeras *GPU* que permitían programación de propósito general, había una penalización muy importante de rendimiento cuando muchos núcleos de la *GPU* accedían a la misma posición de memoria. Esto no afectaba a tareas gráficas, pues era sencillo conseguir que cada núcleo accediese a diferentes vértices, *pixels*, etc..., pero al programar algoritmos arbitrarios en *GPU*, se puso de manifiesto que hay procedimientos que requieren que muchos núcleos accedan a los mismos datos a la vez, con la consiguiente penalización y disminución de rendimiento. El uso de cachés en la *GPU* soluciona este problema. En el chip *GP100*, el caché de segundo nivel se ha aumentado a 4 MB.

sino aumentando el rendimiento: sólo se realizan copias de páginas de memoria entre *CPU* y *GPU* cuando una de ellas intenta acceder a memoria que ha sido modificada por la otra. En caso contrario, acceden a memoria local directamente.

- *Compute preemption* (prioridad de computación): Un problema frecuente cuando se usan *GPUs* para cálculo intensivo es que el sistema puede dejar de responder, o hacerlo muy lentamente, debido a que la *GPU* esté muy solicitada y deje de atender a las tareas gráficas de la interfaz de ventanas del sistema operativo. La arquitectura *Pascal* soluciona este problema mediante cambios de contexto muy rápidos, a nivel de instrucción de programa, permitiendo ejecutar varias aplicaciones sin que el sistema deje de dar una respuesta interactiva.

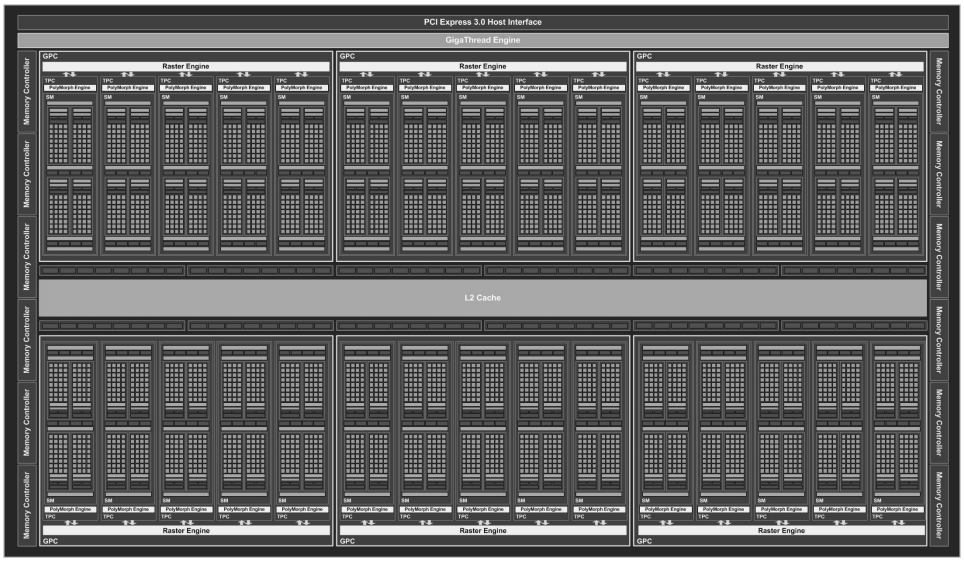
- Operaciones atómicas nativas en *GPU*: En implementaciones paralelas de algoritmos surge la necesidad de modificar datos de manera segura en memoria compartida. En *GPUs* antiguas esto requería bloquear la memoria para lograr comportamiento atómico, acarreando una grave penalización de rendimiento. Al tener ahora operaciones atómicas nativas, no es necesario bloquear la memoria, y el rendimiento paralelo de algoritmos es mayor.

Una variación interesante del chip *GP100* la constituye la *GPU GP102* (figura 8), empleada en la tarjeta *Titan X Pascal*, dirigida a un sector semiprofesional. La diferencia más notable se encuentra en el rendimiento de coma flotante de 64 bits. En el chip *GP100*, dicho rendimiento es la mitad del de 32 bits, mientras que en el *GP102* se reduce a 1/32 del de 32 bits. Esta es probablemente la razón que con mayor peso justifica la diferencia de precio entre ambas *GPUs*, así como el sector a que van dirigidas.

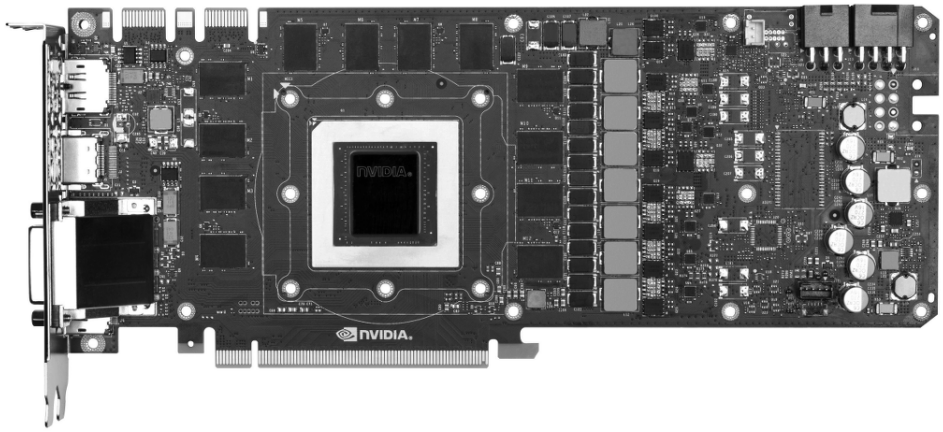
Las *GPU* de la arquitectura *Pascal* se componen de unos bloques comunes, cuyo tamaño y número varía según cada chip. El bloque mayor en la jerarquía se denomina *Graphics Processing Cluster (GPC)*, y es en sí mismo una *GPU*. La *GP102* tiene 6 *GPC*, así que podemos decir que está formada por 6 *GPU* que operan en paralelo. En la *GP102*, cada *GPC* se compone de 5 *Streaming Multiprocessors (SMs)* y de un *Raster Engine*. A su vez, cada *SM* está formado por 128 núcleos de proceso CUDA.

Teniendo en cuenta que la *GP102* posee 28 *SMs* operativos, el número total de núcleos CUDA es de 3584 (figura 8).

Se ha mencionado que el diseño de *GPUs* ha evolucionado hacia una mayor flexibilidad de programación, pero eso no significa que sean ahora



(a) Diagrama esquemático de la *GPU GP102*, con sus *6 GPC, 5 SM por GPC, y 128 núcleos CUDA por cada SM (de los 30 SM que aparecen en el diseño, son funcionales 28, dejando 2 no operativos por criterio de fabricación —en total, los 28 SM contienen 3584 núcleos de coma flotante de 32 bits, con un rendimiento teórico de 11 TFLOPS). El chip contiene 12000 millones de transistores.*



(b) Tarjeta *Titan X*. El chip central con el logotipo *NVIDIA* es la *GPU GP102*.

Figura 8: Diseño de una *GPU* moderna: *NVIDIA Titan X (Pascal)*, año 2016. A diferencia de los subsistemas gráficos de antaño, ahora toda la funcionalidad se encuentra concentrada en el único chip de la *GPU*. Los doce chips que rodean la *GPU* son los 12 GB de memoria que posee la tarjeta.

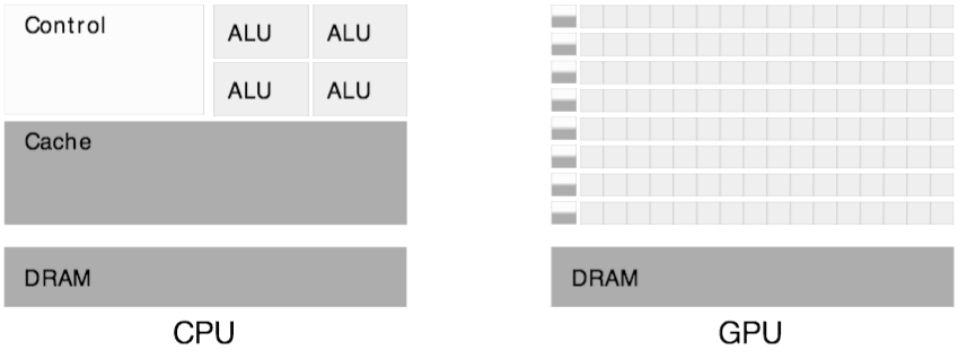


Figura 9: Una *GPU* dedica a operaciones aritméticas un número de transistores mucho mayor que una *CPU* (cada pequeño cuadrado de la figura derecha representa un núcleo de cálculo aritmético, en comparación con la menor área ALU de la *CPU*), pero tiene menos recursos dedicados a caché y a control de flujo de programas *(fuente: NVIDIA [40]).*

análogas a una *CPU*. Las *GPUs* dedican una cantidad mayor de transistores a operaciones aritméticas, gracias a lo cual logran su elevada potencia de cálculo (figura 9).

Sin embargo, tienen menos recursos para control del programa, aspecto que puede acarrear un impacto negativo de rendimiento al programarlas, si no se tiene en cuenta.

A este respecto, las *GPUs* son máquinas *SIMD* (*Single Instruction, Multiple Data*), donde una instrucción se aplica a varios datos en paralelo. Parece que contamos con varios hilos de ejecución en paralelo, pero sin embargo comparten un único contador de programa (es una instrucción única en realidad). Eso significa que, cuando tenemos sentencias condicionales, si se produce una ramificación en la ejecución del programa afectando sólo a unos datos, habrá una penalización en el rendimiento debido a que el resto de hilos se quedarán en pausa hasta que finalice la ramificación.

Se puede extraer la conclusión de que escribir programas en una *GPU* es complejo y difícil, pero no es menos cierto que con el paso del tiempo se han mejorado sus herramientas de programación, tratando de conseguir que programarlas no entrañe dificultad mayor que hacerlo para una *CPU*. De hecho, actualmente podemos escribir programas para *GPU* en lenguajes de alto nivel, disponiendo además de completos depuradores que nos indican las causas de rendimientos inferiores a los esperados si ello sucede.

Son dos las plataformas que gozan de mayor extensión de uso en la actualidad: *OpenCL* [51] y *CUDA* [40]. *OpenCL* es un estándar soportado por varios fabricantes. Prácticamente todos los fabricantes actuales de *GPUs* y

CPUs proporcionan una implementación *OpenCL* para acceder a su hardware. *CUDA* es una plataforma desarrollada por *NVIDIA*, y requiere una *GPU NVIDIA*.

Tanto *OpenCL* como *CUDA* permiten programar la *GPU* en una sintaxis muy similar al lenguaje *C* (de hecho, la implementación realizada en esta Tesis Doctoral utiliza unos mismos archivos de código fuente que, mediante la adecuada elección de directivas del preprocesador de *C*, pueden ser procesados tanto por un compilador estándar de *C*, como por una de estas plataformas de programación de *GPU*).

Usando por tanto un lenguaje de alto nivel, y prestando atención a las características del diseño de una *GPU* que hemos visto, se pueden programar muy diversos algoritmos en *GPU*.

2.4. Programación en CPUs de múltiples núcleos

Como aprovechar los recursos actualmente disponibles implica también el empleo de *CPUs* multi-núcleo, necesitamos elegir la manera de programarlas. Las *CPU* más extendidas no son máquinas *SIMD*, por lo que no cabe considerar los condicionantes expuestos en el apartado anterior sobre programación de *GPUs*.

Existen multitud de librerías de programación, y funciones del Sistema Operativo, que hacen posible escribir programas en paralelo, y lograr su ejecución en varios núcleos.

No obstante, se da la circunstancia de que *OpenCL* permite ejecutar programas no sólo en *GPU*, sino también en *CPU*, y aprovechando todos sus núcleos (en realidad también permite ejecutar un programa paralelo de manera híbrida, en *GPU* y *CPU* a la vez).

Esta flexibilidad de *OpenCL* posibilita una sencilla comparativa de rendimientos de algoritmos en *GPU* y *CPU*, al tiempo que simplifica notablemente el trabajo al emplear la misma plataforma para programar ambos dispositivos.

Si a ello le unimos la disponibilidad de *OpenCL* en diferentes fabricantes, se entiende su elección en la implementación aquí realizada de mallado cuadrangular.

Cabe decir, sin embargo, que *NVIDIA* tiende a implementar *OpenCL*

como una capa sobre *CUDA*. Por ello, si se desea un rendimiento máximo en una *GPU NVIDIA*, sería recomendable el empleo de *CUDA*, siendo conscientes de que dicha plataforma no está disponible en otros fabricantes.

2.5. Acceso a OpenCL

OpenCL se distribuye como una biblioteca de tiempo de ejecución, proporcionada por cada fabricante de *GPU* y generalmente incluida en sus *drivers* gráficos. Para desarrollar un programa que acceda a *OpenCL* suele ser necesario instalar un entorno de desarrollo de *OpenCL (SDK)*, que es también facilitado por cada fabricante y para cada sistema operativo. Es decir, el desarrollador del programa necesita la biblioteca de tiempo de ejecución, pero también el *SDK*. El usuario final sólo necesita la biblioteca para poder ejecutar el programa.

Sin embargo, al emplear *OpenCL* en el desarrollo de una aplicación, aparecen pronto las siguientes cuestiones:

- Al compilar un ejecutable con el *SDK* de *OpenCL*, se enlazan las funciones de *OpenCL* a las que el programa accede. Si el sistema del usuario no posee *OpenCL*, lo más probable es que el programa genere un error de enlace dinámico al arrancar, sin ni siquiera poder avisar al usuario de la necesidad de instalar una implementación *OpenCL*, o incluso sin poder seguir ejecutándose con una implementación alternativa que no requiera *OpenCL*.

- A lo largo de los años se han ido publicando varias versiones de *OpenCL*. En concreto, hasta la fecha han visto la luz las versiones 1.0, 1.1, 1.2, 2.0, 2.1, y 2.2. Cada una de estas versiones ha ido añadiendo funciones nuevas. Esto complica aún más la cuestión anterior, ya que un programa desarrollado para *OpenCL* 2.2 podría generar un error de ejecución al arrancar si el usuario tiene una versión anterior de *OpenCL*, sin que el programa pueda avisar al usuario.

Es cierto que algunos sistemas operativos soportan el enlace dinámico diferido, que permitiría reducir el impacto de estas cuestiones. Pero se ha estimado más apropiado desarrollar una biblioteca ex profeso que proporcione una solución satisfactoria en cualquier sistema operativo, y que haga más cómodo el acceso a *OpenCL*.

Así pues, durante la implementación del mallador, se ha desarrollado conjuntamente la biblioteca *"AccCL"*. Permite que un programa acceda a

OpenCL pero pueda ejecutarse igualmente si el sistema del usuario no dispone de *OpenCL*. Y también permite que un programa pueda ejecutarse en una versión de *OpenCL* anterior que para la que ha sido desarrollado. Proporciona al programa unos mecanismos seguros para detectar dichas situaciones y poder obrar en consecuencia.

Al mismo tiempo, *AccCL* incluye todas las definiciones y prototipos de funciones necesarias para compilar con *OpenCL*, por lo que hace innecesario instalar el *SDK* (de hecho todo el código de mallado se ha escrito y compilado sin instalar un *SDK OpenCL*).

3. Resultados

3.1. Criterios para la elección del algoritmo de mallado

A partir de la clasificación de algoritmos de mallado cuadrangular realizada, y teniendo en cuenta los aspectos mencionados en los apartados anteriores, cabe pronunciarse sobre las características deseables en un algoritmo para que pueda implementarse de manera óptima y eficiente en *CPUs* y *GPUs* actuales.

Dado que no partimos de la premisa de que el dominio vaya a ser asimilable a una figura regular, ni tampoco imponemos que la densidad de mallado sea constante, vamos a centrarnos en mallado **no estructurado**, pues es el único con flexibilidad suficiente para adaptarse a geometrías complejas. No obstante, cuando el dominio sea razonablemente regular y la densidad de mallado sea constante, sería ventajoso que el algoritmo crease un mallado estructurado en aras de una mayor calidad de forma en los cuadriláteros.

No disponemos de un mallado triangular previo. Además, los métodos indirectos no garantizan mayor calidad de elementos que los directos, a la vez que su eficiencia depende de la del procedimiento de triangulación empleado. Por ello, nos decantamos por elegir un **método directo** admitiendo que, una vez tengamos encima de la mesa todos los criterios de decisión, se podría revisar.

Con anterioridad hemos visto que los avances en *CPUs* y *GPUs* actuales van en la dirección de aumentar el número de núcleos de proceso. Por tanto, si pretendemos una implementación óptima en hardware actual, el algoritmo debe ser paralelizable.

Por la misma razón (y también por las penalizaciones de rendimiento que un exceso de sentencias condicionales ocasionan al programar *GPUs*), es preferible optar por algoritmos que no requieran considerar casos particulares durante el progreso del mallado o que, al menos, los casos particulares sean pocos, y su tratamiento sencillo. Conforme aumenta el número de casos particulares a tratar, mayor es la complejidad de la implementación, y más se dificulta que la ejecución en paralelo sea eficiente.

Algunos requisitos que exigimos han quedado implícitos porque prácticamente todos los métodos de mallado mencionados los satisfacen. Sin embargo, es conveniente hacerlos explícitos: Deseamos que los cuadriláteros generados por el mallador tengan una calidad adecuada para su aplicación

en análisis de elementos finitos. Además, el mallado ha de ser automático, sin la intervención del usuario, y no debe existir riesgo de que la discretización producida sea degenerada.

A modo de resumen, las características que buscamos en el algoritmo son las siguientes:

- Debe generar mallados **no estructurados**, pero es deseable que tienda a crear un mallado estructurado cuando el dominio y la densidad de mallado lo permitan.
- En principio preferimos un **método directo**.
- Debe de poderse paralelizar de manera óptima.
- Debe quedar libre de la consideración de casos particulares numerosos y complejos.
- Los elementos cuadrangulares resultantes deben tener una calidad óptima para un análisis de elementos finitos.
- El usuario no debe intervenir en el proceso de mallado.
- No debe existir posibilidad de que ocurran situaciones anómalas que conduzcan a una geometría degenerada.
- Su implementación debe ser muy sencilla, para poder velar adecuada y eficientemente por todos estos requisitos.

Estos condicionantes nos invitan a descartar los métodos basados en frente de avance, puesto que el tratamiento de los casos particulares que surgen es complejo, como ya hemos visto.

Por contra, los métodos de descomposición sí parecen responder mejor a los criterios exigidos. Para empezar, la descomposición genera un conjunto de regiones dividiendo el dominio inicial, lo que facilitará una implementación en paralelo óptima.

3.2. Algoritmos y procedimientos propuestos

De entre los algoritmos de mallado cuadrangular no estructurado disponibles en la actualidad, y teniendo en consideración los criterios que se acaban de exponer, se ha optado por implementar el algoritmo de Sarrate y Huerta [50] por las siguientes razones:

- No necesita resolver encuentros geométricos complicados como los que suelen aparecer en algoritmos de tipo frente de avance (como el "paving"), y que dificultarían la paralelización.

- No se basa en una triangulación previa (pertenece al grupo que antes hemos llamado "métodos directos", por lo que no requiere haber hecho un mallado triangular inicial).

- El mallado se genera por descomposición sucesiva del contorno inicial. La búsqueda de la mejor línea candidata de subdivisión en cada fase del mallado sugiere una sencilla paralelización del problema.

- La implementación es relativamente sencilla porque no hay casos particulares que requieran un tratamiento especial (la única excepción aparece al subdividir contornos de 6 lados, que en algunos casos requieren un tratamiento especial, pero los beneficios de la paralelización se observan en contornos más grandes, por lo que los de 6 lados pueden tratarse de manera más eficiente en código secuencial).

No obstante, el algoritmo de Sarrate y Huerta tiene también algunas características que a priori podrían dificultar su paralelización en *GPU*:

1. La elección de la mejor línea de subdivisión se realiza mediante una función de coste cuyo valor depende de todo el contorno.

2. La evaluación de la función de coste necesita medir áreas y comprobar la intersección entre segmentos de recta y polígonos —ambos problemas requieren emplear sentencias condicionales y bucles.

La primera dificultad implica que todos los procesos paralelos necesitan acceso a la geometría completa del contorno. Los núcleos de las *GPU* actuales tienen memorias rápidas (locales) de tamaño limitado (en general de un tamaño más pequeño que los datos de un contorno que pueda beneficiarse de paralelización). La memoria global de las *GPU* suele ser grande (actualmente en el orden de varios GB), pero es mucho más lenta que las memorias locales

de cada núcleo de proceso. Por ello, la imposibilidad de dividir los datos del contorno para que quepa en la memoria local de cada núcleo, conlleva que va a existir una penalización de rendimiento en el acceso a memoria si se implementa en *GPU*. De todos modos, recordemos que los últimos diseños de *GPU* han introducido cachés de memoria más eficientes que en el pasado, lo que puede aminorar esta penalización.

La segunda dificultad significa que el código de evaluación de la función de coste va a tener ramificaciones, tanto por las sentencias condicionales, como también por los bucles. Pero hemos dicho que los núcleos de proceso de las *GPUs* suelen ser de tipo *SIMD*, cuyo paralelismo es sólo a nivel de datos: cuando un núcleo de tipo *SIMD* de una *GPU* llega a una ramificación del código, sólamente puede ejecutar uno de los dos caminos. En ese punto, una parte de los hilos se seguiran ejecutando, pero el resto quedará en pausa hasta que acabe la ramificación de código de los otros hilos. Como conclusión, de nuevo esta dificultad va a acarrear una penalización de rendimiento en *GPU* (en cualquier caso, las *GPU* suelen tener varios núcleos *SIMD* independientes, y cada uno de ellos sí que puede ejecutar instrucciones diferentes en paralelo).

Sin embargo, se trata de dificultades que pueden salvarse si las tenemos en cuenta en la implementación en *GPU*. Además, no afectan a la paralelización mediante *CPU* de múltiples núcleos, pues ni tienen memorias locales reducidas, ni basan su paralelización en *SIMD* (aunque algunas tienen extensiones *SIMD* para multimedia), sino en *MIMD (“multiple instruction, multiple data”)* que nos permiten ejecutar varias ramificaciones de código de manera simultánea.

Por todo ello, y como conclusión de la elección del algoritmo, podemos decir que:

- El algoritmo va a poderse paralelizar de manera sencilla (la evaluación de la función de coste de una línea es una tarea claramente independiente y separable, de paralelización trivial).

- Su rendimiento con *GPU* podría verse penalizado (aunque dependerá mucho de la arquitectura concreta de cada *GPU*: tamaño de memorias locales, número de núcleos *SIMD* independientes).

- Su rendimiento con *CPUs* multi-núcleo puede ser muy bueno, y se espera que escale muy bien con el número de núcleos.

3.2.1. Descripción del algoritmo de mallado elegido

Una descripción exhaustiva del algoritmo puede consultarse en [50]. No obstante, de manera resumida se exponen a continuación los puntos más significativos.

3.2.1.1. Requisitos de los datos de entrada

Los datos de entrada consisten en una definición del dominio, así como el tamaño deseado de los elementos. El dominio debe cumplir los siguientes requisitos:

- Es un contorno poligonal plano y cerrado, definido por una lista ordenada de vértices que se conectan entre sí mediante segmentos rectos.

- No puede haber intersecciones entre los segmentos rectos que constituyen el contorno (excepto en los vértices, naturalmente).

- Si el dominio tiene agujeros, se definirán mediante costuras formadas por segmentos solapados que comuniquen el contorno exterior con cada agujero. Es decir, finalmente el dominio consiste en la definición de un único contorno poligonal.

- Los vértices del contorno se especifican en orden antihorario (éste no es un requisito realmente imprescindible, pero permite incrementar el rendimiento de algunos cálculos, y por ello se ha tomado la decisión de exigirlo como requisito).

- Si el dominio define varias áreas separadas pero que se tocan en algunos vértices comunes, es posible describir el dominio mediante un contorno diferente para cada área (y aplicar el algoritmo a cada uno de los contornos por separado), o bien definirlas mediante un único contorno, siempre y cuando en los vértices de conexión no se modifique el orden antihorario de cada una de las áreas: todas ellas tienen que estar definidas de manera antihoraria (figura 10).

Por otra parte, es también necesario definir el tamaño deseado de los elementos del mallado. El algoritmo permite generar mallados cuya densidad sea variable a lo largo del dominio. Sarrate y Huerta sugieren emplear una función que proporciona el valor del tamaño de los elementos en cada punto

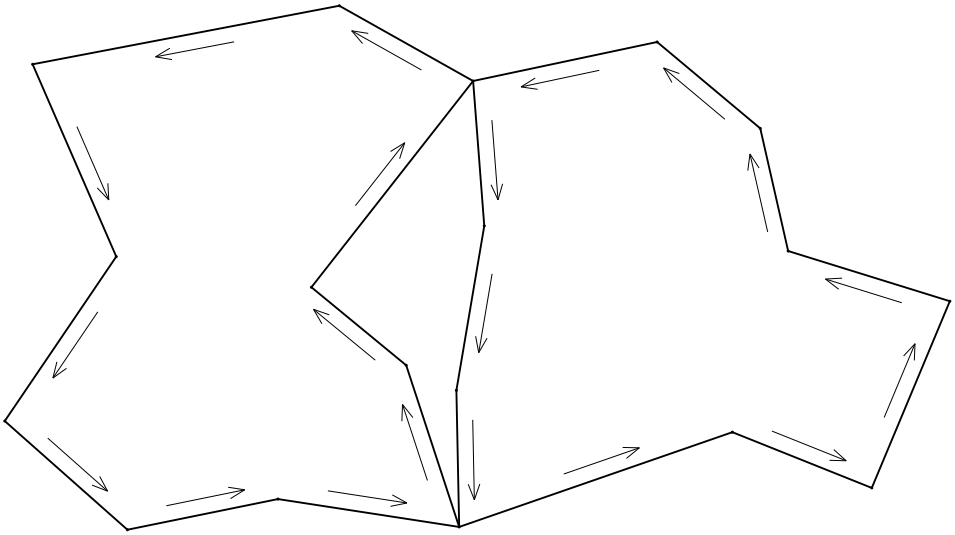


Figura 10: Dos áreas con vértices comunes. Pueden especificarse como dos contornos, o como un único contorno, pero siempre definidos en sentido antihorario.

del dominio (dicha función podría definirse mediante una imagen en la que el valor de cada *pixel* indique el tamaño deseado de elemento para cada zona).

En la implementación realizada en la presente Tesis Doctoral se ha optado por especificar el tamaño de los elementos de otra manera: adjudicando a cada vértice del contorno inicial una tercera "coordenada" que define precisamente el tamaño de los elementos que estén conectados con el vértice. Se ha elegido este camino porque es habitual que, en el momento de definir el contorno, el usuario conozca qué densidad de mallado desea en cada vértice. Asignando a cada vértice su densidad se agiliza el trabajo de preparar los datos de entrada, al no ser necesario definir una función o imagen de "densidades de mallado".

Sin embargo, es cierto que el camino elegido no contempla la situación de poder desear una variación en la densidad de mallado en una zona en que no haya ningún vértice. Siempre que se desea variar la densidad de mallado en una zona concreta, hay que situar al menos un vértice en dicha zona. En cualquier caso, se trata de una decisión en la implementación del algoritmo, si bien se podría recuperar la propuesta inicial de la "función de densidad de mallado" de Sarrate y Huerta.

Como se ha dicho, es posible aplicar el algoritmo a contornos con agujeros si se realiza el preproceso de conectar los agujeros con el perímetro exterior mediante aristas dobles, unificando así el perímetro y los agujeros como un único contorno. En la implementación realizada, se ha programado

este preproceso de manera trivial, detectando qué contornos están incluidos dentro de otros, y conectándolos por los vértices que estén más cerca. Los resultados de la unificación de agujeros obtenidos de esta manera han sido satisfactorios, como puede observarse en la figura 11, así como en el resto de figuras que poseen agujeros.

3.2.1.2. Operación general del algoritmo

El algoritmo consiste en dividir el contorno en dos mitades, por la "mejor" línea candidata posible, de manera que al aplicar sucesivamente el mismo procedimiento a cada mitad resultante, se llegue finalmente a un mallado cuadrangular con la calidad deseada.

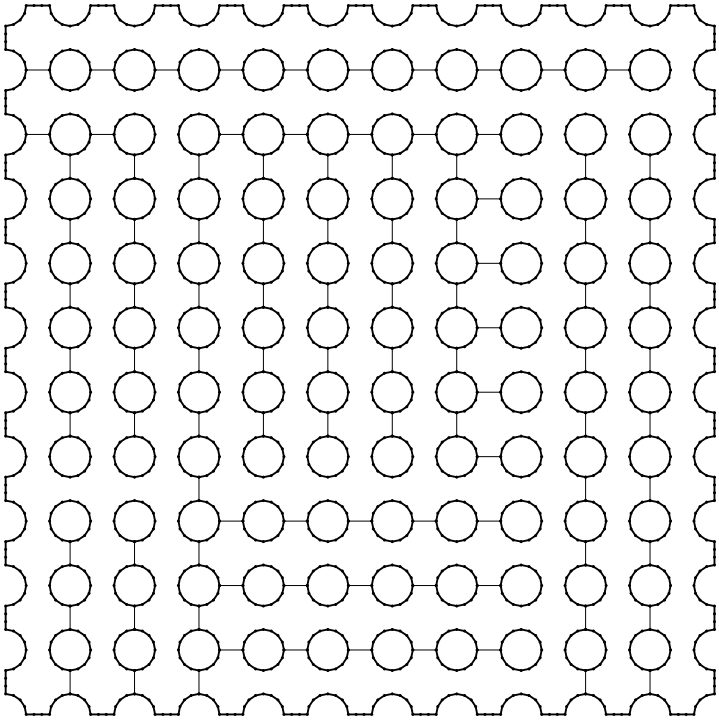
Es por tanto un método sencillo, ya que se trata de un mismo paso que aplicamos siempre de la misma manera a cada nueva subdivisión que se va generando. Al llegar a contornos de 6 vértices puede ser necesario variar el criterio de subdivisión, pero esto se explicará más adelante.

Antes de hacer la búsqueda de la mejor línea de división, es necesario realizar estos pasos previos:

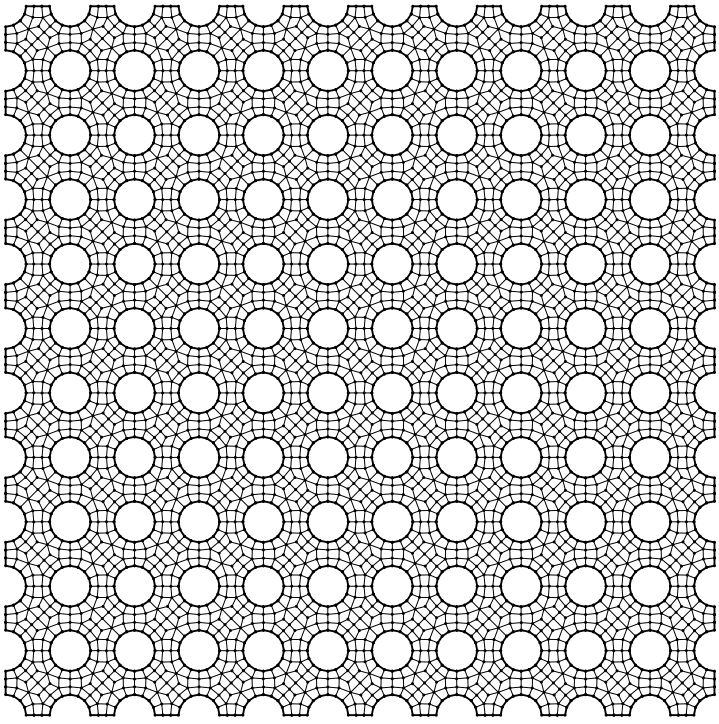
- Si en el contorno inicial hay lados cuya longitud es mayor que el tamaño deseado de elementos, los subdividimos insertando vértices intermedios hasta conseguir que todos los segmentos del contorno cumplan con el tamaño deseado de elementos.

- Como cada lado une dos vértices cuya densidad de mallado puede ser diferente, se hace necesario poder insertar vértices intermedios no equidistantes, sino con una gradación de la distancia entre ellos, de manera que el primer y el último segmento del lado tengan el tamaño especificado en el vértice al que se unen. La implementación de esta "gradación de la distancia" se explicará más adelante.

- Por último, para tener garantía de que el contorno admita una subdivisión en cuadriláteros, se exige que el número total de vértices del contorno sea par. Esto puede obligarnos a modificar ligeramente la densidad del mallado en algún lado. Por ello, si el contorno tiene un número impar de vértices, elegimos el lado de mayor longitud y le obligamos a tener un punto intermedio más de los que estrictamente necesitaría. Al ser el lado de mayor longitud, el error cometido en el tamaño de los elementos será el menor posible.



(a) Contorno inicial (agujeros ya unificados).



(b) Mallado (3872 cuadriláteros).

Figura 11: Panel perforado (número arbitrario de agujeros).

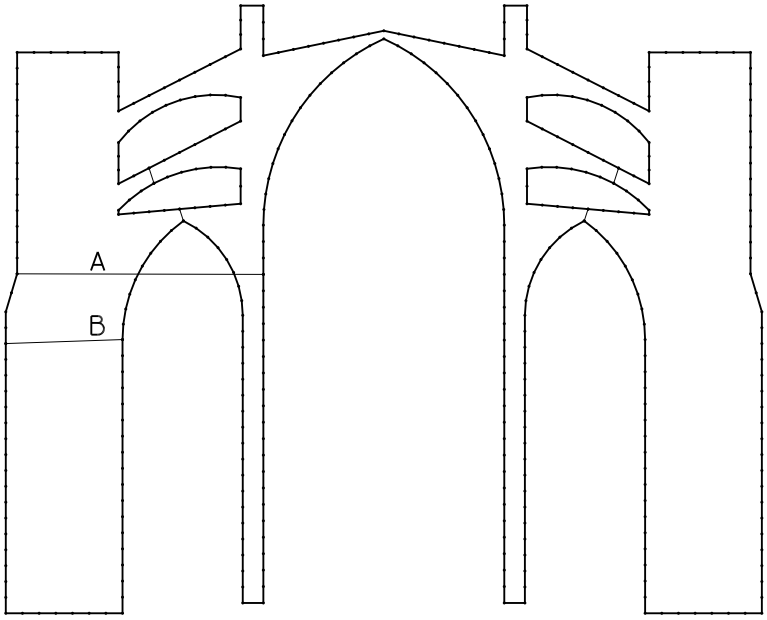


Figura 12: La línea A no es candidata para subdivisión, por tener intersección con el contorno. La línea B sí que es candidata.

Realizados estos pasos previos, estamos en condiciones de buscar la mejor línea de subdivisión.

Para dividir el contorno en dos mitades son líneas candidatas las que cumplen los siguientes requisitos (ver figura 12):

1. Conecta dos vértices del contorno.

2. No posee intersección con el contorno (excepto en sus vértices inicial y final).

3. Es interior al contorno.

4. Divide al contorno en dos partes con área no nula (esto implica que la línea no puede formar parte del contorno, pues en tal caso una de las partes tendría área nula).

Cada línea candidata tiene asociado un coste, que se evalúa mediante una función de coste. Se elige como mejor línea aquella que tiene un menor valor de la función de coste.

Una vez encontrada la mejor línea, le añadimos vértices intermedios si es

necesario para cumplir con la densidad de mallado (procedimiento análogo al paso previo que hemos realizado con todos los lados del contorno inicial), con la posibilidad de tener que modificar el número de vértices estrictamente necesario para lograr que los nuevos contornos tengan número par de vértices.

Y con ello generamos dos nuevos contornos, borramos el contorno que acabamos de subdividir, y aplicamos el mismo procedimiento a los nuevos contornos. Este proceso continúa hasta que todos los contornos son cuadriláteros.

3.2.1.3. Evaluación de la función de coste

La función de coste de cada línea candidata se evalúa de la siguiente manera, premiando aquellas candidatas que se aproximan a la bisectriz en el vértice correspondiente, generan una malla estructurada, unen dos puntos que estén lo más cerca posible, y dividen al contorno en dos mitades de similar área. Una explicación más pormenorizada de cada término puede encontrarse en [50].

En la expresión de la función de coste, que se muestra a continuación, el primer sumando premia las candidatas que se aproximan a la bisectriz, el segundo las que generan una malla estructurada, el tercero las que unen vértices cercanos, y el cuarto las que dividen el contorno en áreas lo más simétricas posible:

$$\text{Función de coste} = 0.52\phi + 0.17\sigma + 0.17\ell + 0.14\alpha \tag{1}$$

donde:

$$\phi = \begin{cases} 1 & \text{si } (\alpha_1+\alpha_2) < \dfrac{\pi}{2} \quad \text{y} \quad (\alpha_3+\alpha_4) < \dfrac{\pi}{2} \\ \mu(\zeta_1,\zeta_2) & \text{si } \dfrac{\pi}{2} \leqslant (\alpha_1+\alpha_2) \leqslant \dfrac{2\pi}{3} \\ & \quad \text{ó} \quad \dfrac{\pi}{2} \leqslant (\alpha_3+\alpha_4) \leqslant \dfrac{2\pi}{3} \\ \psi(\alpha_1+\alpha_2,\alpha_3+\alpha_4) & \text{si } (\alpha_1+\alpha_2) > \dfrac{2\pi}{3} \quad \text{y} \quad (\alpha_3+\alpha_4) > \dfrac{2\pi}{3} \end{cases} \tag{2}$$

$$\psi(\alpha_1+\alpha_2,\alpha_3+\alpha_4) = \frac{|\alpha_1-\alpha_2|+|\alpha_3-\alpha_4|}{(\alpha_1+\alpha_2)+(\alpha_3+\alpha_4)} \tag{3}$$

$$\mu(\zeta_1,\zeta_2) = (1-\zeta_1\zeta_2)+\zeta_1\zeta_2\psi(\alpha_1+\alpha_2,\alpha_3+\alpha_4) \tag{4}$$

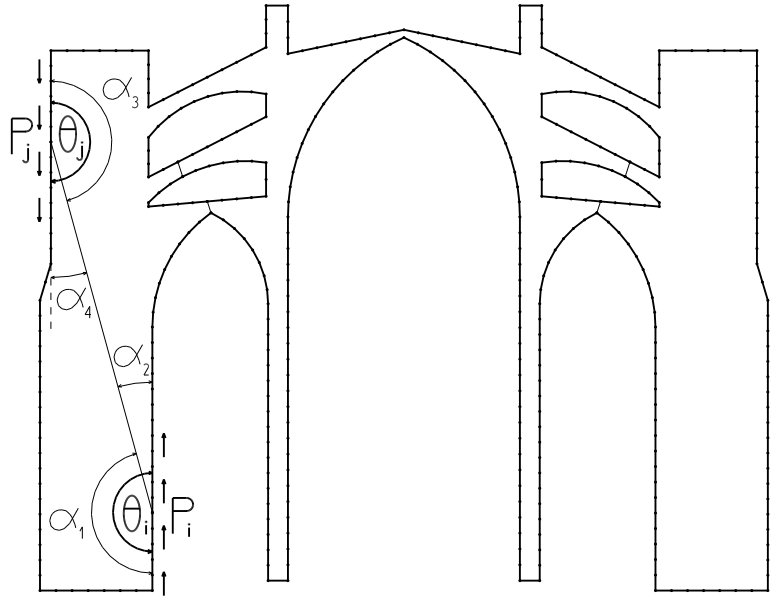


Figura 13: Ángulos para la función de coste de una línea candidata $P_i - P_j$. Las flechas indican el sentido antihorario del contorno. Se trata de un único contorno (los huecos se encuentran unificados con el perímetro exterior a través de lados dobles, como se ha indicado con anterioridad).

$$\zeta_1 = \begin{cases} \dfrac{(\alpha_1 + \alpha_2) - \frac{\pi}{2}}{\frac{\pi}{6}} & \text{si } \dfrac{\pi}{2} \leqslant (\alpha_1 + \alpha_2) \leqslant \dfrac{2\pi}{3} \\ 1 & \text{en otro caso} \end{cases} \tag{5}$$

$$\zeta_2 = \begin{cases} \dfrac{(\alpha_3 + \alpha_4) - \frac{\pi}{2}}{\frac{\pi}{6}} & \text{si } \dfrac{\pi}{2} \leqslant (\alpha_3 + \alpha_4) \leqslant \dfrac{2\pi}{3} \\ 1 & \text{en otro caso} \end{cases} \tag{6}$$

$$\sigma = \frac{\sigma_i + \sigma_j}{200} \tag{7}$$

$$l_{char} = \sqrt{(x_{max} - x_{min})^2 + (y_{max} - y_{min})^2} \tag{8}$$

$$\ell = \frac{l_{ij}}{l_{char}} \tag{9}$$

siendo l_{ij} la longitud de la línea candidata y l_{char} la longitud de la diagonal del rectángulo envolvente del dominio.

$$\alpha = \frac{|a_2 - a_1|}{a_2 + a_1} \tag{10}$$

donde a_1 y a_2 son las áreas de las dos mitades en que la línea candidata divide al contorno.

	σ_k
3	5.6
4	4.0
5	36.0
6	60.0
Otros	80.0

Tabla 3: Valores de σ_i y σ_j en vértices interiores (tómese $k = \{i, j\}$) según el número de elementos comunes (cuatro elementos comunes indicarían una malla localmente estructurada).

	σ_k
$0 \leqslant \theta_k < \frac{2\pi}{3}$	100.0
$\frac{2\pi}{3} \leqslant \theta_k \leqslant 2\pi$	0.0

Tabla 4: Valores de σ_i y σ_j (tómese $k = \{i, j\}$) en el perímetro inicial.

	σ_k
$0 \leqslant \theta_k < \frac{2\pi}{3}$	80.0
$\frac{2\pi}{3} \leqslant \theta_k \leqslant 2\pi$	0.0

Tabla 5: Valores de σ_i y σ_j (tómese $k = \{i, j\}$) en el perímetro de subdominios (generados al subdividir el dominio inicial u otros subdominios).

Con todas las expresiones anteriores queda definido el valor de la función de coste. Sarrate y Huerta incluyen otro término adicional que sólo es usado al dividir contornos de 6 lados, y es multiplicado por cero en contornos con número de lados diferente de 6. Dado que la división de los contornos de 6 lados vamos a realizarla siguiendo el criterio de Bastian y Li [2], no se ha definido dicho término adicional en las expresiones anteriores.

3.2.2. Modificaciones al algoritmo de mallado

Mientras se trabajaba con el algoritmo original, se ha encontrado conveniente realizar algunas modificaciones, expuestas a continuación.

3.2.2.1. Penalización de cercanía al contorno

Una circunstancia que no es tratada en el algoritmo original y que puede dar lugar a mallados de baja calidad es el riesgo de elección de líneas candidatas que pasen demasiado cerca de otros vértices del contorno, en relación al tamaño deseado de los elementos. Esto es especialmente perjudicial porque puede dar lugar a elementos con algún lado mucho menor que el resto.

En la función de coste (1) del algoritmo original, las candidatas que pasan demasiado cerca de otros vértices no se ven penalizadas (figuras 14a y 14b).

Para tratar de evitar que una candidata de este tipo sea elegida, se ha optado por añadir un nuevo término a la función de coste, que tiene en cuenta la relación entre el tamaño local del elemento de mallado y la distancia de la candidata al vértice más cercano:

$$\lambda = \begin{cases} 51\,(0.7 - d_r)^2 & \text{si } d_r < 0.7 \\ 0 & \text{en otro caso} \end{cases} \tag{11}$$

donde:

$$d_r = \frac{d_p}{h(p)} \tag{12}$$

siendo d_p la mínima distancia desde la línea candidata a los puntos del contorno cuya proyección sobre la candidata cae dentro de ella, y $h(p)$ el tamaño deseado de elementos en el punto proyectado.

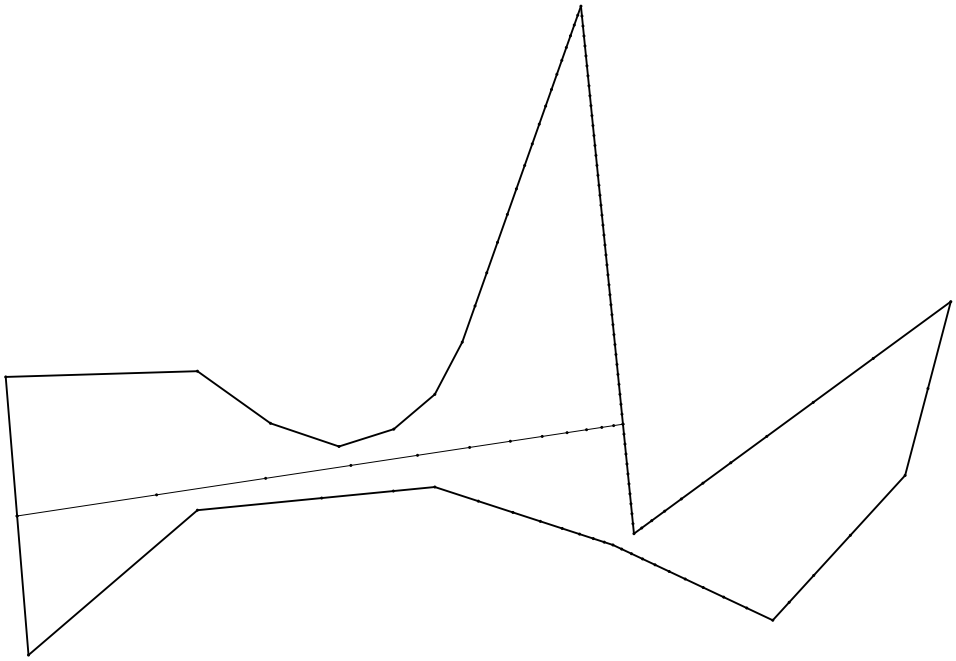
La función de coste aplicando esta penalización, queda:

$$f_{coste,p} = 0.52\phi + 0.17\sigma + 0.17\ell + 0.14\alpha + \lambda \tag{13}$$

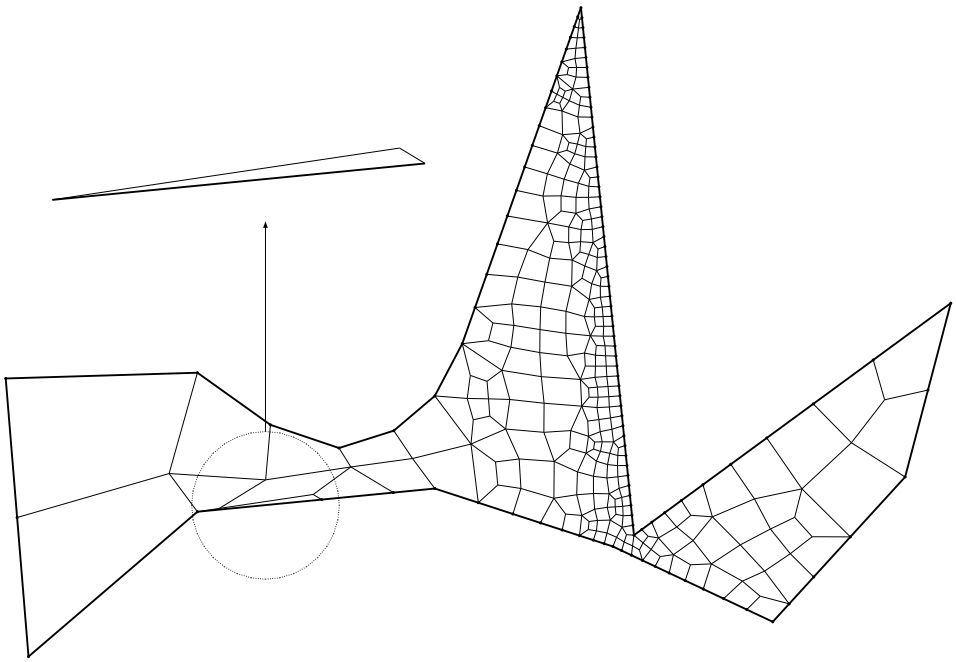
En las figuras 15a y 15b puede observarse que esta ampliación de la función de coste nos proporciona el resultado deseado.

A priori, este nuevo término podría constituir un inconveniente para el rendimiento óptimo del algoritmo, ya que nos obliga a calcular la distancia de cada candidata a cada vértice del contorno. Pero como previamente hemos hecho el test de intersección de la candidata con el contorno (requisito previo de toda candidata es que no exista tal intersección), en dicho test hemos obtenido ya la distancia buscada.

Obsérvese que una de las implementaciones más óptimas para el test de intersección entre un segmento de recta y un contorno poligonal consiste en

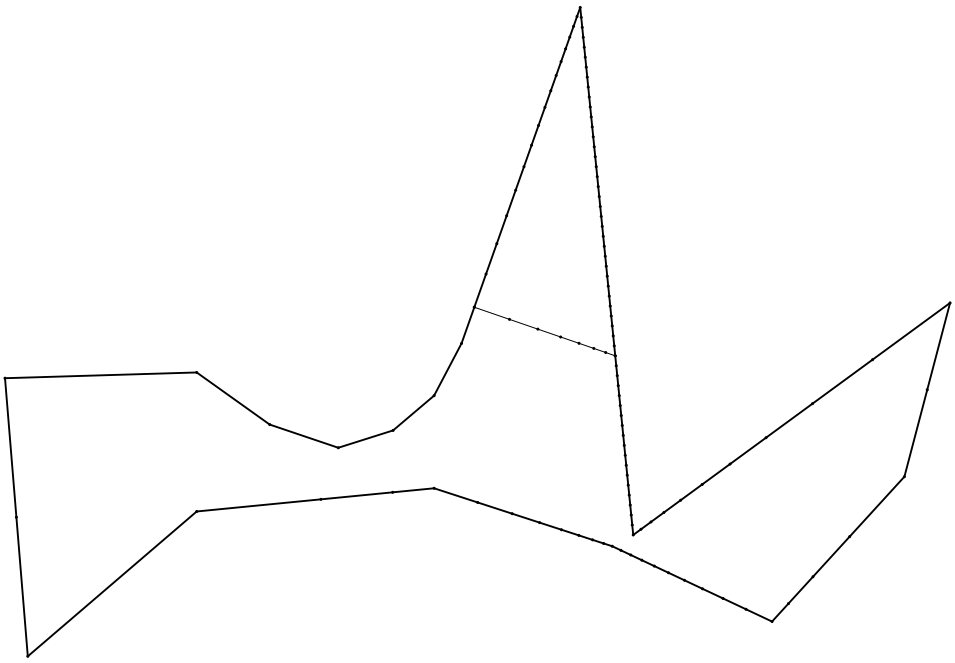


(a) Primera candidata (con algoritmo original).

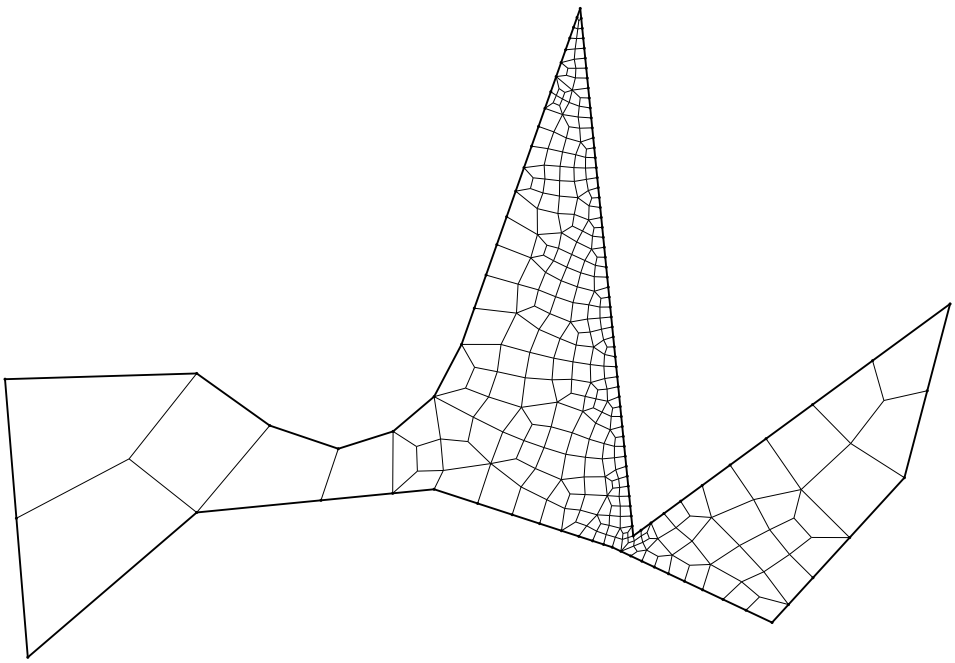


(b) Formas no deseadas en el resultado de (a).

Figura 14: Problemática de proximidad de candidatas a vértices del contorno.



(a) Primera candidata (aplicando penalización de cercanía de candidatas a vértices del contorno).



(b) Las formas no deseadas quedan eliminadas.

Figura 15: Solución penalizando candidatas próximas a vértices del contorno.

sustituir las coordenadas de los vértices del contorno en la ecuación normal de la recta de la candidata, para rápidamente descartar la posibilidad de intersección con segmentos del contorno cuyos extremos estén al mismo lado del segmento de recta. Y además, de esta manera tenemos ya inmediatamente la distancia de la candidata a cada vértice del contorno, por lo que esta modificación del algoritmo no implica aumentar su coste de tiempo.

3.2.2.2. Contornos de 6 lados

Al aplicar el algoritmo original sobre una colección de ejemplos prácticos de contornos poligonales, y probando diferentes densidades de mallado, se observa que la subdivisión de contornos de 6 lados es un momento crítico. Ese paso, en algunas circunstancias, puede dar lugar a cuadriláteros cóncavos, triangulares, o, en definitiva, elementos de baja calidad para un análisis de elementos finitos.

Esto ya fue notado por Sarrate y Huerta, quienes proponen en [50] modificar los pesos de cada término de la función de coste cuando se están subdividiendo contornos de pocos lados (además de añadir un término adicional, como se ha mencionado con anterioridad). Una solución alternativa es la expuesta por Bastian y Li [2], quienes lo resuelven clasificando los contornos de 6 lados en diferentes tipos, y aplican una subdivisión predefinida a cada uno de ellos. Se ha elegido esta última opción porque se ha observado que en general proporciona elementos de mejor calidad.

La clasificación de contornos de 6 lados propuesta por Bastian y Li se basa en tipificarlos detectando lados consecutivos colineales y, en función del tipo encontrado, proporcionar la mejor subdivisión posible. La clasificación de la topología del contorno de 6 lados se realiza según la figura 16. La subdivisión a aplicar se encuentra en la figura 17 para cada uno de los casos (nótese que la figura 17 no incluye el caso 4-1-1, pues Bastian y Li observan que no ocurre en la práctica y, caso de ocurrir, se debería a un dominio mal condicionado que merecería reconsiderarse por parte del usuario).

Por ejemplo, una forma triangular con un vértice intermedio en cada lado es un contorno de 6 lados (un hexágono irregular con forma triangular) y, según Bastian y Li, le correspondería el tipo 2-2-2 (dos lados colineales, dos lados colineales, y otros dos lados colineales) cuya subdivisión óptima es en tres cuadriláteros mediante la inserción de un nuevo vértice en el centro del hexágono triangular (resultado que no podría lograrse subdividiendo el contorno con una única línea candidata).

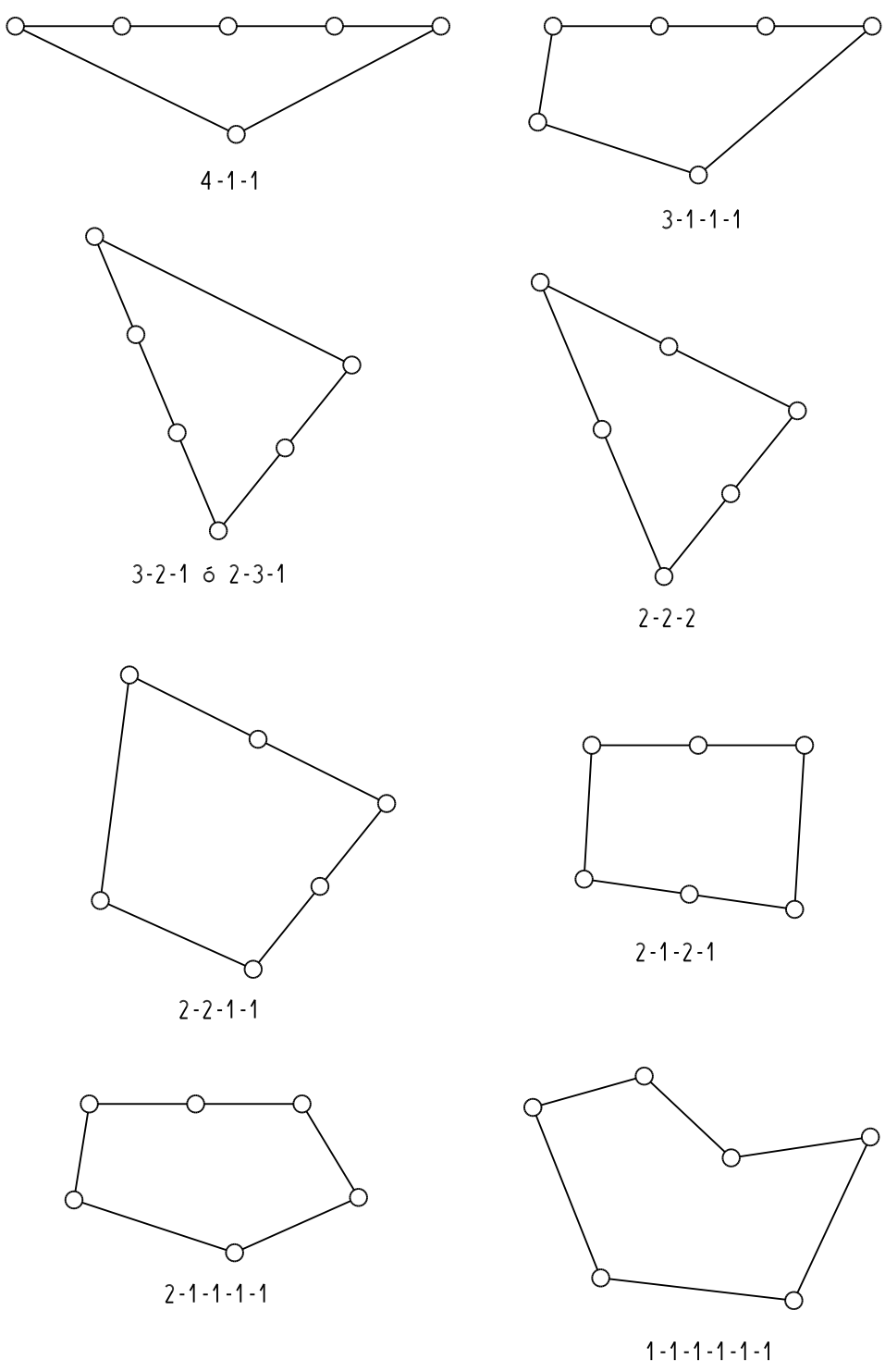


Figura 16: Clasificación de contornos de 6 lados según Bastian y Li [2].

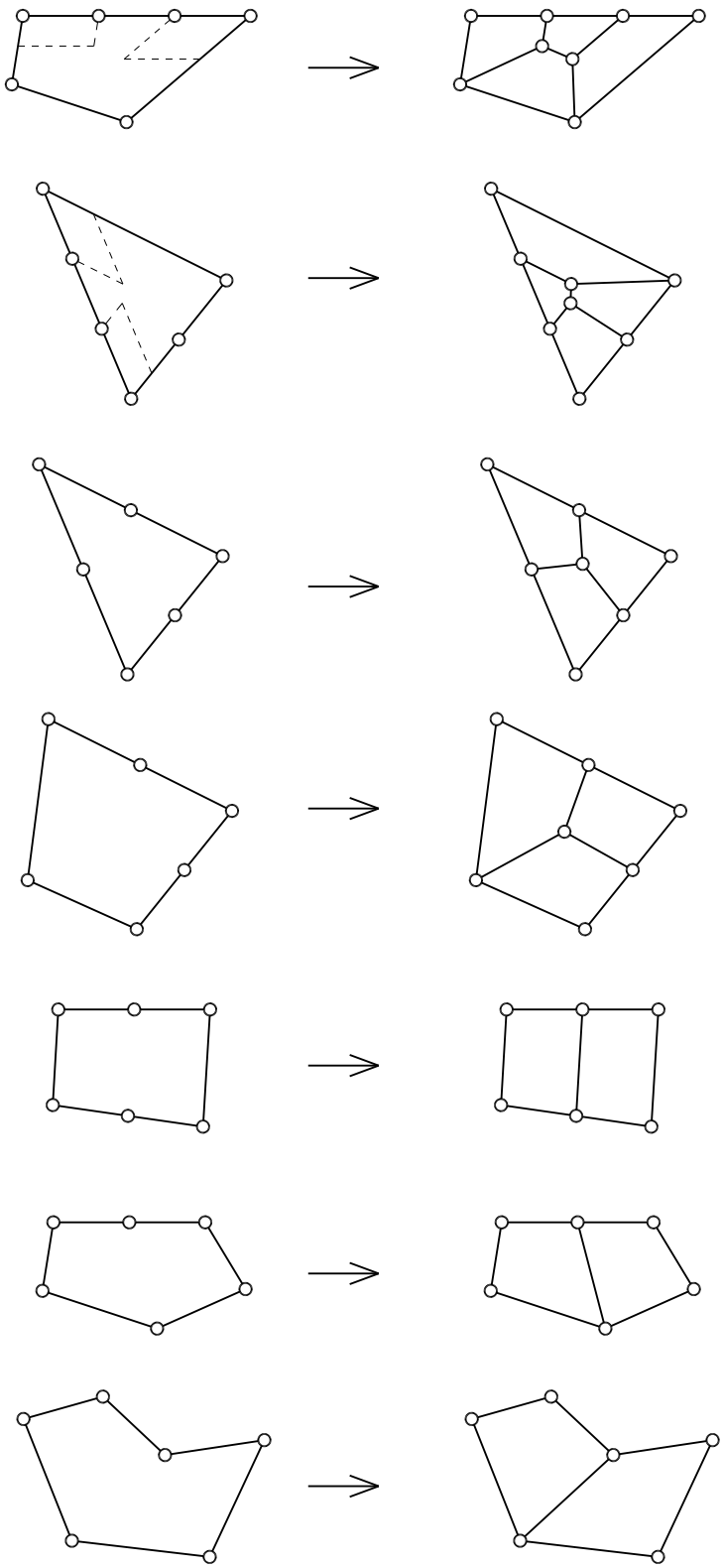


Figura 17: Soluciones para contornos de 6 lados según Bastian y Li [2].

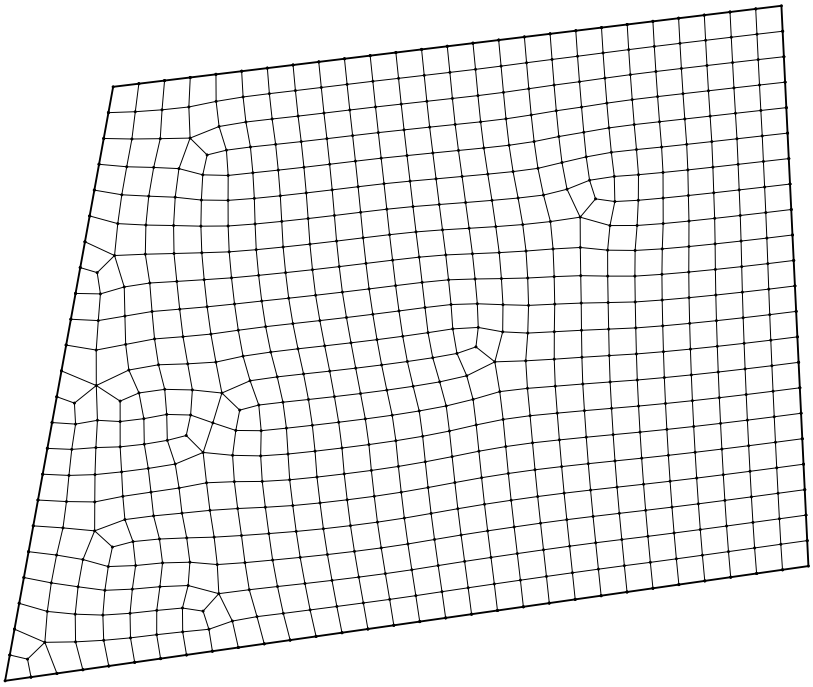


Figura 18: Mallado con subdivisión de hexágonos según Bastian y Li, con evidencias de hexágonos de tipo 2-2-2.

En la figura 18 se puede observar precisamente el resultado de la subdividisiones de hexágonos de tipo 2-2-2. La malla ya ha sido relajada, pero se sigue apreciando que había formas triangulares que han sido descompuestas en tres cuadriláteros. Por lo demás, obsérvese que la función de coste está muy bien ajustada, pues propociona un mallado regular y estructurado siempre que es posible, y sólamente deja de ser estructurado en los puntos en que es estrictamente necesario para adaptarse a un perímetro no ortogonal.

En la práctica, al implementar la clasificación de Bastian y Li, se observa que en la búsqueda de lados colineales es conveniente aplicar un umbral angular generoso, porque si dos lados consecutivos son casi colineales pero no del todo, la subdivisión de mayor calidad es la obtenida considerando que lo fuesen.

Experimentando con contornos de muy diversas formas, se ha observado que es conveniente tomar un umbral angular de incluso 30^o.

Aunque el hecho de tratar los contornos de 6 lados de manera especial pueda parecer en contra de la simplicidad y generalidad marcadas en los objetivos, sin embargo el tiempo consumido por la subdivisión de dichos contornos es prácticamente nulo en comparación con el tiempo total

de mallado. Por ello, no se consideran susceptibles de optimización, y queda justificado tratarlos aparte en aras de una mayor calidad de su subdivisión que, como se ha dicho, constituye un paso crítico en cuanto a riesgo de degeneración geométrica se refiere.

3.2.2.3. Penalización de subdivisiones conflictivas

Es posible proporcionar unos datos de entrada que estén mal concebidos y que sean especialmente difíciles de mallar con cuadriláteros. Nos referimos aquí a contornos tales que, a pesar de ser pequeños y sencillos, incluso un ser humano no sería capaz de encontrar un mallado cuadrangular de alta calidad, salvo que modificase alguno de los datos de entrada.

Por ejemplo, considérese un contorno en el que todos sus lados midan la unidad, y que deseamos mallar con cuadriláteros también de tamaño unidad. Si en ese contorno insertamos un agujero cuyos lados midan la centésima parte de la unidad, y no tomamos la precaución de definir una gradación suave de la densidad de mallado, queda claro que el algoritmo se va a encontrar con situaciones de variaciones muy bruscas del tamaño de elementos, que necesariamente conducirán a elementos de baja calidad.

El problema es parecido a las candidatas cercanas a vértices del contorno, que acabamos de exponer en el apartado anterior, porque ambos casos producen que haya vértices que disten entre sí una distancia mucho menor que el tamaño deseado de elementos en esa zona. Pero no puede solucionarse de la misma manera, porque no es causado por hacer una mala elección de la candidata de subdivisión, sino por unos datos de entrada mal concebidos.

Variaciones así de bruscas del tamaño de elementos constituyen un error conceptual de diseño en el contorno o en la definición de su densidad de mallado, pero, incluso en tales casos, el algoritmo no debería fallar, llegando siempre a un mallado cuadrangular válido aunque su calidad sea baja.

Con esta finalidad, se ha introducido la modificación de penalizar gravemente la función de coste de las líneas candidatas que generen cuadriláteros cóncavos o casi triangulares. Se penalizan sumándole al coste un valor arbitrario grande, y además exigiendo que, si esa candidata es finalmente elegida, tenga como mínimo 2 puntos intermedios de subdivisión, para que los elementos resultantes sean como mínimo hexágonos y su anomalía se arregle en las siguientes iteraciones del algoritmo.

No obstante, es evidente que si el algoritmo necesita aplicar esta medi-

da en algún paso, la calidad en la zona afectada puede ser demasiado baja para un posterior análisis de elementos finitos. Por ello, se recomienda implementar el algoritmo de manera que al detectar este tipo de subdivisiones conflictivas, se alerte al usuario y se le sugiera subsanar las deficiencias de los datos de entrada.

3.2.2.4. Subdivisión en segmentos crecientes

Tal y como se ha adelantado en el apartado *Operación general del algoritmo*, con frecuencia nos encontraremos con la necesidad de resolver el problema de subdividir un segmento recto insertando vértices intermedios, pero no equidistantes, sino en una gradación creciente o decreciente si el tamaño deseado de elemento de mallado en los extremos inicial y final del segmento es diferente.

Esta necesidad puede verse gráficamente en las figuras 14a y 15a, en las cuales la línea de subdivisión elegida conecta vértices con diferente densidad de mallado, de manera que la nueva línea debe crearse con subdivisiones de tamaño variable.

Se puede observar que este problema es análogo a la intersección de una espiral logarítmica con el eje de abscisas (figura 19).

La ecuación de una espiral logarítmica en coordenadas polares (r, θ) es:

$$r = a\mathrm{e}^{b\theta} \tag{14}$$

donde a y b son parámetros de la espiral.

Como estamos planteando la intersección de la espiral con el eje de abscisas, los puntos de intersección son aquellos que corresponden a revoluciones completas de la espiral, es decir:

$$\theta = n \cdot 2\pi \tag{15}$$

$$r = a\mathrm{e}^{bn2\pi} \tag{16}$$

Supongamos que el tamaño del primer segmento ha de ser s_1 y el del último segmento s_2, por lo que deseamos que las distancias entre los puntos de la espiral con el eje de abscisas vayan variando gradualmente desde s_1 hasta s_2.

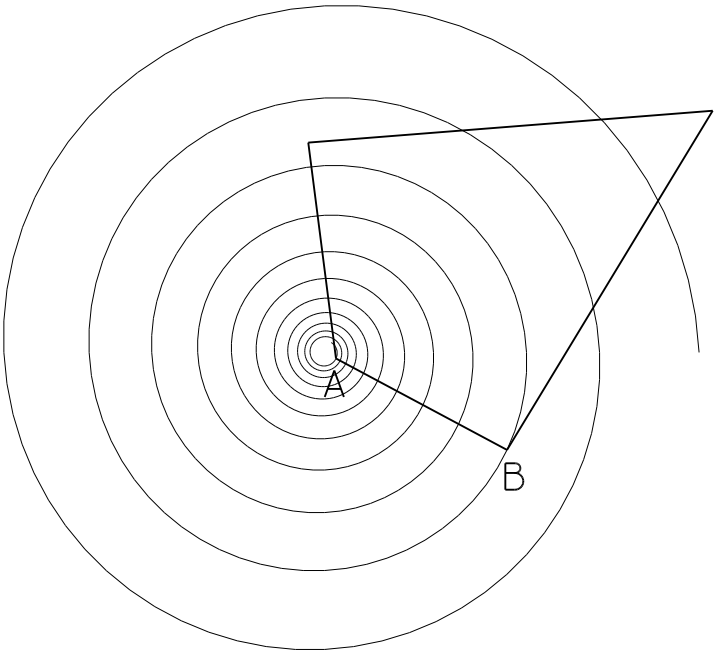


Figura 19: Dividir el lado AB en segmentos de tamaño creciente sugiere calcular la intersección del lado con una espiral logarítmica convenientemente ajustada para que los tamaños del primer y último segmento sean los deseados. El problema se puede plantear análogamente sobre el eje de abscisas.

Podemos obligar a que el tamaño del primer segmento sea s_1 si hacemos que la distancia entre los puntos de la revolución $n = 1$ y la revolución $n = 2$ sea s_1, lo cual nos lleva a la ecuación:

$$\mathrm{e}^{4b\pi} - \mathrm{e}^{2b\pi} = \frac{s_1}{a} \tag{17}$$

De manera análoga, si suponemos que vamos a necesitar c revoluciones para llegar desde el tamaño s_1 hasta s_2, tendremos que exigir que la distancia entre la revolución $n = c-1$ y la revolución $n = c$ sea s_2, lo que nos conduce a la ecuación:

$$\mathrm{e}^{2\pi bc} - \mathrm{e}^{2\pi b(c-1)} = \frac{s_2}{a} \tag{18}$$

Además, la longitud total de la línea que queremos subdividir es conocida, pues es la distancia entre el vértice inicial y el final. Si llamamos L a la longitud de la línea, tendremos que obligar a que la distancia entre el punto de la revolución $n = 1$ y el de la revolución $n = c$ sea igual a L, lo que nos proporciona una nueva ecuación:

$$\mathrm{e}^{2\pi bc} - \mathrm{e}^{2b\pi} = \frac{L}{a} \tag{19}$$

Recapitulando, vemos que las ecuaciones (17), (18) y (19) son un sistema de 3 ecuaciones con 3 incógnitas (a, b, y c). Resolviendo el sistema, obtenemos las soluciones:

$$a = \frac{-s_1(L - s_2)^2}{(L - s_1)(s_1 - s_2)} \tag{20}$$

$$b = \frac{\log\left(\frac{L-s_1}{L-s_2}\right)}{2\pi} \tag{21}$$

$$c = \frac{\log\left(\frac{s_2(L-s_1)^2}{s_1(L-s_2)^2}\right)}{\log\left(\frac{L-s_1}{L-s_2}\right)} \tag{22}$$

donde a y b serán los parámetros de la espiral, y c el número de revoluciones, es decir, el número de puntos para subdividir el segmento.

Para que la solución que se acaba de presentar sea válida, se tiene que cumplir:

$$L > s_1 \tag{23}$$

$$L > s_2 \tag{24}$$

$$s_1 < s_2 \tag{25}$$

donde las dos primeras condiciones van a ser siempre ciertas (sería absurdo tratar de subdividir la línea con segmentos más grandes que la línea), y la tercera, en caso de no ser cierta podemos invertir el vértice inicial y el final de la línea, y pasaría a ser cierta.

Sin embargo, aunque se trata de una solución válida, plantea un problema: en general, el número total de revoluciones c no va a ser un número entero. Pero necesitamos crear un número entero de subdivisiones. Una manera de solucionar este problema sería redondear c a un valor entero, lo que nos introduce la anomalía de que el valor del último segmento ya no será igual a s_2. En la práctica se ha observado que esto conduce a gradaciones de baja calidad en algunos casos.

Por tanto, se hace necesario mejorar el planteamiento:

Primero se obtiene el valor exacto de c a partir de la solución que acabamos de presentar. Redondeamos dicho valor a un número entero, e incluso lo incrementamos o decrementamos en caso necesario (pues en apartados

anteriores hemos dicho que podemos necesitar incrementar o decrementar el número de vértices del contorno para que sea par).

Al valor definitivo de revoluciones (ya entero, y ajustado para asegurar la paridad de vértices del contorno), lo llamamos d.

Y a continuación, planteamos un sistema similar al anterior, esta vez exigiendo que el número de revoluciones sea d, y asumiendo que el primer y el último segmento no van a tener una dimensión igual a s_1 y s_2, pero exigimos que su relación sí que sea igual a $k = \frac{s_2}{s_1}$:

$$k = \frac{s_2}{s_1} \tag{26}$$

$$\mathrm{e}^{2\pi bd} - \mathrm{e}^{2\pi b(d-1)} = \left(\mathrm{e}^{4b\pi} - \mathrm{e}^{2b\pi}\right) k \tag{27}$$

La única incógnita de la ecuación (27) es b. Despejando, queda:

$$b = \frac{\log\left(\frac{s_2}{s_1}\right)}{2\pi(d-2)} \tag{28}$$

Expresando la ecuación (19) con d en vez de c, podemos hallar el nuevo valor de a:

$$\mathrm{e}^{2\pi bd} - \mathrm{e}^{2b\pi} = \frac{L}{a} \tag{29}$$

$$a = \frac{L}{\left(\frac{s_2}{s_1}\right)^{\frac{d}{d-2}} - \left(\frac{s_2}{s_1}\right)^{\frac{1}{d-2}}} \tag{30}$$

En esta nueva solución de b y a queda claro que debe cumplirse $d > 2$, pero esto va a ser siempre así, por lo que no supone ningún problema.

Como conclusión, la subdivisión de un segmento en partes de tamaño creciente se resuelve de la siguiente manera:

1. Conocidos el tamaño deseado del primer y del último segmento, s_1 y s_2, comprobamos que $s_1 < s_2$. Si no se cumple, invertimos el sentido de la línea, y resolvemos el problema simétrico.

2. Averiguamos el número exacto (en general no entero) necesario de puntos, c, mediante la expresión (22).

3. A partir de c, decidimos un número definitivo entero de puntos, d, que podemos modificar, si procede, para garantizar la paridad de vértices del contorno.

4. Hallamos el valor de los parámetros de la espiral logarítmica, a y b, mediante las expresiones (30) y (28).

Y conocidos los parámetros a y b, es trivial obtener los puntos de la línea que nos generan segmentos con tamaño creciente, aplicando la ecuación de la espiral logarítmica (ecuación (16)) desde $n = 1$ hasta $n = d$.

3.3. Eficiencia en CPU y GPU de los algoritmos propuestos

Con una implementación secuencial cuidadosa del algoritmo se puede conseguir, con procesadores actuales, un rendimiento aceptable para muchos usos. En cualquier caso, la forma del contorno posee un impacto importante en el tiempo de mallado, debido a que el coste del test de intersección de cada candidata con el contorno depende de la topología de éste. Véase por ejemplo en las tablas 6 y 7 cómo el mallado de la figura 20 supera los 4 minutos de tiempo cuando se hace de manera secuencial en un núcleo de *CPU*, mientras otras figuras con un número mucho mayor de cuadriláteros necesitan un tiempo muy inferior.

Aunque a priori se obtienen unos tiempos aceptables cuando el mallado no es muy grande, el rendimiento dista bastante de poderse considerar interactivo. Cuando la forma del dominio dificulta los tests de intersección, o cuando el número de cuadriláteros generados crece, el tiempo empleado por el algoritmo de mallado obliga a esperar varios minutos. Cuando el usuario está diseñando un modelo y necesita ir probando cada cambio, ese tiempo de espera supone una pérdida importante de interactividad.

Las mediciones del tiempo de ejecución del algoritmo corriendo como código secuencial en un procesador arrojan la conclusión de que, en general, más de un 80 % del tiempo (y a menudo más de un 90 %) se dedica a la comprobación de intersección de cada línea candidata con el contorno. Estas cifras se obtienen habiendo optimizado ya el test de intersección con descartes rápidos según los signos de las distancias desde la recta candidata a los extremos del segmento comprobado con ella.

La implementación secuencial se ha diseñado de la manera más eficiente posible, evitando hacer cálculos que no van a ser necesarios. Por ejemplo, si hay intersección entre una candidata y el contorno, no es necesario hallar

su función de coste. Se han utilizado muchas optimizaciones de este tipo al escribir el código. Pero, así como una *CPU* se beneficia mucho de ello, por contra una *GPU* no puede sacar el mismo partido debido a ser una máquina *SIMD* (que sufre penalizaciones cuando el código se ramifica) y por tanto puede llegar a desaprovechar mucha de su potencia haciendo cálculos que después serán descartados.

Se ha preferido utilizar el mismo código fuente para la implementación secuencial que para la paralela, aislando el trabajo a paralelizar de manera que el modo secuencial ejecute dicho trabajo en serie, y el modo paralelo ejecute varias instancias de esa misma tarea a la vez. Esto facilita mantener el código y perfeccionar el algoritmo, al no tener que cuidar dos implementaciones diferentes.

La parte del trabajo así aislada ha sido la búsqueda de la línea de subdivisión con menor función de coste de entre todas las candidatas que salen de un vértice del contorno.

De esta manera, en el modo secuencial se evalúa el coste de cada candidata llamando desde dentro de un bucle a la función correspondiente, mientras que en el modo paralelo, es *OpenCL* quien adquiere el control del bucle, pudiendo evaluar el coste de varias candidatas en paralelo. El código fuente de este trabajo interior al bucle es el mismo en ambos modos, con la única peculiaridad de emplear definiciones del preprocesador de *C/C++* para satisfacer la sintaxis de *OpenCL* o de compiladores convencionales, según sea el caso.

Para hacer las mediciones de tiempo se ha elegido un *hardware* de gama de consumo, pues nuestro objetivo es aprovechar al máximo los recursos que habitualmente poseen la mayoría de usuarios. No obstante, dentro de la gama de consumo, se ha preferido tomar los componentes de mayor rendimiento en el momento de escribir estás líneas, para que los resultados obtenidos mantengan una vigencia más prolongada.

Como *CPU* se ha elegido el *Intel Core i7 6700K 4GHz*, de la generación *Skylake*, que posee 4 núcleos de proceso. La *GPU* utilizada ha sido la *NVIDIA Pascal Titan X*, cuyas características hemos descrito antes.

Observando los resultados (tablas 6 y 7) podemos extraer como primera conclusión que es muy importante el incremento de rendimiento al aprovechar todos los recursos disponibles. Por tanto, por este camino logramos dar respuesta al objetivo inicialmente marcado de aumentar la interactividad del mallado, aprovechando eficientemente el diseño del *hardware* actual.

	Número de cuadriláteros	1 núcleo CPU	4 núcleos CPU	Mejora de rendimiento
Figura 20	41697	267s	44s	x6.07
Figura 21	109247	111s	21s	x5.29
Figura 22	30400	19s	4s	x4.75
Figura 18	314899	120s	27s	x4.44

Tabla 6: Tiempos de mallado en CPU Intel Core i7 6700K 4GHz (4 núcleos con *hyperthreading*, comportándose como 8 núcleos virtuales). Comparativa de rendimiento entre usar 1 núcleo (implementación secuencial) y los 4 núcleos de la CPU (paralelizados con OpenCL). El código se ha compilado con coma flotante de precisión simple IEEE en ambos casos. *Nota: La figura 18 ha sido mallada con una densidad mucho más fina, produciendo 314899 cuadriláteros, para estas pruebas de rendimiento.*

	Número de cuadriláteros	1 núcleo CPU	1 GPU	Mejora de rendimiento
Figura 20	41697	267s	43s	x6.21
Figura 21	109247	111s	44s	x2.52
Figura 22	30400	19s	7s	x2.71
Figura 18	314899	120s	48s	x2.50

Tabla 7: Comparativa de tiempos de mallado entre 1 GPU NVIDIA Pascal Titan X (implementación con OpenCL) y 1 núcleo de CPU Intel Core i7 6700K 4GHz (implementación secuencial). El código se ha compilado con coma flotante de precisión simple IEEE, tanto al correr en CPU como en GPU. *Nota: La figura 18 ha sido mallada con una densidad mucho más fina, produciendo 314899 cuadriláteros, para estas pruebas de rendimiento.*

Otra conclusión, que ya sospechábamos, consiste en que la paralelización en *CPU* escala el rendimiento muy bien (por encima del cuádruple[31] en todos los ejemplos realizados), mientras que hay una mayor variabilidad de tiempos al paralelizar en *GPU*. Ya lo intuíamos, por la naturaleza *SIMD* de estos dispositivos, que puede acarrear penalizaciones importantes al ejecutar sentencias condicionales.

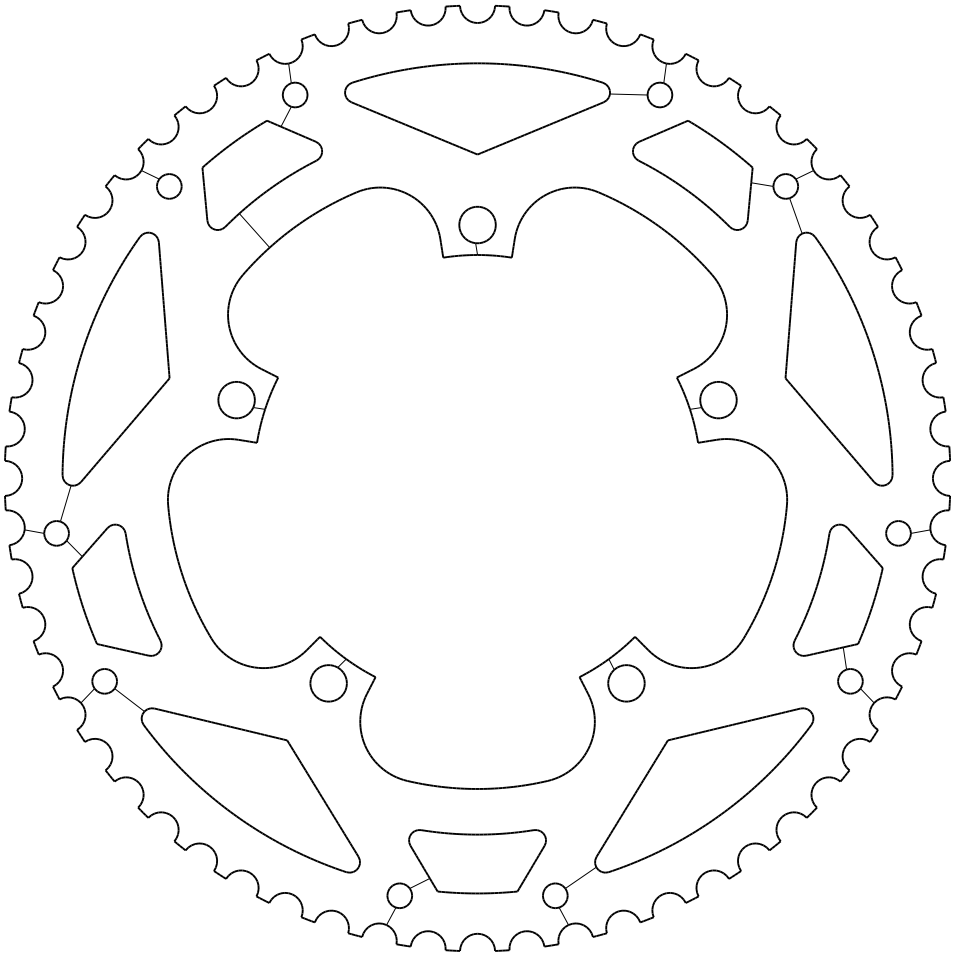
De hecho, el ejemplo que adquiere una mayor ganancia de rendimiento (superior al séxtuple), logra su mejor tiempo en *GPU*. Pero la *GPU* presenta

[31] A primera vista puede extrañar que al paralelizar el algoritmo en un procesador con 4 núcleos se obtenga una mejora de rendimiento superior al cuádruple. La razón de ello es doble: primero, los núcleos actuales de *Intel* proporcionan *hyperthreading* (una tecnología en que cada núcleo se comporta como dos núcleos virtuales, ventaja que no se puede aprovechar con código secuencial, pero sí con *OpenCL* -de hecho *OpenCL* detecta 8 *compute units* en un *Intel Core i7* de 4 núcleos). Segundo, *OpenCL* aprovecha algunas instrucciones vectoriales *SIMD* presentes en procesadores *Intel*.

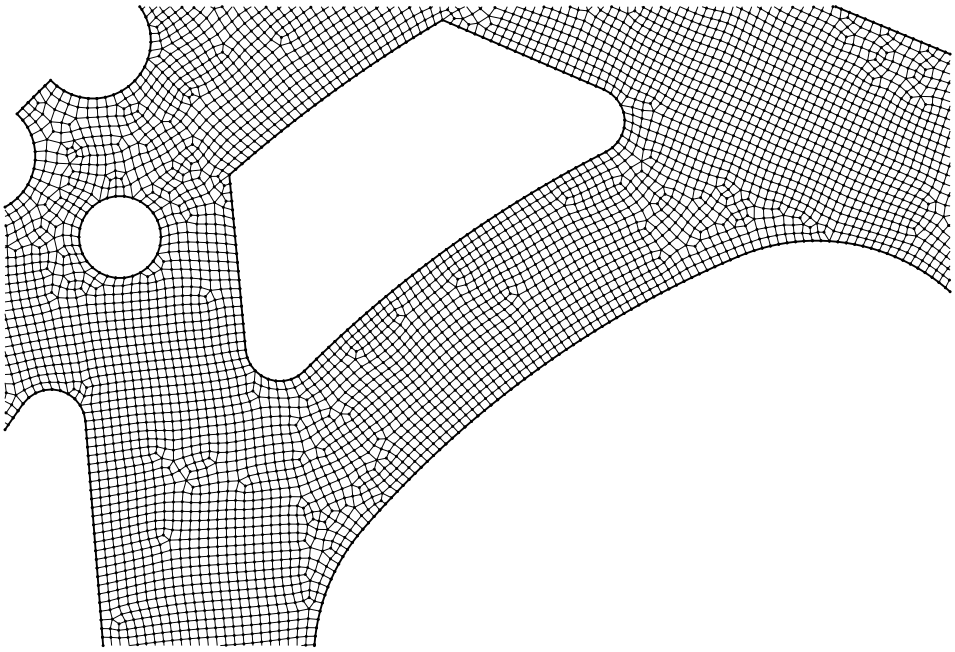
también las menores ganancias en otros casos (unas dos veces y media de mejora, frente a más del cuádruple de la *CPU*).

Con todo, queda patente que la forma del contorno (y en especial la existencia, número, y tamaño de concavidades y agujeros) influye significativamente en los tiempos, ocasionando que el factor predominante en el coste del mallado no sea el número de cuadriláteros.

Es importante recalcar que la implementación en *GPU* ha utilizado exactamente el mísmo código que la de *CPU*. Probablemente una reimplementación específica para *GPU*, enfocada a una arquitectura *SIMD* mejoraría los tiempos, aunque tendría como contrapartida la mayor dificultad del mantenimiento del código, por la dualidad de versiones.

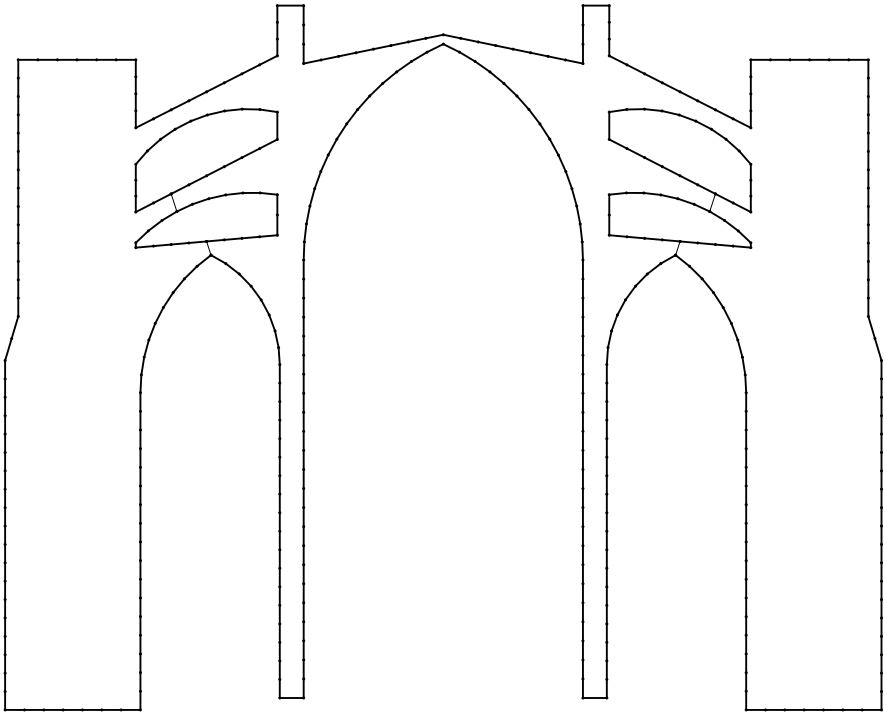


(a) Contorno inicial (agujeros ya unificados).

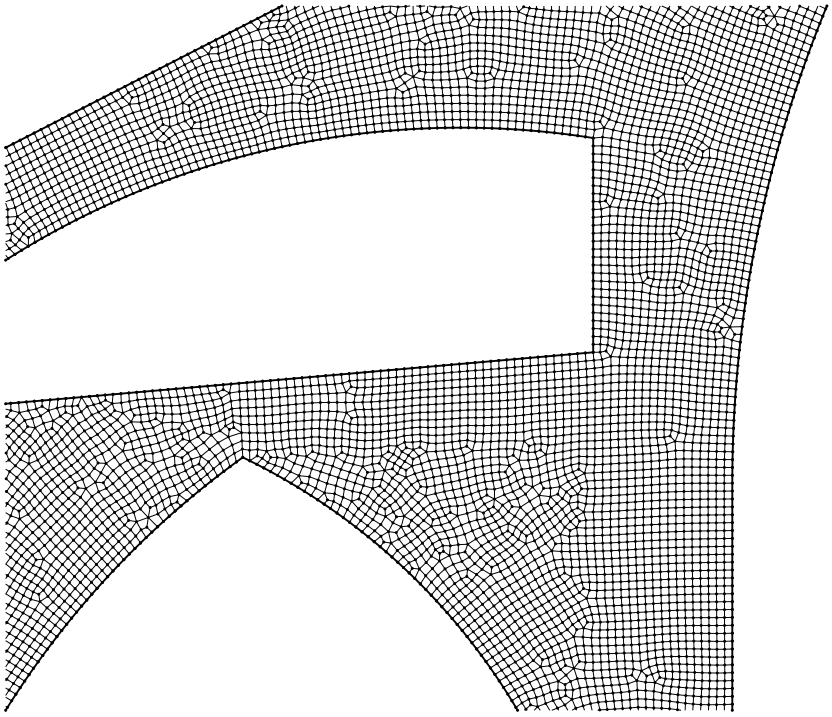


(b) Detalle del mallado.

Figura 20: Plato de bicicleta de 60 dientes (41697 cuadriláteros).

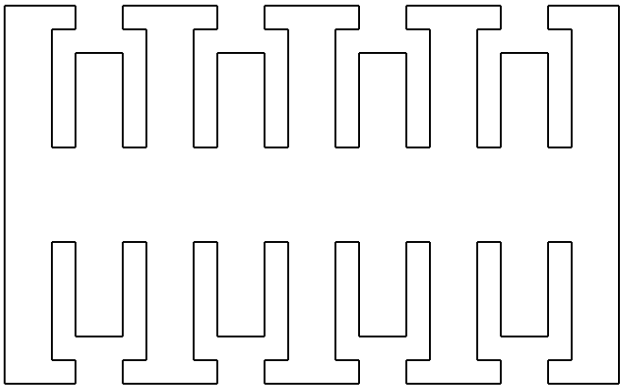


(a) Contorno original de la sección transversal de la Catedral de Palma de Mallorca a partir de [32], con los agujeros ya unificados mediante aristas dobles.

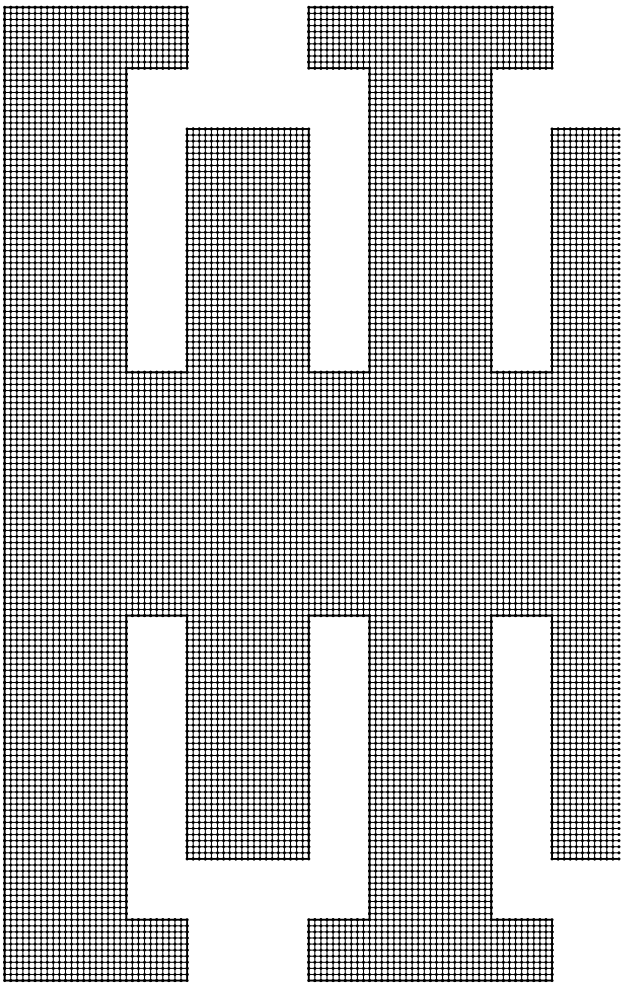


(b) Detalle del mallado.

Figura 21: Modelo de la Catedral de Palma de Mallorca para medición de rendimiento, mallado con 109247 cuadriláteros.



(a) Contorno inicial.



(b) Detalle del mallado.

Figura 22: Contorno empleado por Sarrate y Huerta en [50] para mostrar que el algoritmo genera un mallado estructurado cuando el perímetro lo permite. Aquí se emplea con una densidad de mallado mayor (30400 cuadriláteros), con la finalidad de mediciones de rendimiento, y para verificar que las modificaciones realizadas en el algoritmo no obstaculizan la regularidad cuando ésta es posible.

3.4. Análisis de la calidad del mallado mediante postproceso

Los algoritmos de mallado cuadrangular producen en general un resultado que requiere de un postproceso para que su calidad se acerque lo más posible a la deseada. El postproceso puede ser de tipo *make-up* (capaz de modificar la topología, creando, eliminando, y reconectando nodos y elementos), o de tipo *smoothing* (suavizado, optimización de la forma modificando exclusivamente las coordenadas de los nodos interiores pero manteniendo la topología).

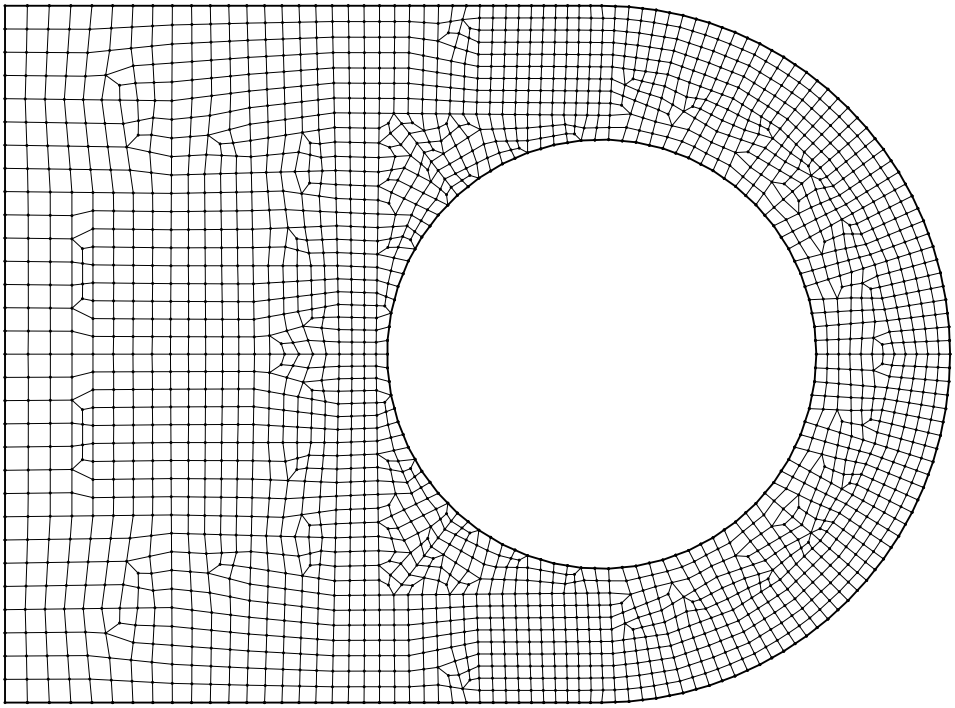
Un postproceso de suavizado nunca podrá lograr una calidad elevada a partir de un mallado que originalmente no posea una topología óptima. Es por esto que se ha dedicado un esfuerzo considerable en la elección del algoritmo de mallado y en los ajustes y detalles de implementación descritos en apartados precedentes, para que así el postproceso pueda ser de sólo suavizado, sin requerir *make-up*.

En la figura 23a se muestra la salida directa del algoritmo de mallado al aplicarlo a un contorno que tiene algunas peculiaridades: por una parte combina zonas rectangulares y circulares, planteando la dificultad de transición entre perímetros rectos y curvos. Además, se ha especificado que el tamaño de los elementos en los vértices de la izquierda sea alrededor del doble de los elementos de los bordes curvos, lo que genera una gradación de tamaño de elementos en el mallado.

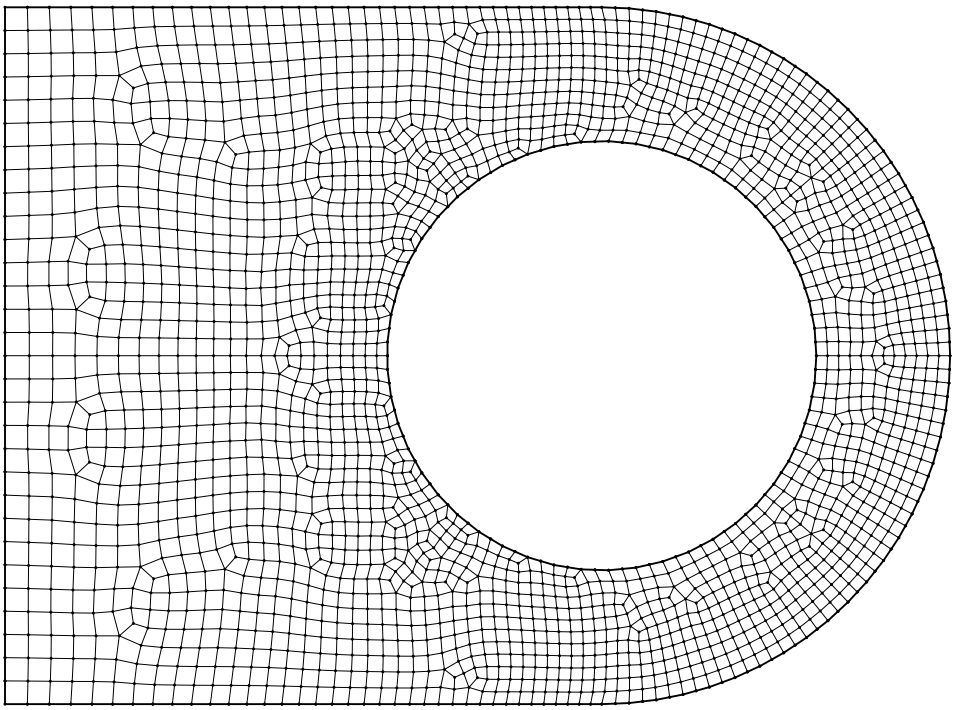
El contorno se ha definido de manera completamente simétrica respecto del eje X, para mostrar que una correcta implementación del algoritmo produce un mallado simétrico si las condiciones de partida tienen simetría.

Se observa que el resultado mostrado en la figura 23a satisface los requisitos que cabría esperar de una buena topología cuadrangular: favorecer el mallado estructurado siempre que sea posible, adecuación a gradaciones de tamaño de elementos cuando éste es variable a lo largo del dominio, empleo de cuadriláteros exclusivamente, adaptación a contornos arbitrarios, así como tendencia a seguir la direccionalidad marcada por el perímetro (los cuadriláteros adquieren direccionalidad radial al llegar a la zona de la corona circular). Todo esto lo ha logrado el algoritmo de manera automática, sin intervención del usuario.

Sin embargo, este resultado dista mucho de la calidad de forma que esperamos de un mallado. Hay cuadriláteros bastante distorsionados, y no todos los elementos tienen el tamaño que corresponde a la zona del dominio en que se encuentran. La topología es buena, pero se requiere modificar las

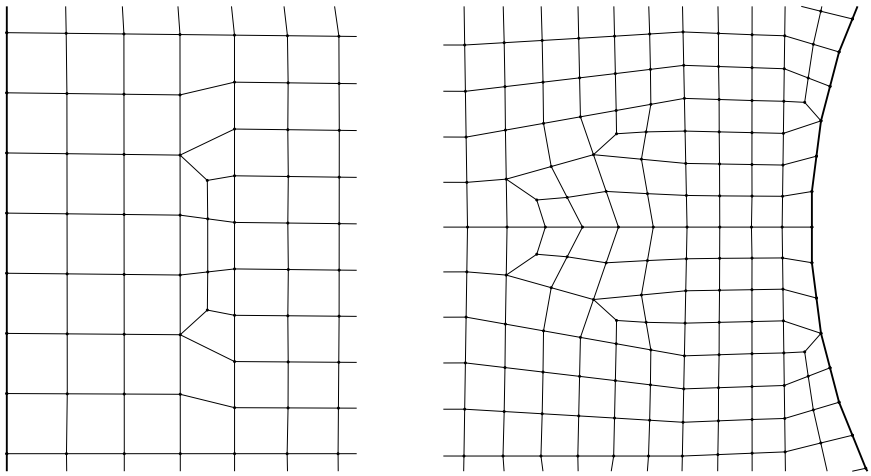


(a) Sin postproceso.



(b) Con postproceso.

Figura 23: Mejora de calidad alcanzable con postproceso. Obsérvese que este ejemplo posee una gradación del tamaño de los elementos (doble de grande en los vértices de la izquierda).



(a) Elementos de tamaño muy diferente al que deberían tener en esa zona.

(b) Distorsión excesiva en algunos elementos.

Figura 24: Deficiencias de calidad sin postproceso.

coordenadas de los nodos para que la calidad sea la deseada.

La figura 23b es un ejemplo de una mejora de calidad del mallado original, aplicando un postproceso de suavizado.

No obstante, antes de progresar en esta dirección, necesitamos definir qué entendemos por "calidad" del mallado. ¿Se puede cuantificar? ¿Es una magnitud objetiva? ¿O depende de criterios? Y, en este último caso, ¿son criterios divergentes?

Si observamos con detenimiento la figura 23a, previa a postproceso, llegamos a la conclusión de que los elementos cuya calidad no es adecuada caen en uno de los dos casos mostrados en las figuras 24a y 24b: O bien tienen una forma demasiado distorsionada, o bien son elementos con un tamaño inapropiado (o ambas cosas a la vez).

Por tanto, si aplicamos un postproceso que solucione estos dos tipos de deficiencia, habremos obtenido un mallado con la calidad deseada.

3.4.1. Cuantificación de la distorsión

Existen diversas propuestas para evaluar la distorsión de forma de los elementos. Oddy *et al* [42] presentan una cuantificación sencilla aplicable a paralelogramos, que puede generalizarse a cuadriláteros arbitrarios.

Lee y Lo [28] emplean en su algoritmo de mallado indirecto otra medida de la distorsión, que les permite elegir qué triángulos fusionar para formar cuadriláteros de la máxima calidad posible (recordemos que su algoritmo genera cuadriláteros a base de agrupar triángulos de un mallado ya existente). Canann *et al* [5] se basan en la medida de Lee y Lo, con algunas modificaciones, para su implementación en el paquete ANSYS.

Knupp [27] propone una formulación general, a modo de *framework* adaptable a diversas necesidades, contemplando un conjunto de medidas que influyen en la calidad.

De entre todas las formulaciones de la medida de distorsión, se ha elegido la de Oddy *et al* por su sencillez de implementación y porque proporciona buenos resultados al calificar cuadriláteros. Sin embargo, el método que se describirá en los siguientes apartados no depende de ella, y puede adaptarse a cualquier otra si se prefiere (cierto es que la formulación final que se facilita lleva embebida la medida de Oddy *et al*, con la finalidad de proporcionar unas ecuaciones simplificadas y listas para su implementación en *software*, pero es posible desarrollarlas empleando cualquier otra medida de la distorsión).

Siguiendo el desarrollo de [49], la *distorsión de Oddy* de un paralelogramo se evalúa como:

$$D_{Oddy} = 2(Q_{Oddy}^2 - 1) \tag{31}$$

siendo Q_{Oddy} la *eficiencia geométrica*:

$$Q_{Oddy} = \frac{l_1^2 + l_2^2}{2A} \tag{32}$$

donde l_1 y l_2 son las longitudes de dos lados contiguos cualesquiera del paralelogramo y A es su área.

Una manera de extender esta expresión a cuadriláteros arbitrarios consiste en evaluarla en los cuatro paralelogramos que pueden construirse localmente en cada uno de los vértices del cuadrilátero (trazando lados paralelos a los lados que se encuentran en el vértice).

Tal y como muestran Sarrate y Coll en [49], esto nos conduce a la siguiente expresión para la obtención de la distorsión de Oddy en un vértice de un cuadrilátero arbitrario:

$$l_{[i]}^2 = (x_{[i+1]} - x_{[i]})^2 + (y_{[i+1]} - y_{[i]})^2$$

$$l_{[i+3]}^2 = (x_{[i+3]} - x_{[i]})^2 + (y_{[i+3]} - y_{[i]})^2$$

$$A_{[i]} = (x_{[i+1]} - x_{[i]})(y_{[i+3]} - y_{[i]}) - (x_{[i+3]} - x_{[i]})(y_{[i+1]} - y_{[i]}) \tag{33}$$

$$Q_{Oddy,i} = \frac{l_{[i]}^2 + l_{[i+3]}^2}{2A_{[i]}}$$

$$D_{Oddy,i} = 2(Q_{Oddy,i}^2 - 1)$$

para cada vértice $i = 1, 2, ...4$ del cuadrilátero (nótese que en las referencias a coordenadas de los vértices $[i+1]$ e $[i+3]$ hay que recorrer el orden circular de vértices en el cuadrilátero, es decir, hay que aplicar $[i] = mod(i-1, 4)+1$ para evitar desbordamiento).

Dado que deseamos un único valor que mida la distorsión global del cuadrilátero, la manera de no pasar por alto problemas locales de forma es tomar el máximo valor de distorsión de todos los vértices del cuadrilátero. Es decir:

$$D_{Oddy}^{cuad} = \max_{i=1,2,...4}(D_{Oddy,i}) \tag{34}$$

expresión que evidentemente no es diferenciable y por tanto su minimización mediante métodos habituales plantea dificultades.

La solución propuesta en [49] para que la magnitud sea diferenciable es tomar el valor medio de la distorsión en los vértices pero, tal y como hacen notar los autores, esto nos puede proporcionar una distorsión inferior a la que cabría adjudicar al cuadrilátero. En el método que se propondrá en los apartados siguientes no ha sido necesario exigir que D_{Oddy} sea diferenciable, razón por la cual tomaremos la expresión del máximo en vez de hacer el promedio, lo que nos permitirá capturar anomalías locales que ocurran sólo en algunos vértices.

Por otra parte, una implementación directa de la expresión (31) puede conducir a la aparición de asíntotas (cuando $A = 0$) o a detectar valores de distorsión relativamente bajos en cuadriláteros degenerados (con vértices cóncavos). Esto último se debe a que al elevar Q_{Oddy} al cuadrado se pierde su signo, que es igual al signo de A (obsérvese que un vértice será cóncavo sí y sólo sí el área A del paralelogramo local al vértice es negativa -suponiendo

que el orden de vértices del cuadrilátero es antihorario, prerrequisito que hemos exigido ya en el algoritmo de mallado).

Estos casos conducen a [49] a la proposición de modificaciones en la formulación de D_{Oddy}. No obstante, en el método propuesto en los próximos apartados, esta casuística se contempla exigiendo $A > 0$ en todos los paralelogramos locales, lo que permite no necesitar modificar la expresión original de D_{Oddy}.

3.4.2. Primera aproximación a una mejora de calidad

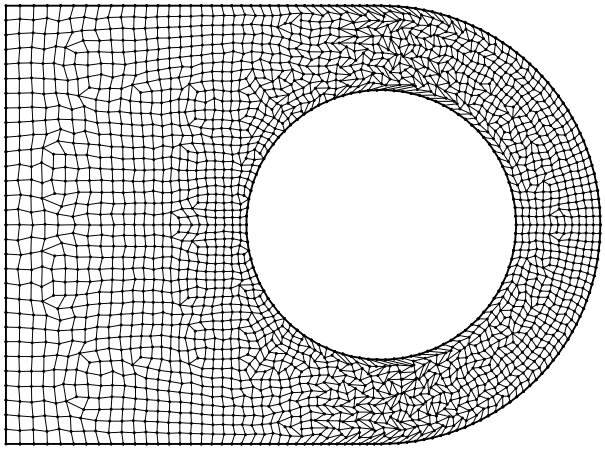
Una respuesta directa a los problemas de calidad observados en las figuras 24a y 24b consistiría en minimizar la distorsión D_{Oddy} (esto debería solucionar los problemas de la figura 24b), y en minimizar el error del tamaño de los elementos respecto al que deberían tener (lo que debería solucionar las deficiencias de la figura 24a).

Aplicando una minimización a fuerza bruta (evaluando en un número masivo de puntos), estas respuestas directas nos conducen a lo mostrado en la figura 25.

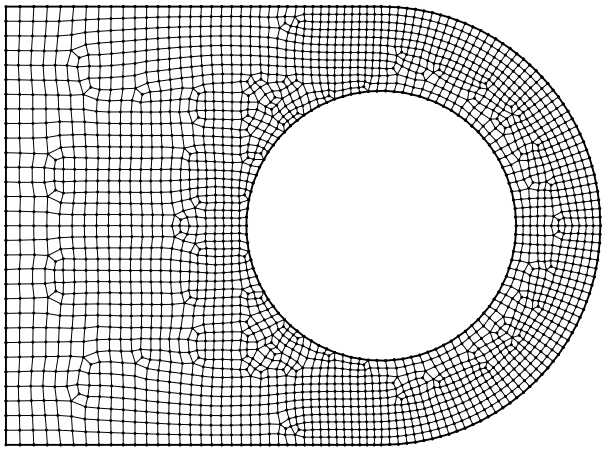
En la figura 25a se han modificado las coordenadas de los vértices de manera que sea mínimo el error del área de cada cuadrilátero respecto del área de un cuadrado cuyo tamaño fuese igual al deseado en cada zona del dominio. Con esto solucionamos la deficiencia de tamaños incorrectos, pero salta a la vista una contrapartida importante: Ha aumentado significativamente la distorsión de bastantes cuadriláteros, y el resultado es inaceptable, bastante peor que el mallado antes del postproceso (figura 23a).

Por contra, en la figura 25b se ha minimizado la distorsión D_{Oddy}. El resultado es a priori mucho más agradable a la vista, pero una inspección cuidadosa nos muestra que -tal y como era de esperar- no se han corregido los problemas de tamaños inadecuados de algunos cuadriláteros, porque esta minimización ha ignorado por completo el tamaño de los cuadriláteros. La figura 25c pone de relieve la existencia de cuadriláteros más pequeños y más grandes que sus vecinos, tras la minimización de distorsión D_{Oddy} (en una zona en que se deseaba que los elementos fueran de tamaño uniforme).

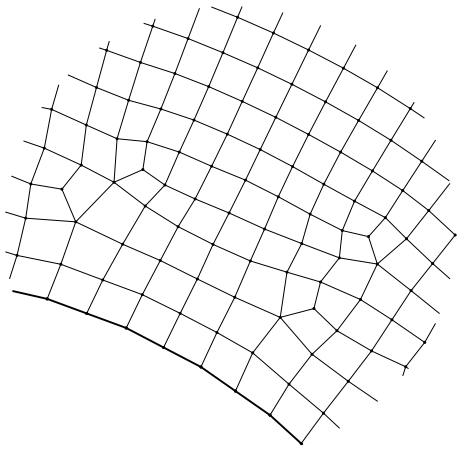
En conclusión, proporcionando por separado la respuesta directa a los problemas de calidad que se había observado, no llegamos a una solución satisfactoria.



(a) Minimización del error de área.



(b) Minimización de la distorsión de Oddy.



(c) Detalle de la minimización de distorsión Oddy, mostrando elementos notablemente más grandes y más pequeños que sus vecinos (en una zona en que todos deberían tener igual tamaño).

Figura 25: Primera aproximación a una mejora de calidad.

Podemos por tanto contestar a la pregunta formulada un poco antes: La calidad del mallado cuadrangular va a depender de al menos dos criterios: la distorsión de los cuadriláteros, y su tamaño. Y conducen en general a soluciones divergentes (no es posible mejorar al máximo uno de los dos criterios sin que se vea perjudicado el otro).

Vislumbramos ya que una solución óptima a ambos criterios va a tener que pasar por algún tipo de compromiso entre ellos.

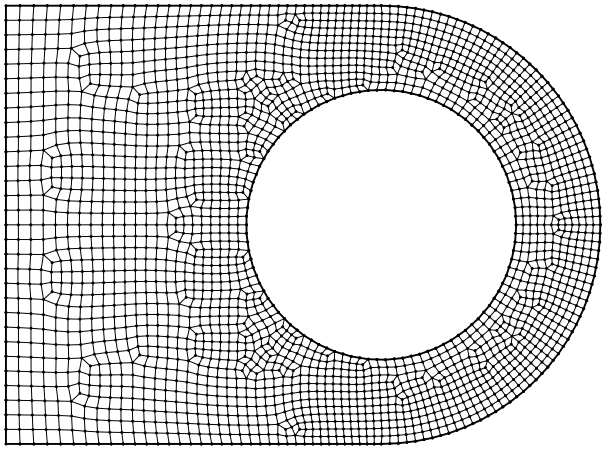
3.4.3. Algunos algoritmos de postproceso de suavizado

La primera sugerencia propuesta en [50] es el postproceso de suavizado mediante el algoritmo de Giuliani [18]. Dicho algoritmo no sólo tiende a minimizar la distorsión, sino que también favorece que se iguale el tamaño de los elementos entre sí. Aunque pueda parecer que ambas cosas son precisamente lo que buscamos, no es exactamente así, dado que en general desearemos poder realizar mallados con gradaciones del tamaño de los elementos (como el ejemplo que estamos viendo en las figuras de estas páginas), y el algoritmo original de Giulani tiende a eliminarlas.

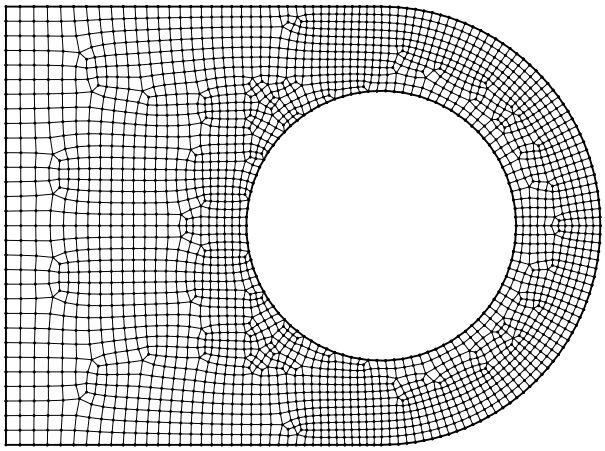
En la figura 26a se puede observar cómo se ha perdido de manera bastante notable la gradación de tamaño de elementos del mallado original (figura 23a).

Es por ello que Sarrate y Huerta propusieron en [50] una variación muy sencilla del algoritmo de Giuliani que respeta el tamaño que los elementos tienen en el mallado. Evidentemente, aunque con esto se logra respetar las gradaciones de tamaño, no se corrigen los errores de tamaño que puedan existir en el mallado previo al postproceso. Se mantiene el tamaño previo de los elementos, sin corregirlo (de hecho el algoritmo no posee información sobre el tamaño deseado de los elementos, así que difícilmente puede hacer correcciones).

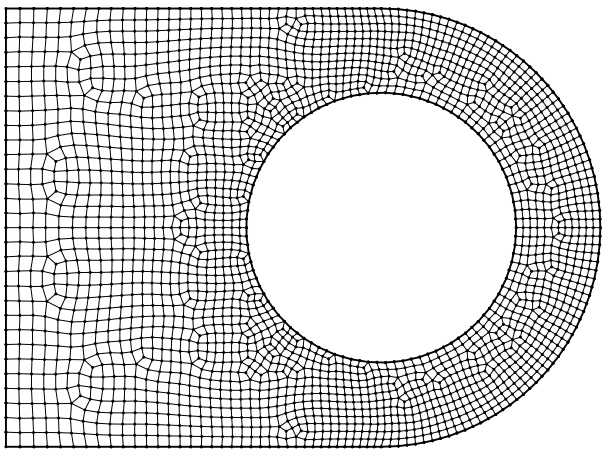
Cabe mencionar sin embargo, que la experiencia parece mostrar que esta modificación del algoritmo de Giuliani no sólo respeta las gradaciones de tamaño, sino que a menudo tiende a exagerarlas, lo que produce una notable sensación de "patrones de ruido" en algunas zonas del mallado, por la aparición de zonas con elementos más pequeños que su entorno. En la figura 26b puede observarse que este postproceso no sólo no ha igualado el tamaño de los elementos, sino que ha incrementado las diferencias, generando zonas con elementos más pequeños que su entorno, con apariencia de "patrones de ruido".



(a) Algoritmo original de Giuliani.



(b) Giulani con las modificaciones de Sarrate y Huerta.



(c) Minimización del producto del error de área por la media de la distorsión.

Figura 26: Algunos algoritmos de postproceso.

En [17], Gargallo, Roca y Sarrate hacen una nueva propuesta de postproceso de suavizado basándose en los criterios de distorsión de Knupp [27]. En este nuevo desarrollo introducen el error de tamaño en la función a minimizar, de manera que el postproceso reduzca no sólo la distorsión sino también los errores de tamaño. Para ello, minimizan una función que es producto de la distorsión (η_{sh}, en rango $[1, \infty[$) por el error de tamaño (η_{si}, también en rango $[1, \infty[$):

$$\eta(elem) = \eta_{sh}(elem)\eta_{si}(elem) \tag{35}$$

donde podemos ver la idea que se ha adelantado antes: lograr una calidad óptima pasa por un compromiso entre la distorsión y el error de tamaño (en este caso el compromiso tomado es el producto de ambos valores).

Dado que el desarrollo de [17] se basa en las medidas de distorsión de Knupp [27] y que además minimiza la media aritmética (y no el máximo) de la distorsión en las esquinas del elemento, no es directamente comparable con los resultados del método que propondremos a continuación. Pero sí que se ha hecho una reinterpretación de lo expuesto en [17], implementándolo minimizando el producto del error del tamaño de elemento (en rango $[1, \infty[$) por la media de la distorsión Oddy en los vértices del elemento (sumándole [1 para que esté también en rango $[1, \infty[$).

Los resultados obtenidos con esta reinterpretación son los mostrados en la figura 26c.

3.4.4. Propuesta de método para postproceso de suavizado

Durante la preparación de apartados anteriores que versaban sobre la elección de un algoritmo de mallado y las modificaciones que ha sido conveniente introducir, surgió muy pronto la necesidad de disponer de un método de postproceso de suavizado que, a partir de la topología producida por el mallador, optimizase su calidad.

Este método debía mantenerse fiel a los mismos requisitos exigidos al algoritmo de mallado. Si suprimimos los requisitos específicos a la topología del mallado cuadrangular, el listado que habíamos presentado en apartados anteriores queda de la siguiente manera, como exigencias a imponer igualmente al método de postproceso:

- Debe de poderse paralelizar de manera óptima.

- Debe quedar libre de la consideración de casos particulares numerosos y complejos.

- Los elementos cuadrangulares resultantes deben tener una calidad óptima para un análisis de elementos finitos.

- El usuario no debe intervenir en el proceso de mallado.

- No debe existir posibilidad de que ocurran situaciones anómalas que conduzcan a una geometría degenerada.

- Su implementación debe ser muy sencilla, para poder velar adecuada y eficientemente por todos estos requisitos.

Conviene destacar que la exigencia que persigue una "calidad óptima" deberá tener en cuenta las cuestiones que se acaban de presentar al hablar sobre postproceso de suavizado. Es decir, necesitamos reducir al máximo la distorsión, pero luchando por alcanzar también el tamaño deseado de elementos en cada zona del dominio. Ya hemos visto que estas dos metas son en general divergentes, lo que nos conduce a establecer un compromiso entre ambas.

Teniendo en cuenta que los elementos son cuadriláteros, surge de inmediato la idea de que las diagonales tienen gran responsabilidad en la distorsión del cuadrilátero, mientras que los lados influyen principalmente en el tamaño del mismo. Evidentemente no se trata de responsabilidades del todo independientes: las diagonales también influyen en el tamaño del cuadrilátero (a igual distorsión serán mayores las diagonales de un cuadrilátero de mayor tamaño), y los lados también repercuten en la distorsión (conforme más iguales sean las longitudes de los lados, más factible será poder lograr una distorsión baja).

Pero queda claro que si dejamos fijo el tamaño de los lados del cuadrilátero, las diagonales adquieren una importancia crucial en la distorsión (ver figura 27). Y si deseamos modificar el tamaño sin alterar la distorsión, hemos de aplicar un mismo factor de escala a todos los lados y a las diagonales.

La diferente repercusión de los lados y las diagonales en la calidad final del cuadrilátero nos recuerda el compromiso que necesitamos tomar entre distorsión y tamaño, y nos invita a plantear el problema como un compromiso entre el tamaño de los lados y el de las diagonales.

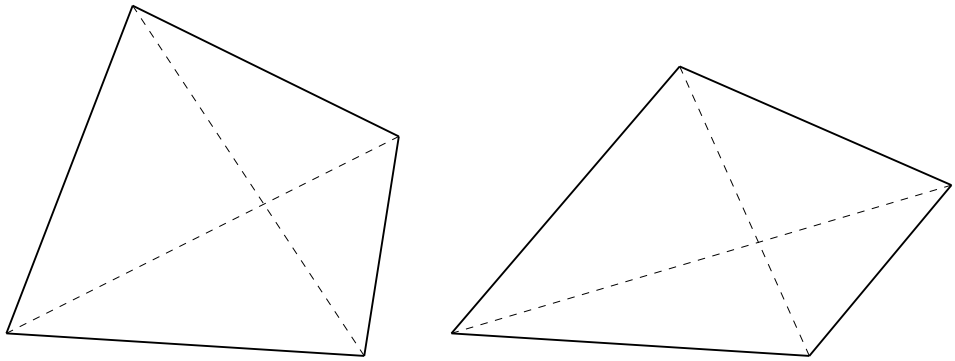


Figura 27: Los lados de ambos cuadriláteros tienen igual longitud, pero sus diagonales varían, alterando la distorsión.

Y formular un compromiso entre longitudes tiene el paralelo físico de los sistemas de muelles, o de equilibrio de barras sometidas a esfuerzo axil.

Existe trabajo previo sobre postproceso de suavizado con muelles (Lohner *et al* [31], así como una implementación en el paquete de *software* ANSYS), pero estos desarrollos no contemplan la búsqueda de una distorsión mínima de elementos cuadrangulares (su formulación presupone mallas triangulares). Cabe por tanto plantearse si es posible llegar a un modelo de muelles que proporcione una solución óptima a la calidad que estamos buscando, en un mallado cuadrangular.

Ello requerirá introducir la magnitud de la distorsión de Oddy en el modelo, para que los muelles tiendan a minimizarla. Y también será necesario calibrar los muelles de los lados respecto de los muelles de las diagonales, para dar una respuesta adecuada al compromiso entre distorsión y tamaño deseados.

3.4.4.1. Sin diagonales y con comportamiento lineal

Se ha estimado conveniente hacer un primer planteamiento con hipótesis de comportamiento lineal de los muelles, en el cual además no haya diagonales, sino sólo muelles en los lados de los cuadriláteros. En el siguiente apartado ampliaremos el modelo para permitir comportamiento no lineal en los muelles de los lados (que será la formulación finalmente adoptada, por lograr resultados de mayor calidad), y en un último paso añadiremos muelles en las diagonales, ajustados para minimizar la distorsión Oddy.

El modelo físico elegido es de barras biarticuladas en sus extremos y sometidas exclusivamente a esfuerzo axil, con sección y material constantes, y con comportamiento elástico y lineal.

Dada una barra i cualquiera de entre las n barras conectadas a un nudo $\boldsymbol{P}$ (siendo $\boldsymbol{P}$ el nudo en la posición previa al postproceso de suavizado), consideramos los siguientes datos (figura 28):

- Sea $\boldsymbol{P_i}$ el extremo inicial de la barra i.
- Sea $\boldsymbol{v_i} := \boldsymbol{P_iP}$ el vector director de la barra i antes del suavizado, con origen en $\boldsymbol{P_i}$ y extremo final en $\boldsymbol{P}$. Por tanto, $\|\boldsymbol{v_i}\|$ es la longitud de la barra i antes del suavizado.
- Sea L_i la longitud ideal de la barra i, obtenida del tamaño deseado de elementos en su entorno próximo del dominio de mallado. Es necesariamente no nula y positiva.
- Sea N_i el esfuerzo axil que posee la barra i a causa de la deformación impuesta $\|\boldsymbol{v_i}\| - L_i$ que sufre en la posición previa al suavizado (deformación impuesta que le impide tener su longitud deseada L_i).
- Sea $\boldsymbol{P_S}$ la posición del nudo $\boldsymbol{P}$ después del suavizado.
- Sea $\boldsymbol{t} := \boldsymbol{PP_S}$ el vector que transforma el nudo $\boldsymbol{P}$ en su posición suavizada $\boldsymbol{P_S}$.
- Sea E el módulo de deformación longitudinal de todas las barras.
- Sea A el área de la sección transversal de todas las barras.

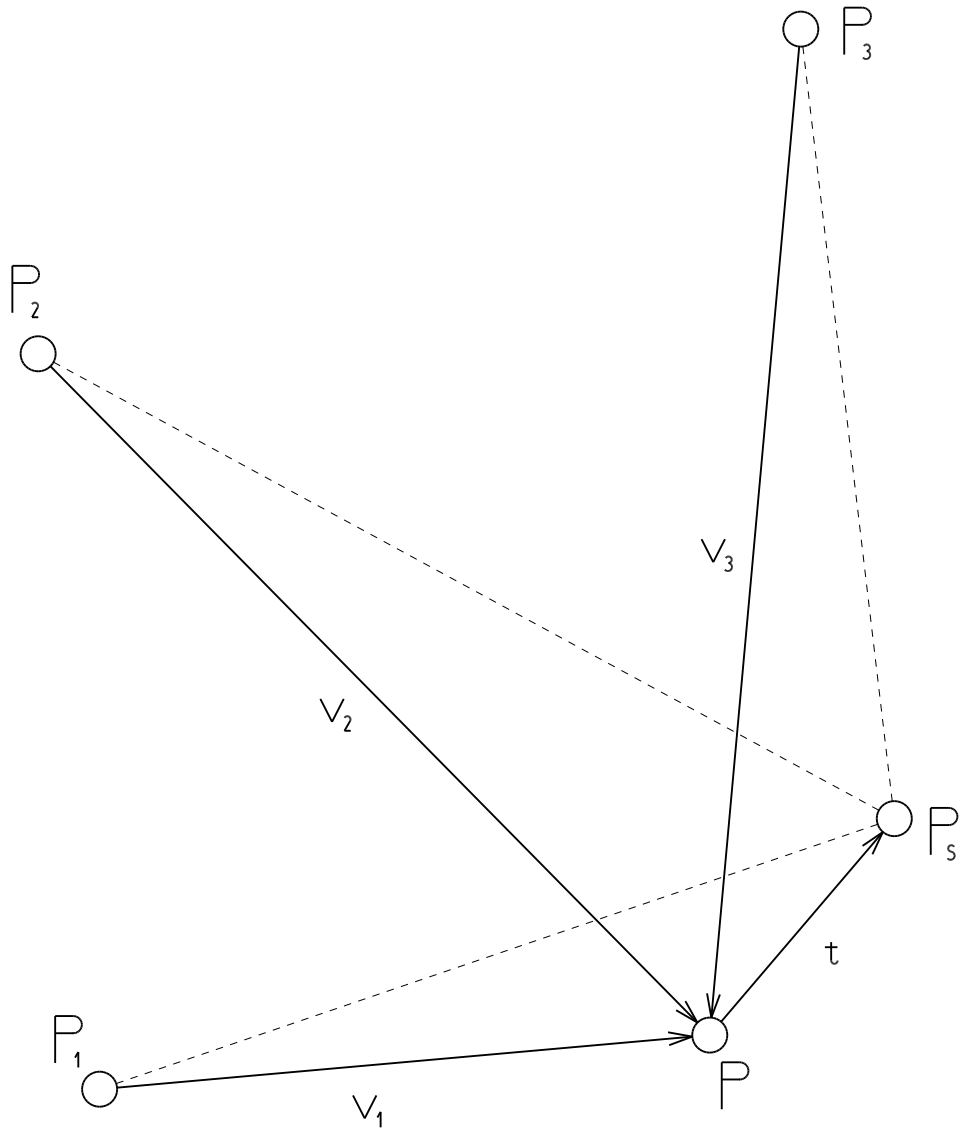


Figura 28: Equilibrio de muelles en un nudo.

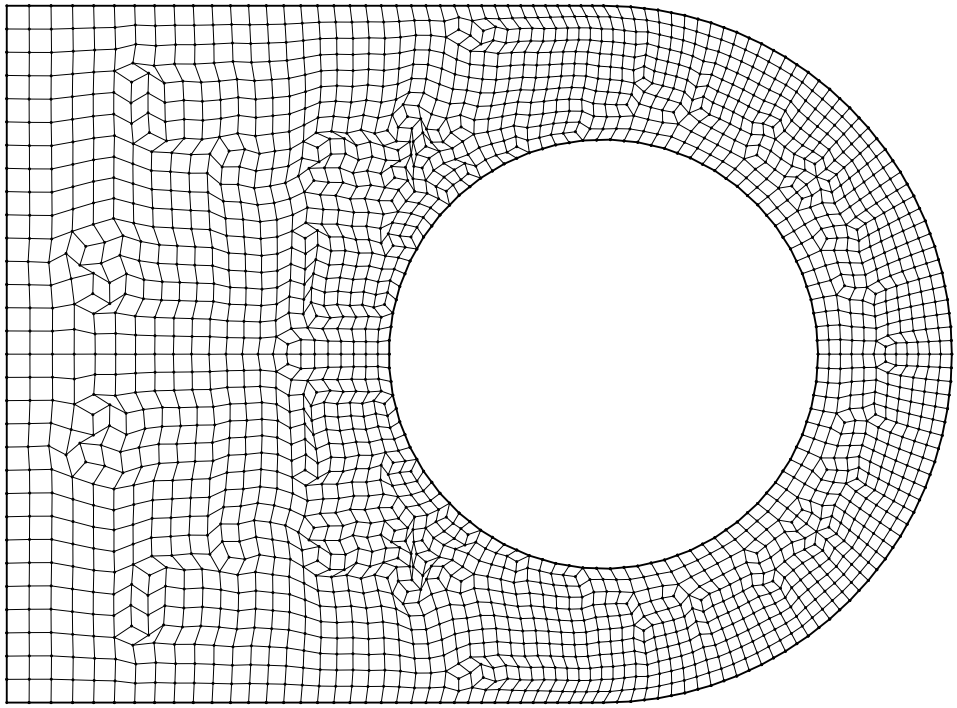


Figura 29: Solución sin diagonales y con comportamiento lineal. Resulta inaceptable, por la degeneración de algunos cuadriláteros y por la elevada distorsión de otros.

Como las barras son biarticuladas y además las cargas están aplicadas exclusivamente en los nudos[32], sólo habrá esfuerzo axil. No hay por tanto necesidad de modelizar el comportamiento de la barra a flexión o a corte, pues dichos esfuerzos son nulos.

A partir de la expresión de la deformación longitudinal de una barra sometida a esfuerzo axil con comportamiento elástico y lineal:

$$\|\boldsymbol{v_i}\| - L_i = \frac{N_i L_i}{EA} \tag{36}$$

por lo que el axil de la barra antes del suavizado es:

$$N_i = \frac{(\|\boldsymbol{v_i}\| - L_i)EA}{L_i} \tag{37}$$

y análogamente, el axil después de suavizar será igual a:

$$N_{S,i} = \frac{(\|\boldsymbol{v_i} + \boldsymbol{t}\| - L_i)EA}{L_i} \tag{38}$$

El vector director unitario de la barra después del suavizado será:

$$\boldsymbol{u_{S,i}} = \frac{\boldsymbol{v_i} + \boldsymbol{t}}{\|\boldsymbol{v_i} + \boldsymbol{t}\|} \tag{39}$$

[32] Las cargas son en realidad movimientos impuestos de traslación en los nudos.

y la fuerza generada por la barra en el nudo después de suavizar $\boldsymbol{P_S}$ será:

$$\boldsymbol{F_i} = \boldsymbol{u_{S,i}} \cdot N_{S,i} = \frac{\boldsymbol{v_i} + \boldsymbol{t}}{\|\boldsymbol{v_i} + \boldsymbol{t}\|} \cdot \frac{(\|\boldsymbol{v_i} + \boldsymbol{t}\| - L_i)EA}{L_i} \tag{40}$$

El equilibro de fuerzas en el nudo suavizado $\boldsymbol{P_S}$ exige que:

$$\sum_{i=1}^{n} \boldsymbol{F_i} = \boldsymbol{0} \tag{41}$$

y desarrollado conduce a:

$$EA \sum_{i=1}^{n} \frac{\boldsymbol{v_i} + \boldsymbol{t}}{\|\boldsymbol{v_i} + \boldsymbol{t}\|} \cdot \frac{\|\boldsymbol{v_i} + \boldsymbol{t}\| - L_i}{L_i} = \boldsymbol{0} \tag{42}$$

que es un sistema de ecuaciones en $\boldsymbol{t}$.

Para implementar su resolución mediante método *Newton-Raphson*, observamos que equivale a hallar las raíces de la función

$$f(\boldsymbol{t}) = \boldsymbol{0} \tag{43}$$

donde

$$f(\boldsymbol{t}) = \sum_{i=1}^{n} \frac{\boldsymbol{v_i} + \boldsymbol{t}}{\|\boldsymbol{v_i} + \boldsymbol{t}\|} \cdot \frac{\|\boldsymbol{v_i} + \boldsymbol{t}\| - L_i}{L_i} \tag{44}$$

resultando las componentes de la matriz Jacobiana iguales a

$$\begin{aligned}
\frac{\partial f_1(\boldsymbol{t})}{\partial t^x} &= \sum_{i=1}^{n} \frac{-L_i(v_i^y + t^y)^2 + \|\boldsymbol{v_i} + \boldsymbol{t}\|^3}{L_i\|\boldsymbol{v_i} + \boldsymbol{t}\|^3} \\
\frac{\partial f_1(\boldsymbol{t})}{\partial t^y} &= \frac{\partial f_2(\boldsymbol{t})}{\partial t^x} = \sum_{i=1}^{n} \frac{(v_i^x + t^x)(v_i^y + t^y)}{\|\boldsymbol{v_i} + \boldsymbol{t}\|^3} \\
\frac{\partial f_2(\boldsymbol{t})}{\partial t^y} &= \sum_{i=1}^{n} \frac{-L_i(v_i^x + t^x)^2 + \|\boldsymbol{v_i} + \boldsymbol{t}\|^3}{L_i\|\boldsymbol{v_i} + \boldsymbol{t}\|^3}
\end{aligned} \tag{45}$$

siendo

$$(v_i^x, v_i^y) = \boldsymbol{v_i} \quad ; \quad (t^x, t^y) = \boldsymbol{t}$$

Con esta formulación, la solución al sistema (42), de manera iterativa sobre todos los nudos interiores del mallado, propociona el resultado de la figura 29, que dista mucho del deseado (incluso se han degenerado algunos cuadriláteros).

Las razones de que este sistema de ecuaciones no nos conduzca al resultado deseado son dos. Primero, no hemos introducido diagonales, por lo que no tenemos control efectivo sobre la distorsión de los cuadriláteros. Segundo, el comportamiento lineal de las barras ocasiona que si una barra dista mucho de alcanzar su longitud deseada pero está conectada a otras muchas que se oponen a ella, no será capaz de deformarlas lo necesario para recuperar parte de su tamaño deseado.

Además de la necesidad de añadir las diagonales, se hace imprescindible introducir en el modelo un comportamiento no lineal de las barras para que, conforme mayor sea la deformación, la fuerza axil de la barra crezca cada vez más rápido, de manera que una barra muy deformada pueda desarrollar la fuerza suficiente para deformar un número elevado de barras que se opongan a ella.

3.4.4.2. Sin diagonales y comportamiento no lineal

Por el momento vamos a continuar el desarrollo sin diagonales, mejorando primero el comportamiento de las barras de manera que sea no lineal, por la razón recién expuesta.

Introduciremos el comportamiento no lineal en el módulo de deformación longitudinal, que ahora ya no será un valor igual para todas las barras, y pasará a ser

$$E_i(\boldsymbol{t}) = 1 + e^{c\left(\left|1-\frac{L_i}{\|\boldsymbol{v_i}+\boldsymbol{t}\|}\right|-d\right)} \tag{46}$$

donde c y d son parámetros que permiten ajustar la función exponencial como se estime necesario (en la implementación en *software* de toda la formulación que sigue se ha tomado $c = 1$ y $d = 0$, proporcionando resultados óptimos, pero se han mantenido los parámetros en el desarrollo por la posibilidad de ajuste que ofrecen).

Véase que definiendo $E_i(\boldsymbol{t})$ de esta manera, el módulo de deformación crece exponencialmente según aumente el valor absoluto de la deformación impuesta de la barra. Es decir, el axil de una barra crecerá más rápido conforme mayor sea su deformación, que es justo el comportamiento que necesitamos para que una barra muy deformada sea capaz de deformar un número elevado de barras que se oponen a ella.

La función de la cual necesitamos obtener las raíces es ahora

$$f(\boldsymbol{t}) = \sum_{i=1}^{n} \frac{\boldsymbol{v_i}+\boldsymbol{t}}{\|\boldsymbol{v_i}+\boldsymbol{t}\|} \cdot \frac{(\|\boldsymbol{v_i}+\boldsymbol{t}\| - L_i) \cdot E_i(\boldsymbol{t})}{L_i} \tag{47}$$

tomando $E_i(\boldsymbol{t})$ el valor de la expresión (46).

Y las componentes de la matriz Jacobiana de la nueva función quedan ahora:

$$\frac{\partial f_1(\boldsymbol{t})}{\partial t^x} = \sum_{i=1}^{n} \frac{\left[\|\boldsymbol{v_i}+\boldsymbol{t}\|^3 + r - L_i\left(v_i^y + t^y\right)^2\right]\cdot E_i(\boldsymbol{t}) - r}{L_i\|\boldsymbol{v_i}+\boldsymbol{t}\|^3}$$

$$\frac{\partial f_1(\boldsymbol{t})}{\partial t^y} = \frac{\partial f_2(\boldsymbol{t})}{\partial t^x} = \sum_{i=1}^{n} \frac{\left[E_i(\boldsymbol{t}) + s(E_i(\boldsymbol{t})-1)\right](v_i^x + t^x)(v_i^y + t^y)}{\|\boldsymbol{v_i}+\boldsymbol{t}\|^3} \qquad (48)$$

$$\frac{\partial f_2(\boldsymbol{t})}{\partial t^y} = \sum_{i=1}^{n} \frac{\left[\|\boldsymbol{v_i}+\boldsymbol{t}\|^3 + q - L_i\left(v_i^x + t^x\right)^2\right]\cdot E_i(\boldsymbol{t}) - q}{L_i\|\boldsymbol{v_i}+\boldsymbol{t}\|^3}$$

donde

$$s = c\left|1 - \frac{L_i}{\|\boldsymbol{v_i}+\boldsymbol{t}\|}\right| \quad ; \quad q = L_i s(v_i^y + t^y)^2 \quad ; \quad r = L_i s(v_i^x + t^x)^2$$

3.4.4.3. Longitud ideal de las diagonales

Introducir las diagonales en el modelo requiere conocer su longitud ideal, para poder evaluar qué deformación impuesta que poseen, y así considerarlas en el equilibrio de fuerzas que estamos planteando.

Dado un cuadrilátero de vértices $\boldsymbol{H}, \boldsymbol{J}, \boldsymbol{D}, \boldsymbol{K}$ (ordenados en sentido antihorario, figura 30) deseamos hallar la longitud de la diagonal $\boldsymbol{HD}$ para que la distorsión de Oddy del cuadrilátero sea mínima, dejando fijos los vértices $\boldsymbol{J}, \boldsymbol{D}, \boldsymbol{K}$ y moviendo $\boldsymbol{H}$ en la dirección de $\boldsymbol{HD}$. El movimiento de $\boldsymbol{H}$ lo expresamos como $\boldsymbol{P} = \boldsymbol{H} + m \cdot \boldsymbol{HD}$, y puede realizarse tanto en el sentido $\boldsymbol{HD}$ como en el contrario (según sea el signo de m).

Recordemos que, de acuerdo con la expresión (34), la distorsión de Oddy del cuadrilátero será la mayor de las que posee localmente en sus cuatro vértices. Como $\boldsymbol{J}, \boldsymbol{D}, \boldsymbol{K}$ permanecen fijos, la distorsión de Oddy en el vértice $\boldsymbol{D}$ no sufre ninguna variación cuando movemos el vértice $\boldsymbol{H}$, porque el ángulo entre los lados $\boldsymbol{JD}$ y $\boldsymbol{DK}$ queda constante.

Por ello, vamos a prescindir de la distorsión de Oddy en $\boldsymbol{D}$, y la consideraremos sólo en los vértices $\boldsymbol{H}, \boldsymbol{J}, \boldsymbol{K}$, en los cuales sí que variará cuando trasladamos $\boldsymbol{H}$, por modificarse los ángulos entre los lados que los unen.

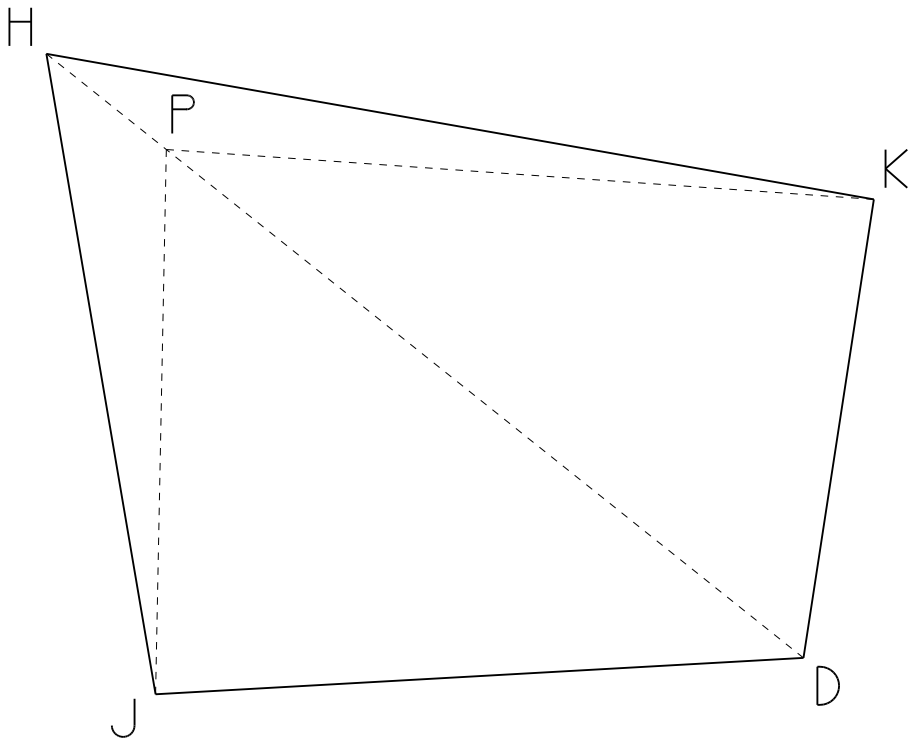


Figura 30: Cuadrilátero con vértices $\boldsymbol{H}, \boldsymbol{J}, \boldsymbol{D}, \boldsymbol{K}$ en sentido antihorario. El punto $\boldsymbol{P}$ indica la nueva posición de $\boldsymbol{H}$ sobre la diagonal $\boldsymbol{HD}$.

Como veremos en breve, la variación de la distorsión Oddy en cada uno de los vértices $\boldsymbol{H}, \boldsymbol{J}, \boldsymbol{K}$ conforme trasladamos $\boldsymbol{H}$ en la dirección de $\boldsymbol{HD}$ es una curva.

Es decir, tendremos tres curvas de la variación de la distorsión Oddy, una para cada uno de dichos vértices. Cada una de ellas podrá tener un mínimo factible o inalcanzable (en algunos casos el mínimo puede conducir a una geometría absurda). Pero además, como hemos definido la distorsión del cuadrilátero como el máximo de las distorsiones locales en los vértices, necesitamos calcular la intersección de las tres curvas, para poder tomar la envolvente del máximo.

En definitiva, la posición ideal para el vértice $\boldsymbol{H}$ tendrá que coincidir exactamente con alguno de estos puntos:

- Un mínimo local de alguna de las tres curvas.

- Un punto de intersección entre ellas.

En la figura 31 se presenta un ejemplo, en el cual el mínimo de la distorsión Oddy ocurre en la intersección entre las curvas $\boldsymbol{J}$ y $\boldsymbol{K}$. Este ejemplo ilustra bien lo que se acaba de exponer: se hace necesario hallar los mínimos de las tres curvas, así como las intersecciones entre ellas, para poder encontrar el mínimo de la distorsión del cuadrilátero y, por tanto, la posición ideal para el vértice $\boldsymbol{H}$ dentro de la diagonal $\boldsymbol{HD}$.

El planteamiento del problema consiste en buscar el punto $\boldsymbol{P}$, que es la nueva posición de $\boldsymbol{H}$ sobre la diagonal $\boldsymbol{HD}$ tal que proporciona el menor valor de distorsión al cuadrilátero.

- Sea $\boldsymbol{HD} = (HD^x, HD^y)$ el vector director de la diagonal, con origen en $\boldsymbol{H}$ y extremo en $\boldsymbol{D}$.

- Sea $\boldsymbol{HJ} = (HJ^x, HJ^y)$ el vector director del lado con origen en $\boldsymbol{H}$ y extremo en $\boldsymbol{J}$.

- Sea $\boldsymbol{HK} = (HK^x, HK^y)$ el vector director del lado con origen en $\boldsymbol{H}$ y extremo en $\boldsymbol{K}$.

- Sea $\boldsymbol{JD} = (JD^x, JD^y)$ el vector director del lado con origen en $\boldsymbol{J}$ y extremo en $\boldsymbol{D}$.

- Sea $\boldsymbol{KD} = (KD^x, KD^y)$ el vector director del lado con origen en $\boldsymbol{K}$ y extremo en $\boldsymbol{D}$.

- Sea $\boldsymbol{JK} = (JK^x, JK^y)$ el vector con origen en $\boldsymbol{J}$ y extremo en $\boldsymbol{K}$.

- Sea $\boldsymbol{PJ} = (PJ^x, PJ^y)$ el vector con origen en el punto buscado $\boldsymbol{P}$ y extremo en $\boldsymbol{J}$.

- Sea $\boldsymbol{PK} = (PK^x, PK^y)$ el vector con origen en el punto buscado $\boldsymbol{P}$ y extremo en $\boldsymbol{K}$.

La solución viene dada al determinar el escalar m en la expresión:

$$\boldsymbol{P} = \boldsymbol{H} + m \cdot \boldsymbol{HD} \tag{49}$$

debiendo hallar los valores candidatos de m de entre los mínimos locales de las curvas de variación de la distorsión y las intersecciones entre ellas, para después elegir el candidato que proporcione una distorsión mínima del cuadrilátero.

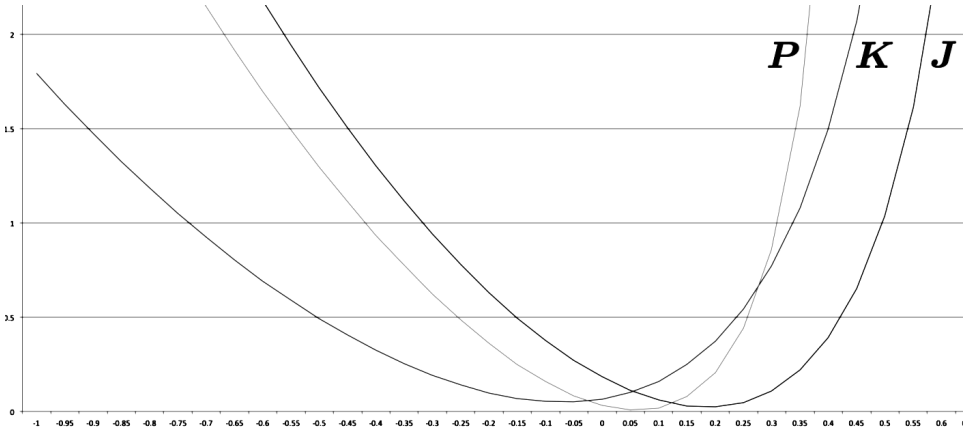


Figura 31: Variación de la distorsión Oddy en los vértices $\boldsymbol{P}, \boldsymbol{J}, \boldsymbol{K}$ cuando movemos $\boldsymbol{H}$ en la dirección $\boldsymbol{HD}$. En este ejemplo, el mínimo de la función máximo se produce en la intersección entre las curvas $\boldsymbol{J}$ y $\boldsymbol{K}$. El eje de abscisas es el escalar m de la ecuación (49).

A partir de la expresión (33), podemos evaluar la distorsión local de Oddy en cada uno de los vértices $\boldsymbol{P}, \boldsymbol{J}, \boldsymbol{K}$ en función del escalar m, es decir, en función de la traslación de $\boldsymbol{P}$ a lo largo de la diagonal. Desarrollando llegamos a estas expresiones, que son las ecuaciones de las tres curvas que se grafían en la figura 31:

$$D_{Oddy,P}(m) = 2\left[\frac{\left[(HJ^x - mHD^x)^2 + (HK^x - mHD^x)^2 + (HJ^y - mHD^y)^2 + (HK^y - mHD^y)^2\right]^2}{4[m[HD^x(HJ^y - HK^y) + HD^y(HK^x - HJ^x)] + HJ^x HK^y - HJ^y HK^x]^2} - 1\right] \quad (50)$$

$$D_{Oddy,J}(m) = 2\left[\frac{\left[(HJ^x - mHD^x)^2 + (HJ^y - mHD^y)^2 + \|\boldsymbol{JD}\|^2\right]^2}{4(-mHD^x JD^y + mHD^y JD^x - JD^x HJ^y + JD^y HJ^x)^2} - 1\right] \quad (51)$$

$$D_{Oddy,K}(m) = 2\left[\frac{\left[(HK^x - mHD^x)^2 + (HK^y - mHD^y)^2 + \|\boldsymbol{KD}\|^2\right]^2}{4(mHD^x KD^y - mHD^y KD^x + KD^x HK^y - KD^y HK^x)^2} - 1\right] \quad (52)$$

Para averiguar los valores candidatos de m correspondientes a la intersección de las curvas de variación de la distorsión local en $\boldsymbol{J}$ y $\boldsymbol{K}$, hemos de resolver la ecuación

$$D_{Oddy,J}(m) = D_{Oddy,K}(m) \quad (53)$$

que, tras desarrollarla con las expresiones (51) y (52), equivale a hallar las raíces de una ecuación cúbica de la forma

$$e_{JK}m^3 + f_{JK}m^2 + g_{JK}m + h_{JK} = 0 \quad (54)$$

donde

$$e_{JK} = \|\boldsymbol{HD}\|^2 (s - r)$$

$$f_{JK} = 2(br - as) + \|\boldsymbol{HD}\|^2 (t - u)$$
$$g_{JK} = 2(bu - at) - dr + cs$$
$$h_{JK} = ct - du$$
$$a = HD^x HJ^x + HD^y HJ^y$$
$$b = HD^x HK^x + HD^y HK^y$$
$$c = \|\boldsymbol{JD}\|^2 + \|\boldsymbol{HJ}\|^2$$
$$d = \|\boldsymbol{KD}\|^2 + \|\boldsymbol{HK}\|^2$$
$$r = HD^y JD^x - HD^x JD^y$$
$$s = HD^x KD^y - HD^y KD^x$$
$$t = KD^x HK^y - KD^y HK^x$$
$$u = JD^y HJ^x - JD^x HJ^y$$

ecuación que puede tener tres raíces reales y distintas, tres raíces reales múltiples, o una raíz real y dos imaginarias. La solución algebraica de ecuaciones cúbicas se recordará en el subapartado siguiente.

Análogamente, podemos hallar los valores candidatos de m que proceden de la intersección de las curvas de $\boldsymbol{J}$ y $\boldsymbol{P}$, resolviendo la ecuación

$$D_{Oddy,J}(m) = D_{Oddy,P}(m) \tag{55}$$

que nos obliga a hallar las raíces de una nueva ecuación cúbica

$$e_{JP}m^3 + f_{JP}m^2 + g_{JP}m + h_{JP} = 0 \tag{56}$$

donde

$$e_{JP} = \|\boldsymbol{HD}\|^2 (z - 2r)$$
$$f_{JP} = 2(ar + br - az) + \|\boldsymbol{HD}\|^2 (w - 2u)$$
$$g_{JP} = 2(au + bu - aw) + cz - r\left(\|\boldsymbol{HJ}\|^2 + \|\boldsymbol{HK}\|^2\right)$$
$$h_{JP} = cw - u\left(\|\boldsymbol{HJ}\|^2 + \|\boldsymbol{HK}\|^2\right)$$
$$w = HJ^x HK^y - HJ^y HK^x$$
$$z = HD^y JK^x - HD^x JK^y$$

tomando a, b, c, r, u los mismos valores que en la ecuación (54).

Y añadimos también los valores candidatos de m debidos a la intersección de las curvas de $\boldsymbol{K}$ y $\boldsymbol{P}$:

$$D_{Oddy,K}(m) = D_{Oddy,P}(m) \tag{57}$$

que resulta en la ecuación cúbica

$$e_{KP}m^3 + f_{KP}m^2 + g_{KP}m + h_{KP} = 0 \tag{58}$$

donde

$$\begin{aligned}
e_{KP} &= \|\boldsymbol{HD}\|^2 (z - 2s) \\
f_{KP} &= 2(as + bs - bz) + \|\boldsymbol{HD}\|^2 (w - 2t) \\
g_{KP} &= 2(at + bt - bw) + dz - s\left(\|\boldsymbol{HJ}\|^2 + \|\boldsymbol{HK}\|^2\right) \\
h_{KP} &= dw - t\left(\|\boldsymbol{HJ}\|^2 + \|\boldsymbol{HK}\|^2\right)
\end{aligned}$$

tomando a, b, d, s, t, w, z los mismos valores que en las ecuaciones (54) y (56).

Se observa que las ecuaciones (56) y (58) presentan expresiones simétricas entre sí, mientras que la ecuación (54) es notablemente diferente. Esto era de esperar porque (56) y (58) son la intersección entre las curvas de la distorsión en $\boldsymbol{P}$ (punto móvil) y en el vértice (fijo) que queda a un lado de la diagonal ($\boldsymbol{J}$ y $\boldsymbol{K}$, respectivamente), mientras que (54) es la intersección de las curvas de distorsión en los dos vértices fijos $\boldsymbol{J}$ y $\boldsymbol{K}$.

De momento, teniendo en cuenta sólo los puntos de intersección entre las curvas de la variación de la distorsión, disponemos de un máximo de 9 valores candidatos de m (si las ecuaciones (54), (56) y (58) tienen tres raíces reales distintas cada una), y un mínimo de 3 (si sólo hay tres raíces reales en total).

En cualquier caso, nos falta todavía considerar los valores candidatos de m que proceden de los mínimos de cada curva. Lo hacemos a continuación:

Si $a^2 - c\|\boldsymbol{HD}\|^2 \geqslant 0$, la curva $D_{Oddy,J}(m)$ alcanza extremos locales en:

$$m = \frac{a \pm \sqrt{a^2 - c\|\boldsymbol{HD}\|^2}}{\|\boldsymbol{HD}\|^2} \tag{59}$$

en caso contrario, los alcanza en:

$$m = 1 \pm \frac{\sqrt{-2a + c + \|\boldsymbol{HD}\|^2}}{\|\boldsymbol{HD}\|} \tag{60}$$

Si $b^2 - d\left\|\boldsymbol{HD}\right\|^2 \geqslant 0$, la curva $D_{Oddy,K}(m)$ alcanza extremos locales en:

$$m = \frac{b \pm \sqrt{b^2 - d\left\|\boldsymbol{HD}\right\|^2}}{\left\|\boldsymbol{HD}\right\|^2} \tag{61}$$

en caso contrario, los alcanza en:

$$m = 1 \pm \frac{\sqrt{-2b + d + \left\|\boldsymbol{HD}\right\|^2}}{\left\|\boldsymbol{HD}\right\|} \tag{62}$$

Si $(a+b)^2 - 2\left\|\boldsymbol{HD}\right\|^2 \left(\left\|\boldsymbol{HJ}\right\|^2 + \left\|\boldsymbol{HK}\right\|^2\right) \geqslant 0$, la curva $D_{Oddy,P}(m)$ alcanza extremos locales en:

$$m = \frac{a + b \pm \sqrt{(a+b)^2 - 2\left\|\boldsymbol{HD}\right\|^2 \left(\left\|\boldsymbol{HJ}\right\|^2 + \left\|\boldsymbol{HK}\right\|^2\right)}}{2\left\|\boldsymbol{HD}\right\|^2} \tag{63}$$

en caso contrario, los alcanza en:

$$m = \frac{-2w \pm z\sqrt{\frac{2z\left[2w(a+b)+z\left(\left\|\boldsymbol{HJ}\right\|^2+\left\|\boldsymbol{HK}\right\|^2\right)\right]+4\left\|\boldsymbol{HD}\right\|^2 w^2}{\left\|\boldsymbol{HD}\right\|^2 z^2}}}{2z} \tag{64}$$

Recapitulando, ya estamos en condiciones de hallar la longitud ideal de la diagonal. De las ecuaciones (54), (56), (58), (59), (60), (61), (62), (63) y (64) obtenemos un conjunto de hasta 15 valores candidatos de m (9 de las intersecciones entre curvas si todas las raíces son reales y distintas, y 6 de la búsqueda de mínimos).

Para cada candidato m_i encontrado, evaluamos

$$D^{Sol}_{Oddy}(m_i) = \text{máx}\left\{D_{Oddy,J}(m_i); D_{Oddy,K}(m_i); D_{Oddy,P}(m_i)\right\} \tag{65}$$

y elegimos aquel candidato m_i cuyo valor $D^{Sol}_{Oddy}(m_i)$ sea menor que el resto.

Al sustituir el valor m_i elegido en la ecuación (49), se obtiene $\boldsymbol{P}$, y por tanto la dimensión ideal de la diagonal tal que la distorsión de Oddy del cuadrilátero es mínima.

3.4.4.3.1. Solución de ecuaciones cúbicas

Para obtener las raíces de las ecuaciones (54), (56) y (58) se ha encontrado conveniente la solución en forma cerrada conocida como *método de Cardano*, que se resume a continuación.

Este procedimiento fue publicado por primera vez por Gerolamo Cardano (1501-1576) en su obra Ars Magna [7], si bien la solución se debe a Niccolò Tartaglia, y probablemente a Scipione del Ferro, que no llegaron a publicarla.

Dada una ecuación cúbica en su forma general

$$em^3 + fm^2 + gm + h = 0 \tag{66}$$

tenemos garantía de que $e \neq 0$ (en caso contrario no sería una ecuación cúbica). Por tanto es posible llegar a la forma reducida de la ecuación dividiendo por e:

$$m^3 + a_2 m^2 + a_1 m + a_0 = 0 \tag{67}$$

donde

$$a_2 = \frac{f}{e} \quad ; \quad a_1 = \frac{g}{e} \quad ; \quad a_0 = \frac{h}{e} \tag{68}$$

Sea $D = Q^3 + R^2$, donde

$$Q = \frac{3a_1 - a_2^2}{9} \quad ; \quad R = \frac{9a_1 a_2 - 27a_0 - 2a_2^3}{54} \tag{69}$$

Si $D = 0$ y $Q = 0$ hay una raíz real triple:

$$m_1 = m_2 = m_3 = -\frac{a_2}{3} \tag{70}$$

Si $D = 0$ y $Q \neq 0$ hay una raíz real doble y una simple:

$$m_1 = m_2 = \frac{a_1 a_2 - 9a_0}{18Q} \tag{71}$$

$$m_3 = \frac{a_2^3 - 4a_1 a_2 + 9a_0}{9Q}$$

Si $D > 0$, la ecuación cúbica tiene una raíz real y dos imaginarias, siendo la raíz real igual a

$$m_1 = \sqrt[3]{R + \sqrt{D}} + \sqrt[3]{R - \sqrt{D}} - \frac{a_2}{3} \tag{72}$$

Si $D < 0$ las tres raíces son reales, pero se requiere una incursión en los números complejos para hallarlas *(casus irreducibilis)*, a pesar de que sus valores son reales. No obstante, con la interpretación geométrica de Françoise Viète (1540-1603) y René Descartes (1596-1650) se pueden expresar las tres raíces reales en forma trigonométrica sin necesidad de emplear números complejos [37]:

$$\begin{aligned} m_1 &= 2\sqrt{-Q}\cos\left(\frac{\theta}{3}\right) - \frac{a_2}{3} \\ m_2 &= 2\sqrt{-Q}\cos\left(\frac{\theta+2\pi}{3}\right) - \frac{a_2}{3} \\ m_3 &= 2\sqrt{-Q}\cos\left(\frac{\theta+4\pi}{3}\right) - \frac{a_2}{3} \end{aligned} \tag{73}$$

donde

$$\theta = \arccos\left(\frac{R}{\sqrt{-Q^3}}\right) \tag{74}$$

teniendo asegurado que $Q < 0$ cuando $D < 0$.

La solución de ecuaciones cúbicas siguiendo el *método de Cardano* se puede por tanto implementar en *software* de manera sencilla.

3.4.4.4. Diagonales con comportamiento no lineal

Conociendo el tamaño ideal de las diagonales, que acabamos de obtener, podemos introducir en el modelo de muelles las barras correspondientes. También las dotaremos de comportamiento no lineal, para que puedan desarrollar una fuerza mayor si se encuentran muy deformadas respecto de su longitud ideal.

Sin embargo, su comportamiento no lineal es aportado automáticamente al formular su módulo de deformación $E_i(\boldsymbol{t})$ en función de la distorsión de Oddy, no siendo necesario emplear la función exponencial como hemos hecho en las barras de los lados del cuadrilatero.

No es necesario hacer ningún cambio a la función cuyas raíces hallamos al hacer equilibrio de fuerzas en un nudo -expresión (47)-, con la salvedad de que L_i debe tomarse como la longitud ideal que hemos hallado, y que el módulo de deformación de las diagonales adopta el valor:

$$E_i(\boldsymbol{t}) = m_d \cdot D_{Oddy}^{cuad,i}(\boldsymbol{t}) + 1 \tag{75}$$

donde $D_{Oddy}^{cuad,i}(\boldsymbol{t})$ es la distorsión Oddy (antes del suavizado) del cuadrilátero al que pertenece la diagonal, y m_d es un escalar para calibrar el compromiso entre la calidad por tamaño y la calidad por distorsión (conforme mayor sea m_d, más repercusión tendrán las diagonales frente a los lados en el resultado final, y viceversa).

En las pruebas experimentales realizadas, se ha observado que el valor $m_d = 0.5$ proporciona un compromiso satisfactorio y resultados óptimos. Valores menores producen menores errores en el tamaño de los elementos, pero con un aumento de la distorsión. Valores mayores disminuyen ligeramente la distorsión, pero con penalización en el error de tamaño.

$D_{Oddy}^{cuad,i}(\boldsymbol{t})$ es función de $\boldsymbol{t}$ porque recordemos que $\boldsymbol{t}$ es el vector que equilibra el nudo que estamos procesando (expresiones (44) y (47)), transformándolo desde su posición no suavizada a la suavizada. Por tanto, $\boldsymbol{t}$ modifica la distorsión de Oddy de los cuadriláteros conectados a dicho nudo, y la distorsión es función de dicho vector.

Con anterioridad hemos mencionado que los lados del cuadrilátero tienen una mayor repercusión en su tamaño, y las diagonales en su distorsión, pero que no obstante no era posible establecer una división completa de la repercusión de cada uno. Las diagonales también influyen en el tamaño de los elementos y, por ello, conviene escalar el valor L_i por un factor igual a la relación entre el tamaño que el cuadrilátero debería tener y el que actualmente tiene. De esta manera, la diagonal ayuda a los lados a alcanzar el tamaño ideal, al tiempo que también ayuda a minimizar la distorsión.

Hay que recordar aquí que hemos decidido usar la formulación original de la distorsión de Oddy. Por ello, hemos de considerar el signo del área del paralelogramo local al calcular la distorsión en cada vértice del cuadrilátero.

Es decir, si al evaluar (33) obtenemos un área nula o negativa, en lugar de continuar calculando la distorsión en ese vértice, le daremos un valor arbitrariamente grande. De esta manera, si un cuadrilátero está degenerado, el valor de $E_i(\boldsymbol{t})$ de sus diagonales será muy grande, por la gran necesidad de que dichas diagonales alcancen una longitud cercana a su ideal, que solucione la degeneración del cuadrilátero.

Como detectamos los casos de area nula o negativa, evitamos dar por buena la distorsión de cuadriláteros degenerados, así como que aparezcan asíntotas debidas a divisiones por un área nula.

Aunque no hemos modificado la función de equilibrio de fuerzas en el nudo (47), recordemos que los elementos de la matriz Jacobiana los teníamos

expresados como sumatorio de la contribución de cada barra -ver expresión (48)- y en ellos se había desarrollado la derivada de la función exponencial en que consistía el módulo de deformación $E_i(\boldsymbol{t})$.

Como ahora en las diagonales tenemos otro módulo de deformación, los sumandos correspondientes a diagonales en elementos de la matriz Jacobiana adoptan el siguiente valor:

$$\begin{aligned}
\frac{\partial f_1(\boldsymbol{t})}{\partial t^x} &= \sum_{i=1}^{nd} \frac{\left(t^x + v_i^x\right)\left(n_{tv} - L_i\right) d_x n_{tv}^2 + E_i(\boldsymbol{t}) \cdot \left[n_{tv}^3 - L_i \left(t^y + v_i^y\right)^2\right]}{L_i n_{tv}^3} \\
\frac{\partial f_1(\boldsymbol{t})}{\partial t^y} &= \sum_{i=1}^{nd} \frac{\left(t^x + v_i^x\right)\left[\left(n_{tv} - L_i\right) d_y n_{tv}^2 + E_i(\boldsymbol{t}) \cdot L_i \left(t^y + v_i^y\right)\right]}{L_i n_{tv}^3} \\
\frac{\partial f_2(\boldsymbol{t})}{\partial t^x} &= \sum_{i=1}^{nd} \frac{\left(t^y + v_i^y\right)\left[\left(n_{tv} - L_i\right) d_x n_{tv}^2 + E_i(\boldsymbol{t}) \cdot L_i \left(t^x + v_i^x\right)\right]}{L_i n_{tv}^3} \\
\frac{\partial f_2(\boldsymbol{t})}{\partial t^y} &= \sum_{i=1}^{nd} \frac{\left(t^y + v_i^y\right)\left(n_{tv} - L_i\right) d_y n_{tv}^2 + E_i(\boldsymbol{t}) \cdot \left[n_{tv}^3 - L_i \left(t^x + v_i^x\right)^2\right]}{L_i n_{tv}^3}
\end{aligned} \tag{76}$$

donde

$nd =$ nº de diagonales que confluyen en el nudo.

$$d_x = \frac{\partial E_i(\boldsymbol{t})}{\partial t^x} = m_d \frac{\partial D_{Oddy}^{cuad,i}(\boldsymbol{t})}{\partial t^x}$$

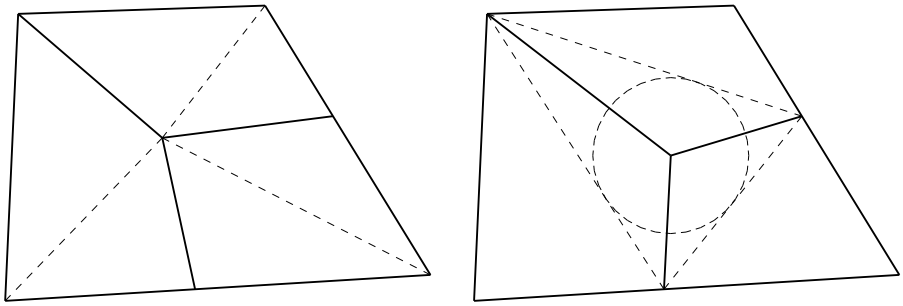
$$d_y = \frac{\partial E_i(\boldsymbol{t})}{\partial t^y} = m_d \frac{\partial D_{Oddy}^{cuad,i}(\boldsymbol{t})}{\partial t^y}$$

$$n_{tv} = \|\boldsymbol{v_i} + \boldsymbol{t}\|$$

La matriz Jacobiana así obtenida para las diagonales se debe sumar a la de los lados de (48), llegando a la matriz Jacobiana total de todas las barras (lados y diagonales).

Obsérvese que aunque $D_{Oddy}^{cuad,i}(\boldsymbol{t})$ no es diferenciable, sin embargo conocemos el punto de la diagonal para la cual la distorsión es mínima (punto $\boldsymbol{P}$, hallado en el apartado anterior), por lo que podemos obtener una aproximación a la derivada de la siguiente manera:

$$\frac{\partial D_{Oddy}^{cuad,i}(\boldsymbol{t})}{\partial \boldsymbol{HD}} \approx \frac{D_{Oddy}^{ideal,i}(\boldsymbol{t}) - D_{Oddy}^{cuad,i}(\boldsymbol{t})}{\|\boldsymbol{HP}\|} \tag{77}$$



(a) (b)

Figura 32: Nudo con sólo tres cuadriláteros. Estos nudos pueden tener una estabilidad numérica delicada debido a que sólo confluyen en ellos tres diagonales, y dos de ellas tienden a orientarse de manera colineal.

$$\frac{\partial D_{Oddy}^{cuad,i}(\boldsymbol{t})}{\partial t^x} \approx \frac{\partial D_{Oddy}^{cuad,i}(\boldsymbol{t})}{\partial \boldsymbol{HD}} \cdot \frac{P^x - H^x}{\|\boldsymbol{HP}\|} \tag{78}$$

$$\frac{\partial D_{Oddy}^{cuad,i}(\boldsymbol{t})}{\partial t^y} \approx \frac{\partial D_{Oddy}^{cuad,i}(\boldsymbol{t})}{\partial \boldsymbol{HD}} \cdot \frac{P^y - H^y}{\|\boldsymbol{HP}\|} \tag{79}$$

donde $D_{Oddy}^{ideal,i}(\boldsymbol{t})$ es la distorsión Oddy si la diagonal tuviese la longitud ideal $\|\boldsymbol{PD}\|$, $D_{Oddy}^{cuad,i}(\boldsymbol{t})$ es la distorsión Oddy con la longitud actual de la diagonal $\|\boldsymbol{PD}\|$, y los puntos $\boldsymbol{P}$ y $\boldsymbol{H}$ son los del apartado anterior. Si el denominador es nulo, indicaría que $\boldsymbol{H} = \boldsymbol{P}$, es decir, que ya nos encontramos en el punto de solución ideal, y en tal caso tomaremos las derivadas nulas.

Si se ha decidido escalar la longitud ideal de la diagonal como se ha recomendado, se debe afectar de la misma escala al punto $\boldsymbol{P} = (P^x, P^y)$ en las expresiones anteriores.

3.4.4.5. Nudos con tres cuadriláteros

Cuando el dominio tiene zonas con gradación en el tamaño de los elementos, o en las proximidades de perímetros con formas difíciles, es habitual que surjan nudos que sólo conectan tres cuadriláteros (figura 32).

Al plantear equilibrio en estos nudos con el método de los muelles, sólo confluyen en ellos tres diagonales, dos de las cuales tienden a enfrentarse de manera colineal (figura 32a), lo que puede acarrear una estabilidad numérica delicada.

Las soluciones que se han encontrado más convenientes para el equilibrio en estos nudos es, o bien cuidar en extremo la precisión numérica de coma flotante (tanto empleando el tipo de datos de mayor precisión que posea el hardware -preferiblemente mayor que "*double*"-, como reordenando las operaciones para conservar el máximo de dígitos significativos), o bien adoptando una solución alternativa para estos nudos: por ejemplo elegir como posición suavizada del nudo el incentro del triángulo formado por los extremos de las aristas que confluyen en el nudo (figura 32b).

Aunque pueda parecer que tomar este incentro en vez de la solución ideal pueda perjudicar a la calidad final del postproceso, esto no es así (como corraboran los resultados obtenidos). Hay que tener en cuenta que estos puntos son minoría en el mallado, y necesariamente están rodeados por nudos que tengan cuatro o más cuadriláteros (en caso contrario la malla no podría ser cuadrangular), los cuales se suavizarán de manera óptima y por tanto corrigiendo la posible posición no-óptima de los nudos con tres cuadriláteros.

Al final, las mediciones realizadas muestran que prácticamente no hay diferencia de calidad en el mallado entre las opciones de aumentar la precisión numérica para estos nudos, o elegir el incentro. La opción de tomar el incentro es atractiva porque es rápido de cálcular, hay garantía de que cae en el interior del triángulo mencionado, y evita tener que cuidar en extremo la precisión de coma flotante en estos nudos, al mismo tiempo que los nudos vecinos se adecuarán a ello en su equilibrio de muelles.

3.4.4.6. Planteamiento del método propuesto

Una vez expuestas las magnitudes y condiciones en que se basa el método propuesto de postproceso de suavizado, podemos hacer su planteamiento general.

El suavizado se realiza de manera iterativa. En cada iteración, los vértices interiores del mallado se mueven a su posición suavizada. El procedimiento concluye cuando los desplazamientos de los nudos en una iteración son despreciables.

Se alcanza una calidad elevada ya en las primeras iteraciones, por lo que también es posible hacer un suavizado parcial aplicando un número fijo y reducido de iteraciones. No obstante, la calidad del tamaño de los elementos mejora con el número de iteraciones, por lo que si se desea una calidad máxima es preferible continuar iterando hasta que los desplazamientos sean

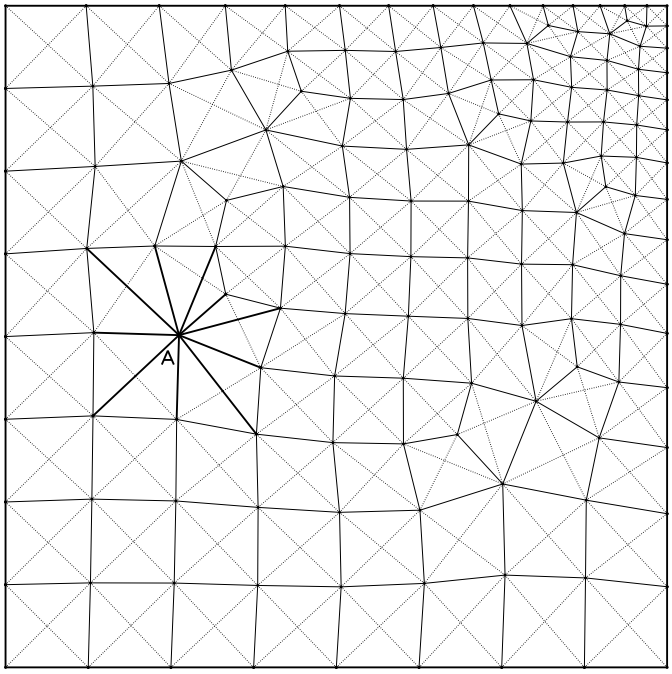


Figura 33: Planteamiento postproceso de suavizado. Las barras que intervienen en el equilibrio del nudo A se han marcado con mayor grosor.

despreciables.

Es posible elegir entre desplazar todos los vértices al finalizar cada iteración (lo que permite paralelizar el proceso) o, si se implementa de manera secuencial, se puede optar por desplazar cada vértice en cuanto se resuelve su equilibrio (esto hace que, dentro de una misma iteración, los vértices restantes tengan en cuenta ya los desplazamientos de los vértices ya equilibrados, lo que reduce el número de iteraciones necesarias). La calidad obtenida es casi idéntica en ambos casos, si bien el resultado puede no ser el mismo (desplazar todos los vértices a la vez conserva la simetría si el mallado es simétrico, mientras que desplazando cada vértice se puede perder la simetría en algunos casos).

En cada iteración, se localizan para cada nudo las barras que están conectadas a él. Por ejemplo, en la figura 33, el nudo **A** es compartido por 5 cuadriláteros, y por tanto hay que plantear su equilibrio mediante 10 barras (5 barras correspondientes a lados de los cuadriláteros, y otras 5 de diagonales conectadas al nudo).

El equilibrio del nudo es la ecuación (43), donde $f(\boldsymbol{t})$ se construye sumando la contribución de todas las barras conectadas al nudo (tanto las

```
1  Postproceso de suavizado mediante muelles
2
3   Entrada: {lista de vértices},
4            {lista de cuadriláteros},
5            {tamaño deseado de elemento (por vértice)}
6   Repetir
7     Para cada P ← vértice interior de {lista de vértices}
8       t ← [0,0]
9         Repetir
10          f ← [0,0]
11          J ← [0,0,0,0]
12          Para cada A ← arista conectada a P
13            L_i ← longitud deseada de A (dato de entrada)
14            f ← f + f(A,L_i,t) —evaluando la ecuación (47)
15            J ← J + J(A,L_i,t) —evaluando la ecuación (48)
16          Para cada D ← diagonal conectada a P
17            m ← mejor candidato m_i de la expresión (65)
18            L_i ← longitud ideal de D según el escalar m
19            (opcional) L_i ← L_i·media(L_i de lados)/media(Long. de lados)
20            f ← f + f(D,L_i,t) —evaluando la ecuación (47) con la (75)
21            J ← J + J(D,L_i,t) —evaluando la ecuación (76)
22          t_nueva ← Resolver el sistema J(t)·(t_nueva − t) = −f(t)
23          paso ← t_nueva − t
24          t ← t_nueva
25        Mientras (norma(f) > umbral_sol) Y (norma(paso) > umbral_paso)
26      P ← P + t
27  Mientras max( norma(t) ) > umbral_suavizado
```

Algoritmo 1: Planteamiento del método propuesto (secuencial).

correspondientes a lados como a diagonales, expresión (47)), con la única particularidad de que $E_i(\boldsymbol{t})$ se toma de (46) en barras de lados y de (75) en barras diagonales, y que además L_i es el tamaño deseado del lado en barras de lados, y el tamaño ideal de la diagonal en barras de diagonales (obtenida del mejor candidato de (65)).

De la ecuación (43) se obtiene el valor de $\boldsymbol{t}$ que transforma el nudo en su posición suavizada. Esta ecuación se puede resolver mediante *Newton-Raphson*, construyendo la matriz Jacobiana como suma de la contribución de las barras de lados -expresión (48)- y de las barras de diagonales -expresión (76).

Operando así en todos los nudos, completamos una iteración, y progresamos hasta que los vectores $\boldsymbol{t}$ de todos los nudos sean despreciables.

El pseudocódigo del procedimiento se presenta en el algoritmo 1 para la versión secuencial en la que cada vértice se desplaza inmediatamente al resolver su equilibrio, y en el algoritmo 2 para la versión paralelizable, desplazando todos los vértices a la vez al finalizar cada iteración.

```
1  Versión desplazando todos los vértices a la vez
2
3   Entrada: {lista de vértices},
4            {lista de cuadriláteros},
5            {tamaño deseado de elemento (por vértice)}
6
7   Ps ← {lista de nueva posición de vértices} ← {lista de vértices}
8
9   Repetir
10    Para cada P ← vértice interior de {lista de vértices}
11      t ← [0,0]
12
13        Repetir
14          f ← [0,0]
15          J ← [0,0,0,0]
16          Para cada A ← arista conectada a P
17            L_i ← longitud deseada de A (dato de entrada)
18            f ← f + f(A, L_i, t) —evaluando la ecuación (47)
19            J ← J + J(A, L_i, t) —evaluando la ecuación (48)
20          Para cada D ← diagonal conectada a P
21            m ← mejor candidato m_i de la expresión (65)
22            L_i ← longitud ideal de D según el escalar m
23            (opcional) L_i ← L_i·media(L_i de lados)/media(Long. de lados)
24            f ← f + f(D, L_i, t) —evaluando la ecuación (47) con la (75)
25            J ← J + J(D, L_i, t) —evaluando la ecuación (76)
26          t_nueva ← Resolver el sistema J(t)·(t_nueva − t) = −f(t)
27          paso ← t_nueva − t
28          t ← t_nueva
29        Mientras (norma(f) > umbral_sol) Y (norma(paso) > umbral_paso)
30
31      Ps[P] ← P + t
32
33    Para cada P ← vértice interior de {lista de vértices}
34      P ← Ps[P]
35
36  Mientras max( norma(t) ) > umbral_suavizado
```

Algoritmo 2: Planteamiento del método propuesto (paralelizable).

3.4.5. Comparativa de resultados con postproceso

Se presenta a continuación una comparativa de los resultados obtenidos con diferentes algoritmos de postproceso de suavizado. Una colección de ejemplos se encuentra en las figuras 34 a 41.

Se han tomado mediciones de la distorsión de Oddy, del error de tamaño de lados (relativo en tanto por cien respecto del deseado), y del error de crecimiento de tamaño (este último sólo en zonas de mallado estructurado, por tener verdadero sentido en aristas colineales consecutivas —caso que ocurre principalmente cuando hay cuatro aristas por nudo, es decir, mallado estructurado).

El error de crecimiento de tamaño se ha medido comparando la relación entre las longitudes de dos aristas colineales consecutivas, y la relación que deberían tener si sus longitudes fueran las deseadas. Se ha expresado como error de ángulo de crecimiento, en grados.

En la tabla 8 se aportan todas estas mediciones, para el mallado original sin postproceso (figuras 34 y 35), el algoritmo original de Giuliani [18] (figuras 36a, 37a, y 38a), el algoritmo de Giuliani con la modificación de Sarrate y Huerta [50] (figuras 36b, 37b, y 38b), el método de minimización del producto de la distorsión por el error de tamaño (reinterpretando Gargallo, Roca y Sarrate [17] con la distorsión Oddy, figuras 39a, 40a, y 41a), así como del método propuesto basado en muelles (figuras 39b, 40b, y 41b).

Los resultados indican que el método propuesto se comporta de manera óptima en los requisitos que exigíamos.

En el apartado de distorsión Oddy, la calidad obtenida es superior a la de los algoritmos basados en Giuliani (la media es similar, pero el percentil 99º de la distorsión se reduce cerca de un 40 % —aspecto importante porque dichos métodos deberían producir una distorsión extraordinariamente baja al no velar por mantener el tamaño deseado de los elementos). La distorsión obtenida con el método propuesto es muy similar a la del método de minimización del producto de la distorsión por el error de tamaño (logra bajar la media de 0.18 a 0.15 pero aumenta ligeramente el percentil 99º, de 0.90 a 1.04).

En el error del tamaño de los lados de elementos consigue el mejor resultado de todos los métodos estudiados, con un error relativo medio del 7.35 %, y logrando que el 75 % de las aristas tengan un error relativo por debajo del 10 %. Los otros métodos no alcanzan estos resultados.

	Distorsión media	Distorsión percentil 99º	Error medio de tamaño	Aristas con error $\leqslant 10\,\%$	Error máximo de crecimiento	Media valor absoluto del error de crecimiento
Sin postproceso	0.31	4.27	8.69 %	68 %	18.98º	0.60º
Postproceso de Giuliani	0.16	1.56	7.77 %	72 %	3.75º	0.39º
Postproceso de Giuliani (cambios de [50])	0.14	1.65	9.33 %	64 %	4.97º	0.72º
Minimización producto (reinterpreta [17])	0.18	0.90	8.10 %	69 %	5.30º	0.92º
Método propuesto	0.15	1.04	7.35 %	75 %	2.65º	0.37º

Tabla 8: Comparativa de mediciones de calidad tras cada método de postproceso estudiado. El método propuesto es óptimo, destacando la notable reducción del error de crecimiento de tamaño (mayor fidelidad a los gradientes de tamaño impuestos por el usuario). También reduce el error relativo de tamaño de los otros métodos, y proporciona una distorsión de Oddy óptima (mejorando la de los métodos basados en Giuliani, y resultando muy similar a la obtenida con el método de minimización del producto de la distorsión por el error de tamaño).

Muy destacable es el bajo valor del error de crecimiento de tamaño, reduciendo a la mitad el error máximo de otros métodos (recordemos que este ejemplo tiene una gradación de tamaño de elementos impuesta por el usuario, y por tanto el error de crecimiento de tamaño es también un indicador importante). Con un error máximo de 2.65º y una media del valor absoluto del error de 0.37º es el método que mejor logra conservar las gradaciones de tamaño de elementos.

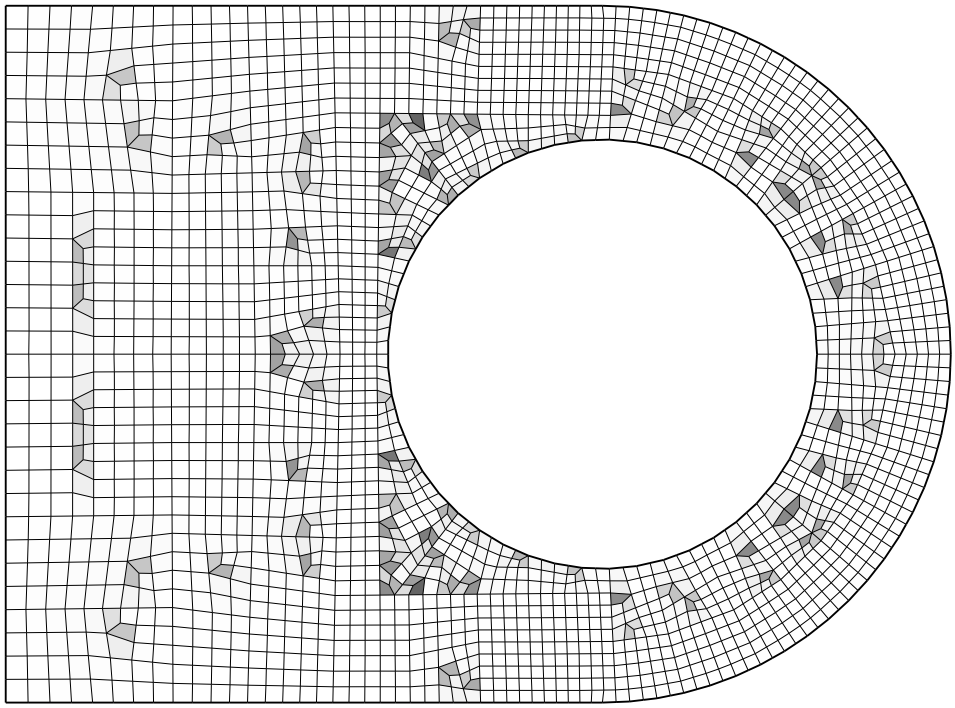
A la vista de la tabla 8 se podría percibir la falsa impresión de que el método original de Giuliani, pese a obtener peores resultados que el propuesto[(33)], no parece quedar lejos. Sin embargo, como se ha dicho con anterioridad, el método original de Giuliani tiende a igualar los tamaños de todos los elementos, circunstancia que no ha disparado notablemente las mediciones numéricas de calidad, pero que sí que se aprecia visualmente en la figura 36a (la variación de tamaño impuesta por el usuario queda muy difuminada).

En definitiva, se observa que el método propuesto responde de manera excelente a los criterios de calidad exigidos, porque queríamos reducir tanto como fuera posible los errores de tamaño, respetando las gradaciones deseadas por el usuario, y todo ello con la menor distorsión posible de los cuadriláteros. El modelo de muelles, con las diagonales controlando la distorsión del elemento, y las aristas cuidando del tamaño, logra responder muy bien a todos estos requisitos, sin descuidar ninguno.

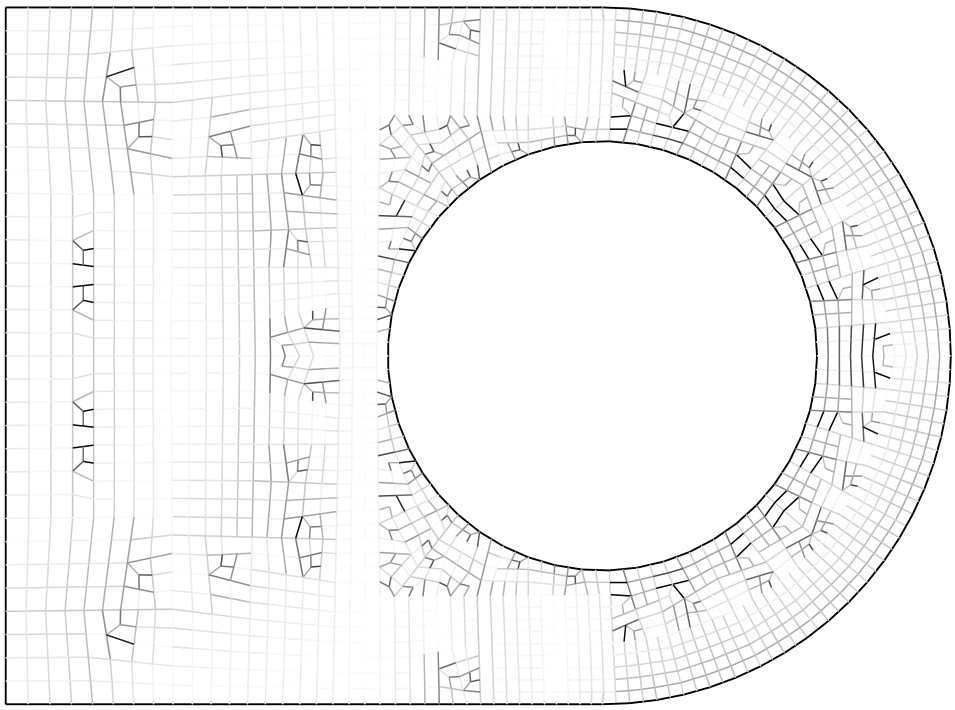
También quedan eliminados los "patrones de ruido" que aparecen en métodos como el de Giuliani modificado por [50] (zonas de elementos de tamaño notablemente diferente al que deberían tener). Comparando visualmente las figuras 36b y 39b, se observa como en el método propuesto (segunda figura) se han eliminado la presencia de estas zonas, que sí que están patentes en la primera.

Además de la calidad, la implementación del método se puede hacer de manera muy eficiente tanto en *CPU* como en *GPU*, gracias a sus posibilidades de paralelización. En cada iteración, el suavizado de cada nudo se puede hacer de manera independiente al del resto de nudos, lo que permite una paralelización masiva sin necesidad de comunicación de datos entre núcleos de proceso (sólo es necesaria la comunicación al finalizar cada iteración).

[(33)] Excepto en la distorsión media, que reduce ligeramente.



(a) Distorsión de Oddy en mallado previo al postproceso. A mayor distorsión, tonalidad más oscura (distorsión media=0.31; distorsión percentil 99º=4.27).



(b) Error relativo de tamaño de aristas en mallado previo al postproceso. A mayor error, tonalidad más oscura (error relativo medio=8.69 %; el 68 % de aristas no rebasan el error relativo 10 %).

Figura 34: Mediciones de calidad en mallado original.

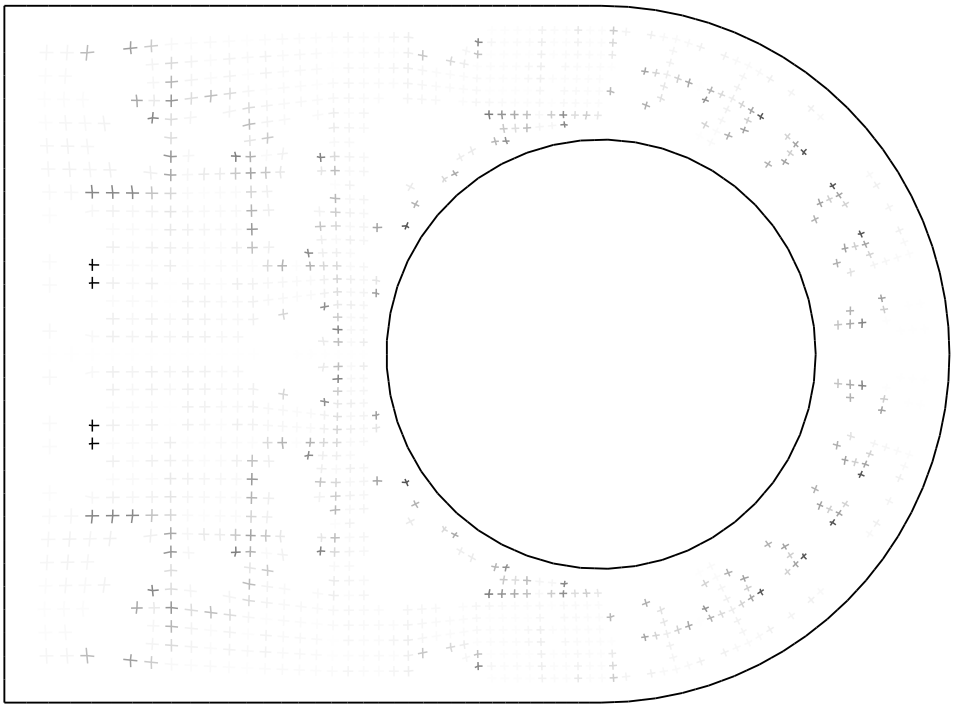
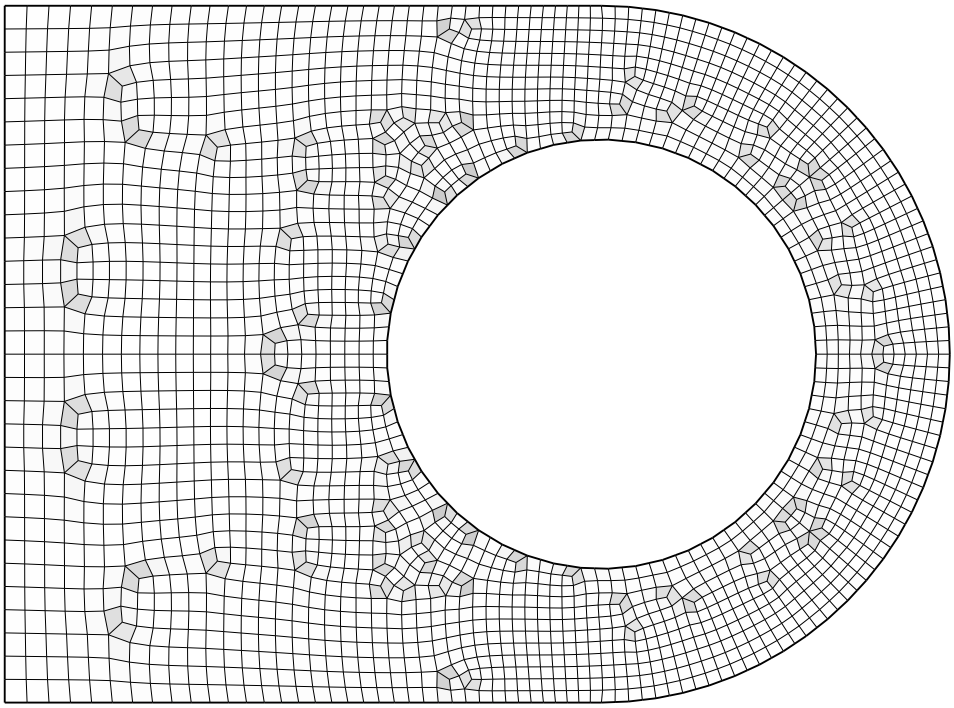
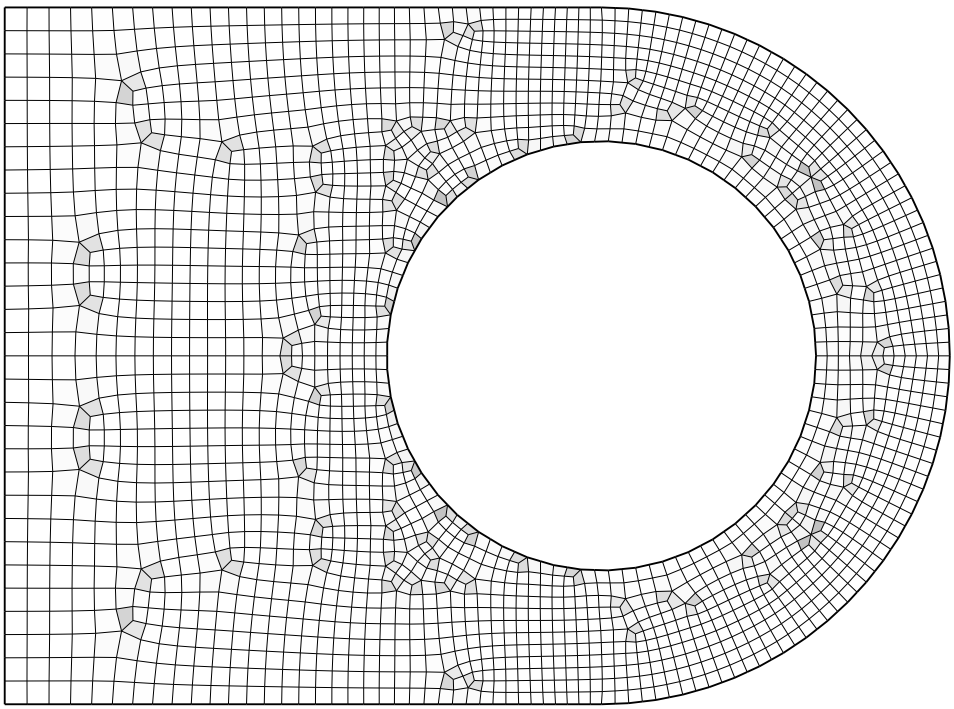


Figura 35: Error de variación de tamaño en zonas de mallado estructurado previo al postproceso, expresado como error de pendiente de crecimiento en grados. A mayor error, tonalidad más oscura (máximo=18.98°; media del valor absoluto=0.60°).

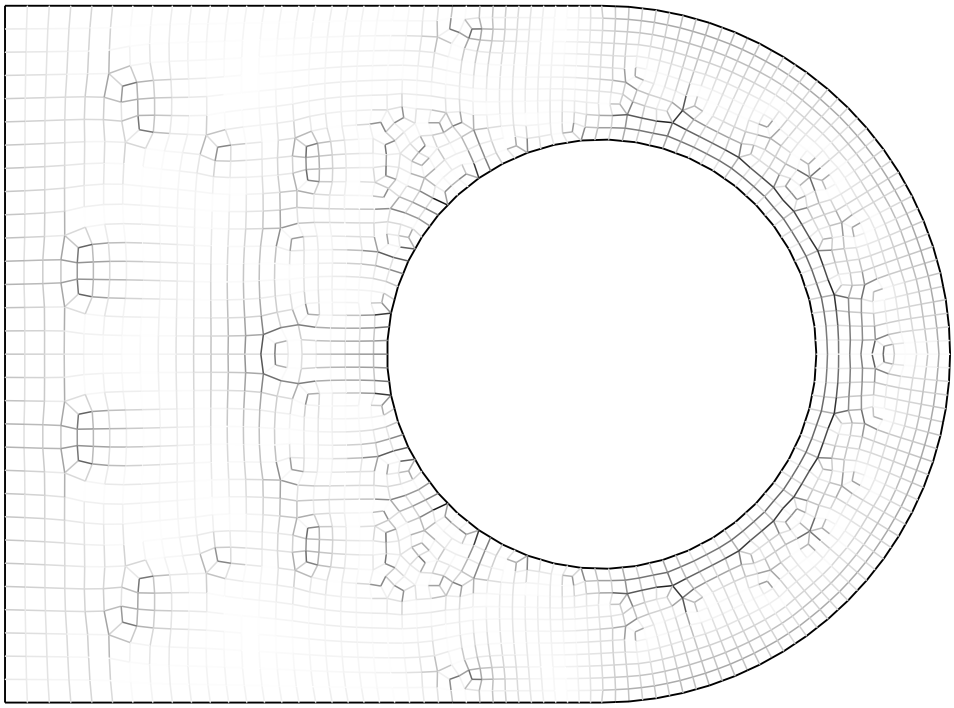


(a) Distorsión de Oddy tras postproceso de Giuliani. A mayor distorsión, tonalidad más oscura (distorsión media=0.16; distorsión percentil 99º=1.56).

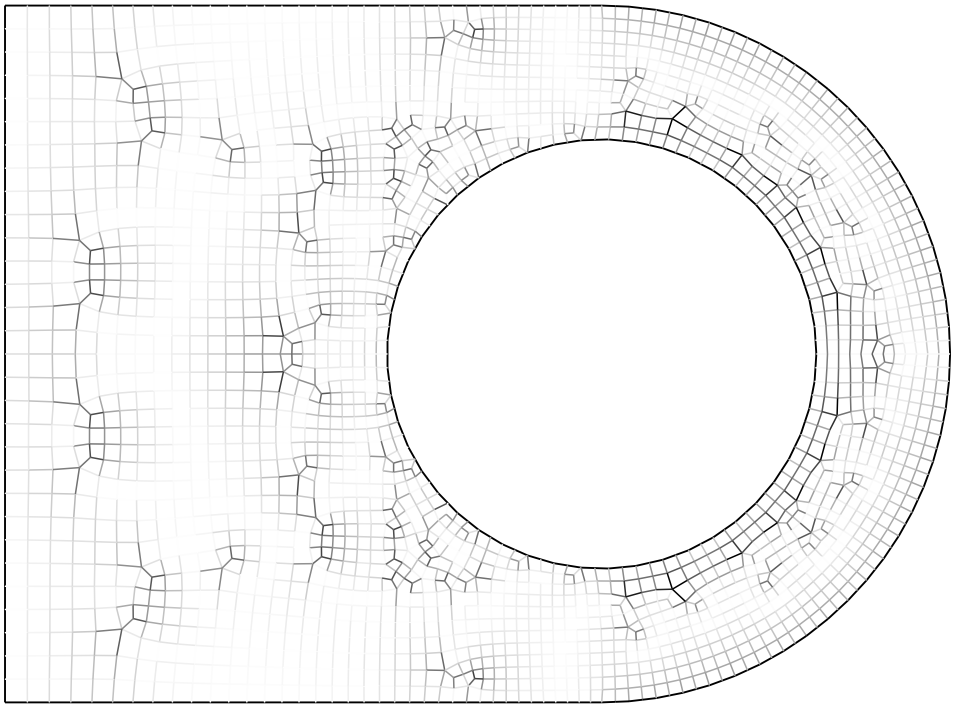


(b) Distorsión de Oddy tras postproceso de Giuliani con modificación de [50]. A mayor distorsión, tonalidad más oscura (distorsión media=0.14; distorsión percentil 99º=1.65).

Figura 36: Distorsión tras postprocesos que no corrigen el tamaño.

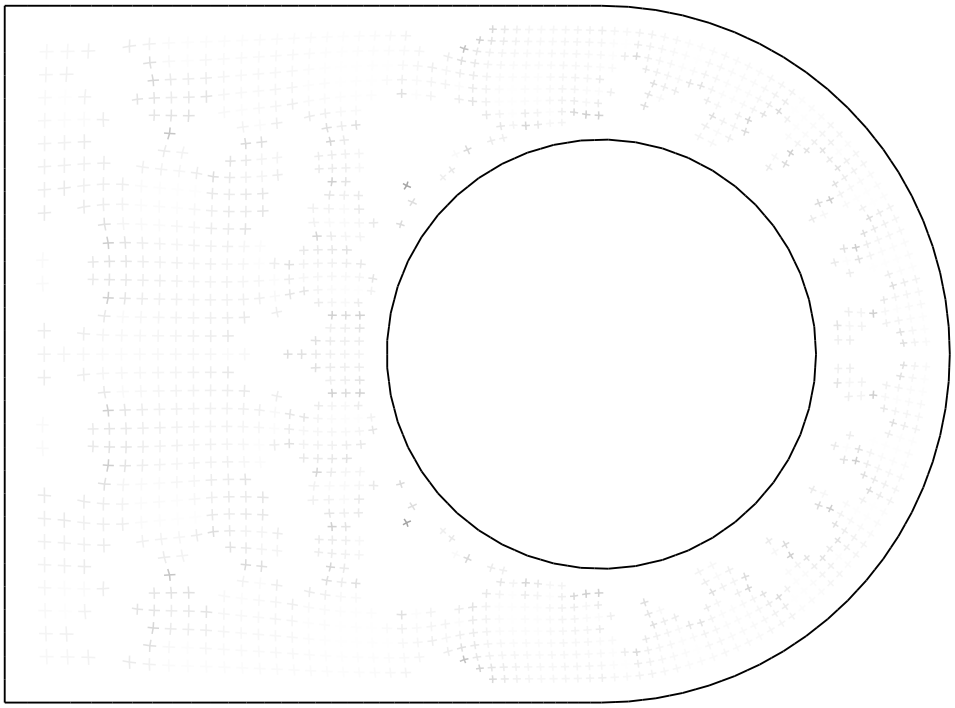


(a) Error relativo de tamaño de aristas tras postproceso de Giuliani. A mayor error, tonalidad más oscura (error relativo medio=7.77 %; el 72 % de aristas no rebasan el error relativo 10 %).

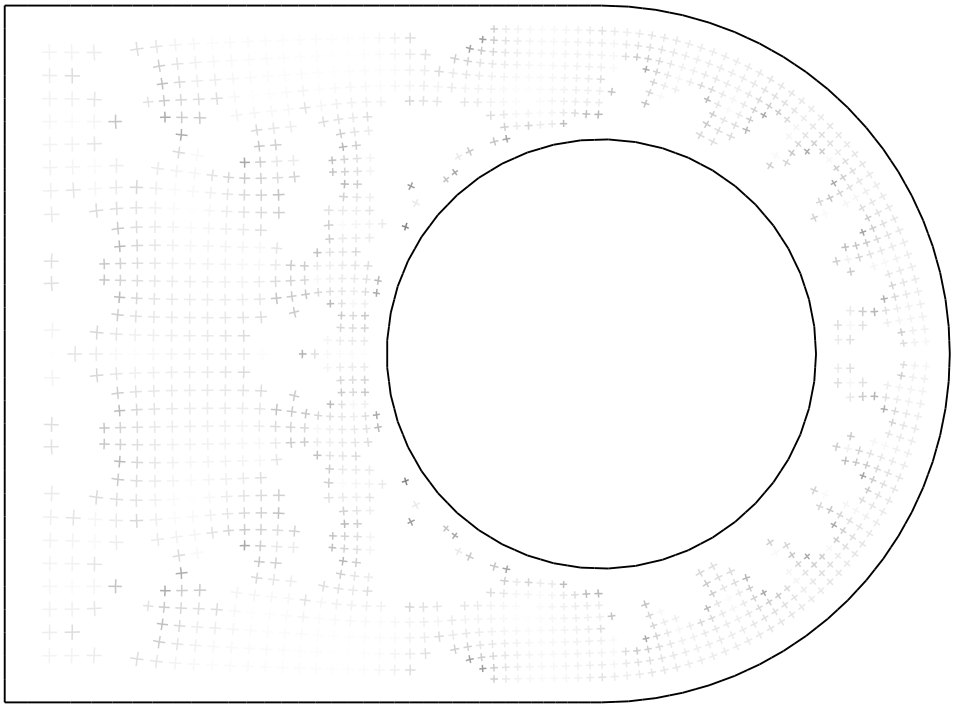


(b) Error relativo de tamaño de aristas tras postproceso de Giuliani con modificación de [50]. A mayor error, tonalidad más oscura (error relativo medio=9.33 %; el 64 % de aristas no rebasan el error relativo 10 %).

Figura 37: Error de tamaño tras postprocesos que no corrigen tamaño.

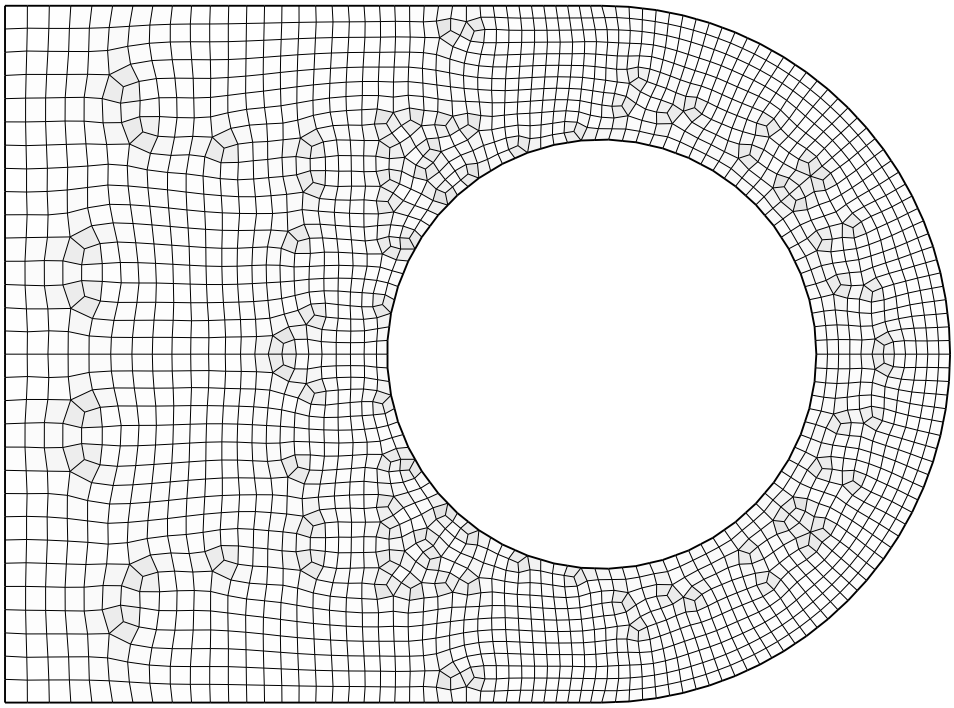


(a) Error de variación de tamaño en zonas de mallado estructurado tras postproceso de Giuliani, expresado como error de pendiente de crecimiento en grados. A mayor error, tonalidad más oscura (máximo=3.75°; media del valor absoluto=0.39°).

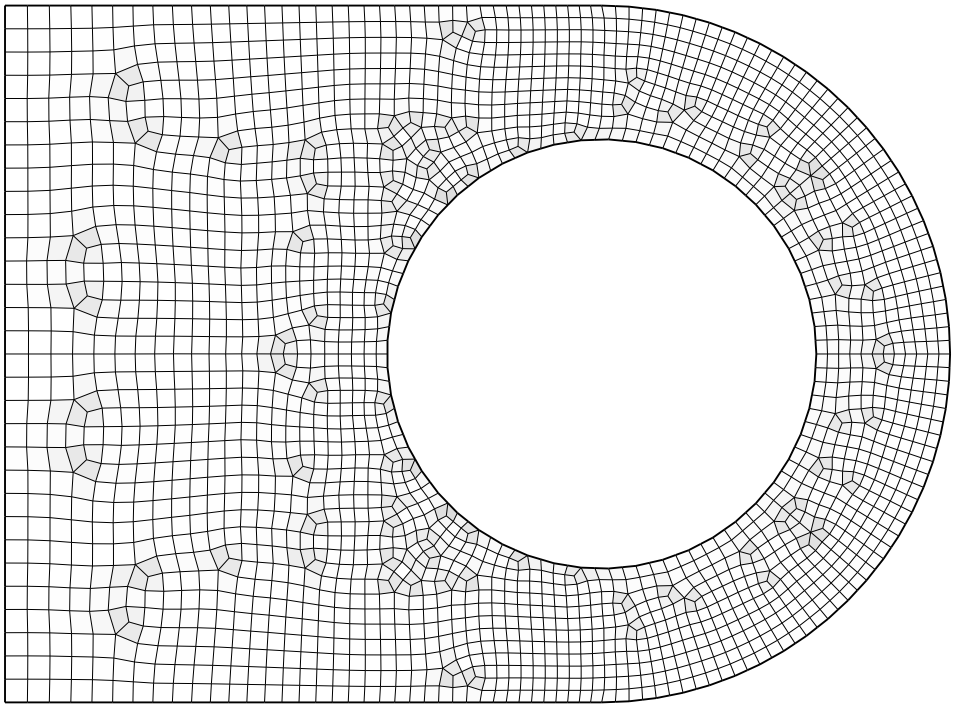


(b) Error de variación de tamaño en zonas de mallado estructurado tras postproceso de Giuliani con modificación de [50], expresado como error de pendiente de crecimiento en grados. A mayor error, tonalidad más oscura (máximo=4.97°; media del valor absoluto=0.72°).

Figura 38: Error de variación de tamaño en métodos que no corrigen tamaño.

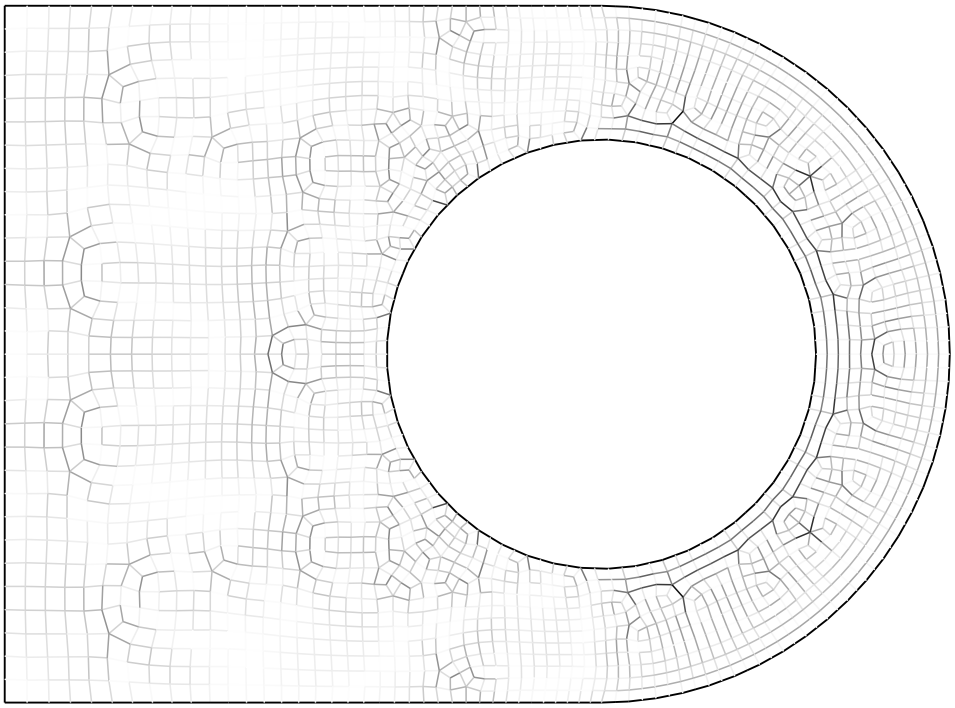


(a) Distorsión de Oddy tras postproceso de minimización del producto del error de área por la media de la distorsión. A mayor distorsión, tonalidad más oscura (distorsión media=0.18; distorsión percentil 99º=0.90).

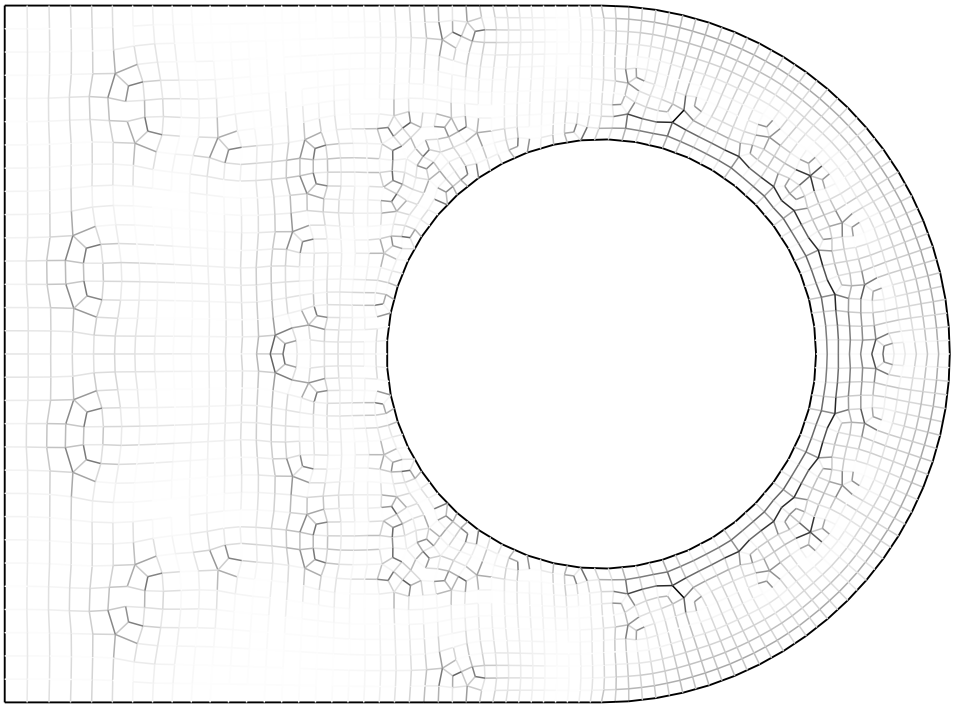


(b) Distorsión de Oddy tras postproceso con el método propuesto. A mayor distorsión, tonalidad más oscura (distorsión media=0.15; distorsión percentil 99º=1.04).

Figura 39: Distorsión tras postprocesos que corrigen el tamaño.

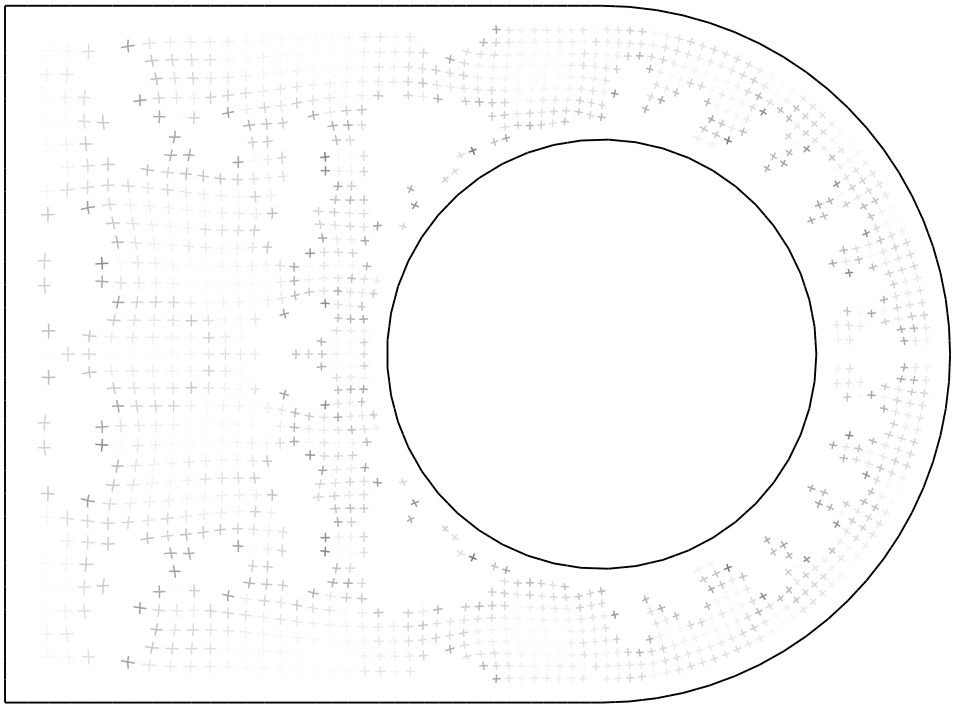


(a) Error relativo de tamaño de aristas tras postproceso de minimización del producto del error de área por la media de la distorsión. A mayor error, tonalidad más oscura (error relativo medio=8.10 %; el 69 % de aristas no rebasan el error relativo 10 %).

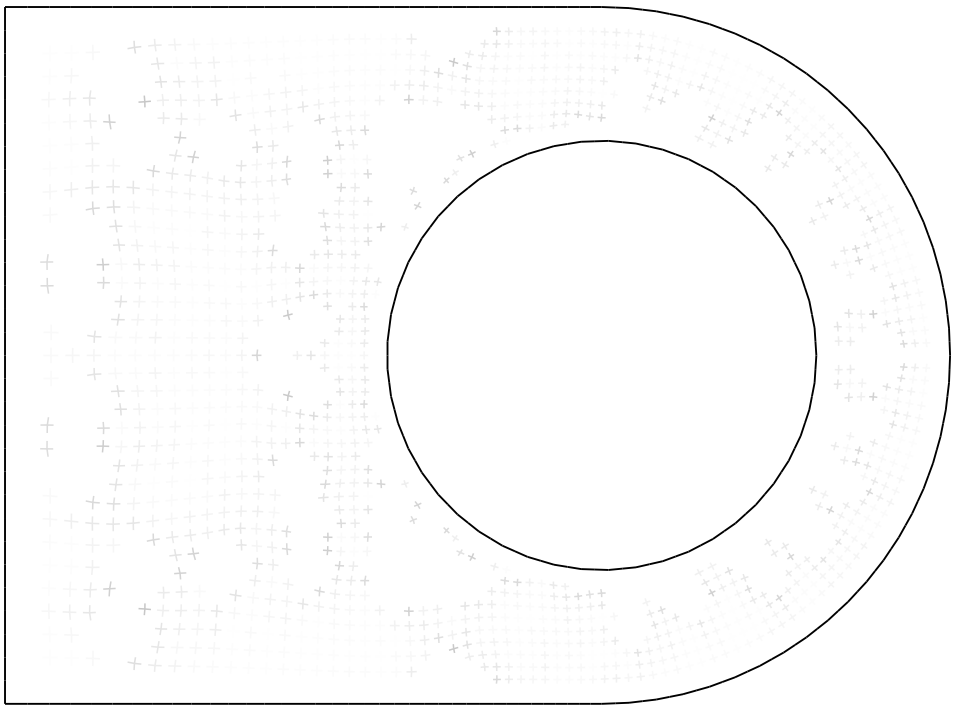


(b) Error relativo de tamaño de aristas tras postproceso con el método propuesto. A mayor error, tonalidad más oscura (error relativo medio=7.35 %; el 75 % de aristas no rebasan el error relativo 10 %).

Figura 40: Error de tamaño tras postprocesos que corrigen tamaño.



(a) Error de variación de tamaño en zonas de mallado estructurado tras postproceso de minimización del producto del error de área por la media de la distorsión, expresado como error de pendiente de crecimiento en grados. A mayor error, tonalidad más oscura (máximo=5.30°; media del valor absoluto=0.92°).



(b) Error de variación de tamaño en zonas de mallado estructurado tras postproceso con el método propuesto, expresado como error de pendiente de crecimiento en grados. A mayor error, tonalidad más oscura (máximo=2.65°; media del valor absoluto=0.37°).

Figura 41: Error de variación de tamaño en métodos que corrigen tamaño.

3.5. Ejemplo de aplicación. Radiosidad

Para mostrar el uso de mallados cuadrangulares, y en concreto de los métodos descritos en esta Tesis Doctoral, se ha elegido como ejemplo de aplicación el análisis de *radiosidad*.

En el mundo anglosajón se conoce como *Radiosity Method*, debido a que equilibra la *exitancia* radiante, que en inglés se denomina a menudo *radiosity*, aunque con significado no siempre equivalente[34] (la traducción al castellano como *radiosidad* se refiere sobre todo al método *radiosity*, siendo menos frecuente su empleo como equivalente de *exitancia*).

Se trata de un equilibrio de radiación luminosa, planteado como método de elementos finitos. El algoritmo original data de 1984 (Goral, Torrance, Greenberg, y Battaile [19]; y Nishita y Nakamae [38]). Ambas fuentes se basan en métodos numéricos de la Termodinámica para Transferencia de calor, con origénes en los años 1950 (Hottel [21], y Eckbert y Drake [15]).

A pesar de ser una herramienta que permite estimar valores del flujo radiante y diversas magnitudes radiométricas y fotométricas -con directa aplicación en evaluación de luminotecnia-, sin embargo el uso más popular de la *radiosidad* se ha producido en áreas cuyo objetivo no es la estimación de magnitudes físicas[35].

Cabe decir sin embargo que el método de *radiosidad* ha tenido siempre como dificultad la necesidad de disponer de un mallado robusto y de calidad, con el condicionante de que si el número de elementos es elevado, los costes de cálculo dejan de ser razonables[36]. Hay formulaciones de *radiosidad* jerárquica [10] que permiten abordar el análisis de problemas grandes de manera más eficiente pero, a pesar de ello, lograr un mallado óptimo en términos de calidad y de número de elementos no es trivial.

Esto ha frenado su uso masivo, y ha conducido a que surjan métodos alternativos no basados en elementos finitos, tales como seguimiento estocásti-

[34] La equivalencia entre *exitancia* y *radiosity* no es necesariamente literal, puesto que en algunos contextos se considera que la *exitancia* engloba toda la radiación que emerge de una superficie por unidad de área (suma de la emitida, reflejada, y transmitida), mientras que en otros usos se refiere exclusivamente a la emitida. Por contra, el término *radiosity* se emplea siempre para referirse a la suma de la emitida, reflejada, y transmitida.

[35] Por ejemplo, la realidad virtual, la infografía, y la animación por ordenador, utilizan el método de *radiosidad* por el realismo de las imágenes que produce (sobre todo en escenas donde la iluminación indirecta sea protagonista).

[36] La matriz del problema no es dispersa, porque generalmente cada elemento recibe radiación directa de muchos otros elementos.

	Expresión Inglesa	Equivalente en Fotometría
Flujo radiante ó *Potencia radiante* (W)	*Radiant power/flux*	Flujo luminoso ó *Potencia luminosa* (lm)
Irradiancia (W/m^2)	*Irradiance*	Iluminancia $(lx = lm/m^2)$
Exitancia radiante (W/m^2)	*Radiosity (ó Radiant exitance)*	Emitancia luminosa $(lx = lm/m^2)$
Radiancia $(W \cdot sr^{-1} \cdot m^{-2})$	*Radiance*	Luminancia $(cd/m^2 = lm \cdot sr^{-1} \cdot m^{-2})$

Tabla 9: Magnitudes de Radiometría empleadas en análisis de *radiosidad*.

co de fotones y rayos de luz, procedimientos que no requieren descomponer la escena en un mallado de elementos.

Todo lo que hemos expuesto sobre mallado cuadrangular nos abre la posibilidad de ser aplicado en el método de *radiosidad*, aliviando estas dificultades que han frenado parcialmente su empleo generalizado. Los algoritmos descritos en esta Tesis Doctoral nos permiten generar un mallado cuadrangular de manera eficiente, con variación del tamaño de elementos, otorgando al usuario flexibilidad y respuesta interactiva al mallar la geometría. El método propuesto de postproceso mediante muelles ayuda a obtener unos elementos lo más regulares posible, que a la vez tienden al tamaño deseado en cada zona del dominio.

Además, la práctica totalidad de bibliografía sobre *radiosidad* emplea mallado cuadrangular en sus ejemplos y planteamientos, mientras que la mayoría de implementaciones reales en *software* se han basado en mallado triangular, por la menor disponibilidad de algoritmos de mallado cuadrangular robusto. Esto constituye otro argumento a favor para considerar que la *radiosidad* se puede beneficiar de los algoritmos que hemos descrito.

En la tabla 9 se indican magnitudes que intervienen en el método de *radiosidad*. La *irradiancia* y la *exitancia* son análogas en cuanto a que ambas miden la densidad del flujo radiante por unidad de superficie, pero difieren en el sentido del mismo: La *irradiancia* expresa la densidad del flujo total que incide sobre una superficie, mientras que la *exitancia* mide la densidad del flujo que emerge de la superficie.

Por otra parte, la *radiancia* mide la densidad del flujo en función de

la dirección. Las superficies *Lambertianas* emiten, reflejan, y transmiten la radiación al igual en todas direcciones, por lo que su *radiancia* es constante respecto de la dirección.

El análisis de *radiosidad* se emplea principalmente con hipótesis de superficie *Lambertiana*, pero es posible también ampliarlo a modelos con *radiancia* variable respecto de la dirección[37].

La ecuación a resolver para un problema ya discretizado en n elementos, supuestos todos ellos superficies *Lambertianas* es [54]:

$$B_i = E_i + \rho_i \sum_{j=1}^{n} B_j F_{ij} \quad \text{para } i = 1, 2, ..., n \tag{80}$$

donde,

B_i es la *exitancia* (*radiosidad*) del elemento i en W/m^2

E_i es la emisión del elemento i en W/m^2 (sólo si i es fuente de luz)

ρ_i es la reflectividad de i (fracción de luz incidente que es reflejada)

B_j es la *exitancia* (*radiosidad*) de un elemento j en W/m^2

F_{ij} es la fracción (adimensional) de energía que llega a j desde i

Aunque pueda parecer confuso que se esté empleando F_{ij} para evaluar la luz que llega a i (a priori la intuición sugeriría utilizar F_{ji}), sin embargo este hecho se debe a que la ecuación (80) no expresa energía, sino densidad de energía. Como indica [54], si consideramos la relación de reciprocidad

$$F_{ij} A_i = F_{ji} A_j \tag{81}$$

de inmediato obtenemos una expresión más intuitiva al convertir (80) en una ecuación de flujo radiante, que en efecto emplea F_{ji}:

$$A_i B_i = A_i E_i + \rho_i \sum_{j=1}^{n} A_i B_j F_{ij} = A_i E_i + \rho_i \sum_{j=1}^{n} A_j B_j F_{ji} \tag{82}$$

donde A_i y A_j son las áreas de los respectivos elementos.

[37] Considerar la direccionalidad en el análisis incrementa en gran medida los recursos de computación necesarios, y conduce generalmente a dos estrategias: o bien discretizar las direcciones, o bien hacer un cálculo estocástico. Además, hay que tener en cuenta que si las superficies no son *Lambertianas*, la radiación de luz que llegue desde cada superficie al observador dependerá de su posición, por lo que no es sencillo obtener una solución independiente del punto de vista, a menos que contenga la radiación que viaja en cada dirección.

En cualquier caso, es más habitual plantear el problema con la ecuación (80) que con (82).

F_{ij} recibe el nombre de factor de forma de i a j. Depende de la forma de los elementos i y j, así como de su orientación relativa, y de que existan o no obstáculos que impidan (total o parcialmente) que la radiación llegue desde i a j. Es un valor comprendido entre 0 y 1.

Según la analogía de Nusselt [22], el factor de forma F_{ij} puede medirse proyectando el elemento j sobre una semiesfera de radio unidad centrada en i y a continuación volviendo a proyectar el resultado sobre la base circular plana de la semiesfera. También es necesario considerar la posible existencia de obstáculos que impidan que la radiación llegue desde i a j, detalle que vuelve inviable llegar a una expresión algebraica para el factor de forma, y obliga a estimarlo mediante muestras discretas.

El mayor coste de cómputo del análisis de *radiosidad* proviene, con diferencia, de la obtención de los factores de forma [11]. Para hallar cada F_{ij} es necesario comprobar si cualquier otro elemento del modelo obstaculiza (*"hace sombra"*) total o parcialmente cuando i transfiere energía a j.

La posibilidad de almacenar los factores de forma una vez obtenidos es muy deseable, pero puede no ser viable: el número de factores a almacenar tendería a n^2 (ó $\frac{n^2}{2}$ si tenemos en cuenta la relación de reciprocidad (81), sólo pudiendo evitar los factores que se anulen -habitualmente minoría).

Por ello, algunas implementaciones almacenan los factores de forma en un *caché* en *RAM* o en disco, que con frecuencia alcanza varios GB. Otras implementaciones no los almacenan, volviéndolos a obtener para cada ecuación de la iteración actual, con la consiguiente penalización de rendimiento.

3.5.1. Método de Radiosidad implementado

La ecuación (80) constituye la formulación clásica de la *radiosidad*. La resolución iterativa de dicha ecuación conduce a hallar la densidad de flujo radiante que llega a cada elemento en cada iteración, es decir, su *irradiancia*.

Es más eficiente resolver el problema en dirección opuesta, contemplando la *exitancia* en lugar de la *irradiancia*. De esta manera, en cada iteración hallamos la densidad de flujo radiante que un elemento envía hacia cada elemento del modelo, en vez de obtener la que él recibe. Este procedimiento se conoce como *radiosidad* con refinamiento progresivo [11].

Al replantear la ecuación (80) para hallar la energía que el elemento i envía a cada uno de los elementos del modelo, necesitamos emplear el factor de forma F_{ji} en lugar del F_{ij}. Pero es preferible utilizar F_{ij} cuando emitimos energía desde i al resto de elementos del modelo, porque de esta manera podemos obtener los F_{ij} hacia todos los elementos j proyectando sobre una única semiesfera centrada en i. Gracias a la relación de reciprocidad (81) podemos cambiar F_{ji} por F_{ij} [11].

Así pues, la energía que j envía a i resulta

$$B_j \text{ debido a } B_i = \rho_j B_i F_{ji} = \rho_j B_i F_{ij} \frac{A_i}{A_j} \tag{83}$$

Este planteamiento es más eficiente porque podemos elegir para cada iteración el elemento que tenga más energía pendiente de emitir, que será el que más contribuya a llegar lo antes posible a una solución cercana a la ideal. De esta manera, las primeras iteraciones nos proporcionan un resultado mucho más útil que en el planteamiento original del método de *radiosidad*.

El método implementado ha sido precisamente este, para obtener una convergencia más temprana a la solución.

Para el cálculo de los factores de forma F_{ij} se ha utilizado el procedimiento del *hemicubo* acelerado en *hardware* [12], consistente en situar medio cubo en el centro de cada elemento, y proyectar el resto de los elementos del modelo sobre cada una de sus caras, con eliminación de superficies ocultas (para considerar la posible obstaculización/sombras entre elementos). Esta técnica transforma la analogía de Nusselt en una proyección sobre caras planas, que puede beneficiarse de *hardware* de *Buffer Z* disponible en todas las *GPU* actuales. De esta manera aprovechamos la *GPU* para acelerar la parte más costosa del análisis de *radiosidad*.

Como el *hemicubo* es una estimación discreta (sus caras están constituidas por *pixels*), es susceptible de presentar riesgo de *aliasing*, pero la resolución elegida (lado del cubo igual a 850 *pixels*) ha sido suficiente para que la estimación de los factores de forma sea buena en los modelos utilizados. En general, la resolución de los *hemicubos* debe elegirse de forma que la proyección de cada elemento sea siempre de varios *pixels*. En caso contrario, la precisión disminuye mucho, al tiempo que se produce *aliasing* y se anulan factores de forma que en realidad no son nulos.

El método así descrito se ha programado en lenguaje *C++*, empleando la librería gráfica *OpenGL* para acelerar en *hardware* el cómputo de *hemicubos* y para la visualización final de las imágenes.

Los factores de forma se han calculado bajo demanda (cada factor obtenido solamente cuando es necesario) y se han ido almacenando en un *caché*, lo que ha acelerado muy significativamente las iteraciones que necesitaban factores previamente calculados. Como contrapartida, este *caché* ha llegado a ocupar 3 GB en el modelo de la *Caja Cornell* (con el mallado de la figura 43) y cerca de 40 GB en el modelo del Palacio de Sponza (figura 51).

Las ecuaciones mostradas en este apartado no consideran el color. Es trivial introducirlo, resolviendo la ecuación para cada franja de longitud de onda del espectro luminoso. En la implementación realizada, se ha resuelto exclusivamente para los tres colores primarios R, G, B, aunque el método admite resolución espectral. Nótese que los factores de forma no dependen de la longitud de onda[38], por lo que su coste de obtención no incrementa al trabajar con color.

3.5.2. Modelo de la Caja Cornell

La *Caja Cornell* (*"Cornell Box"*) es una maqueta realizada en el *"Program of Computer Graphics"* de la Universidad Cornell, a partir del año 1984. Es casi cúbica, de tamaño ligeramente mayor a medio metro de lado.

Se introdujo por primera vez en el primer artículo publicado sobre el método de *radiosidad* [19]. Realizando fotografías y diversas mediciones de luz de la maqueta, se empleó como un medio para validar el método de *radiosidad*, comparando así los resultados con las mediciones obtenidas de la maqueta.

A lo largo de los años la *Caja Cornell* ha tenido varias versiones, según el algoritmo que se estuviese validando, siendo también aplicada a otros métodos además de la *radiosidad*. Como la información de la maqueta es pública [13], se ha utilizado también por desarrolladores de *software* gráfico de todo el mundo para comprobación de resultados.

La versión utilizada en el presente análisis es la disponible en [13], con la pared izquierda roja y la derecha verde, siendo todas las superficies *Lambertianas*.

El modelo se ha dibujado con las coordenadas especificadas en dicha página *web* oficial, y los contornos se han pasado directamente al algoritmo

[38] Si las superficies no fuesen *Lambertianas* y se modelizasen fenómenos de refracción y dispersión de luz, los factores de forma dependerían de la longitud de onda.

de mallado cuadrangular que hemos descrito, así como postprocesado con el método de muelles propuesto.

Los datos de color originales están descritos por longitud de onda. Se ha hecho una transformación aproximada a los primarios R, G, B, efectuándose el análisis en R, G, B en vez de en franjas de longitudes de onda.

Un primer análisis, con 363 elementos (que apenas necesita 500 KB para almacenar todos los factores de forma) puede verse en la figura 42. El tamaño de los elementos es demasiado grande para poder capturar en detalle el comportamiento de la luz en el modelo (en particular las sombras de los prismas son muy imprecisas). No obstante, a pesar de que el tamaño de los elementos es inadecuado, nos proporciona una primera aproximación al equilibrio de radiación. De hecho, se ha simulado correctamente la reflexión difusa de las paredes roja y verde sobre las superficies blancas.

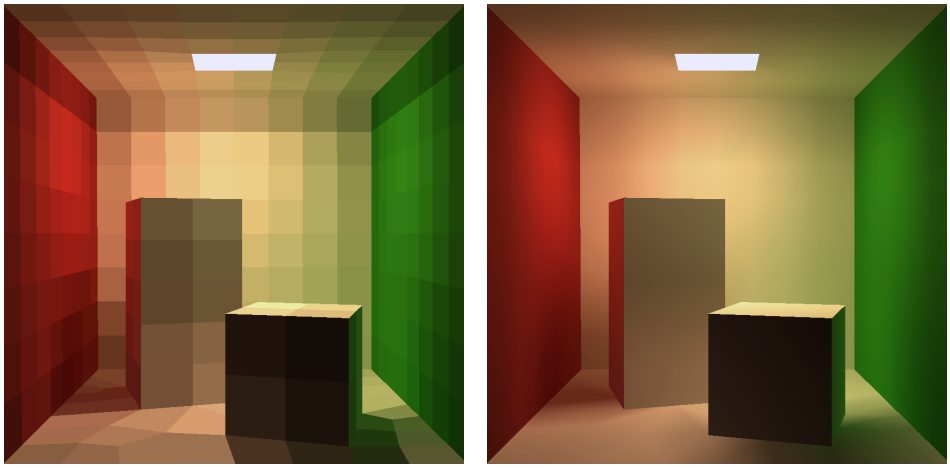
Aumentando la finura de la discretización en las zonas en que necesitamos más detalle (aplicando la gradación del tamaño de elementos del algoritmo de mallado cuadrangular, y relajándolo con el postproceso basado en muelles) obtenemos el mallado del suelo de la caja que puede verse en la figura 44. El resto de superficies de la *Caja Cornell* se ha mallado especificando un tamaño de elemento constante y, siendo rectangulares, obsérvese que el algoritmo ha producido en ellas un mallado estructurado tal y como sería deseable (figura 43a).

Este nuevo modelo, con 27122 elementos, necesita cerca de 3 GB para almacenar todos los factores de forma.

En la figura 46 se facilita una comparativa de la solución de este modelo (con interpolación lineal) con la imagen sintética de referencia facilitada en la página web de la *Caja Cornell*.

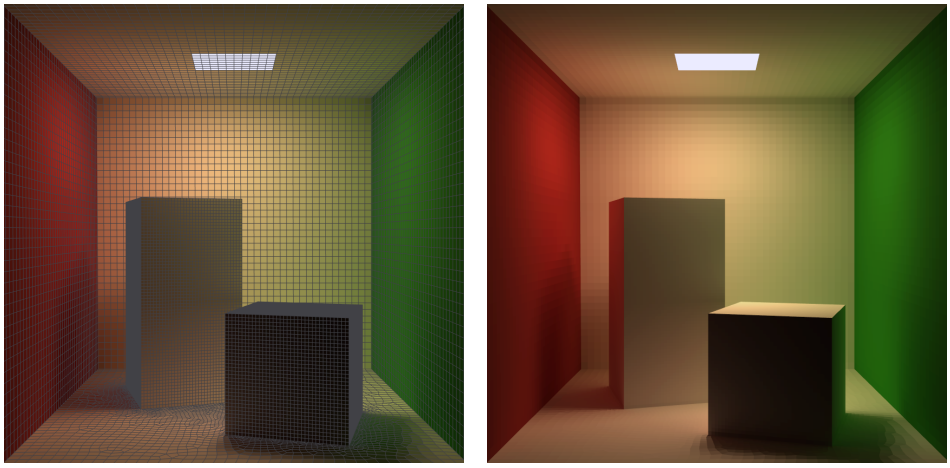
El perímetro de las sombras (tanto las nítidas como las de penumbra difusa) en ambas imágenes es prácticamente idéntico, así como las reflexiones difusas de las paredes de color. La cromaticidad no es exacta, debido a que el análisis se ha realizado en valores R, G, B.

El método de *radiosidad* permite simular complejos fenómenos de reflexión difusa, como la zona de ligera luminosidad que parece verse detrás del prisma alto. No se debe a iluminación directa (está en sombra), sino a la reflexión difusa de la cara trasera del prisma, que es blanca. Una ampliación de esta zona puede verse en la figura 47a.



(a) Solución sin interpolar. (b) Solución con interpolación lineal.

Figura 42: Mallado y solución de la *Caja Cornell* con elementos de tamaño grande. El análisis no posee la finura necesaria para definir las sombras con nitidez suficiente, quedando patente al interpolar la solución.



(a) Mallado completo. (b) Solución sin interpolar.

Figura 43: Mallado y solución de la *Caja Cornell* empleando elementos más pequeños en las zonas en que es necesario capturar más detalles.

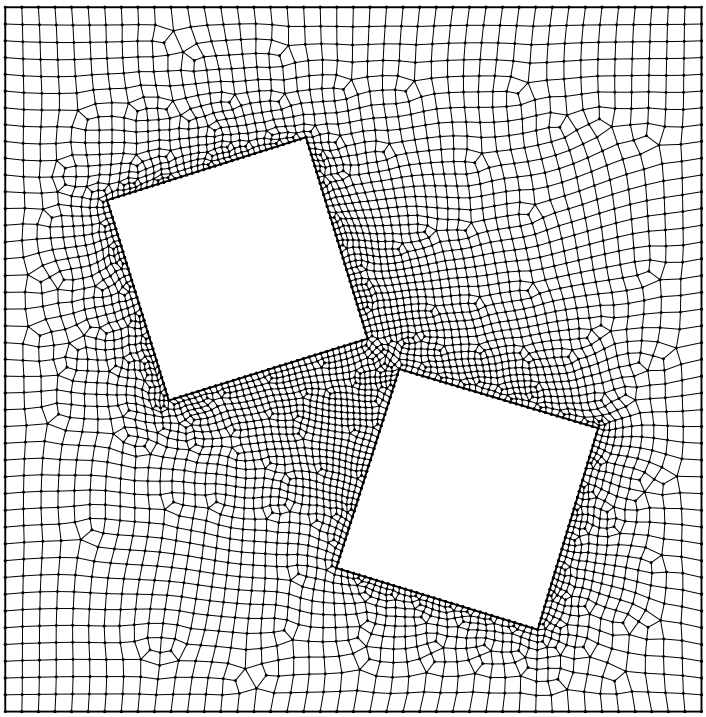


Figura 44: Mallado del suelo de la *Caja Cornell*.

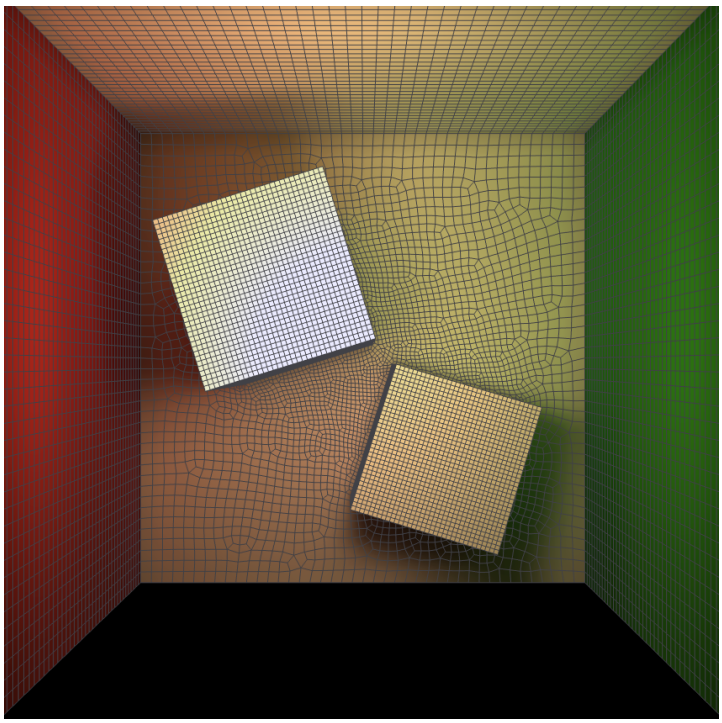
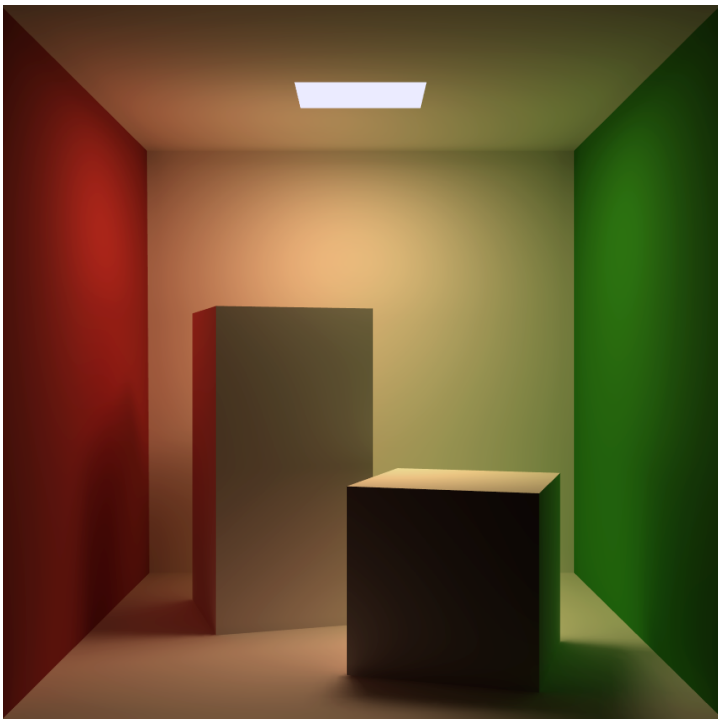
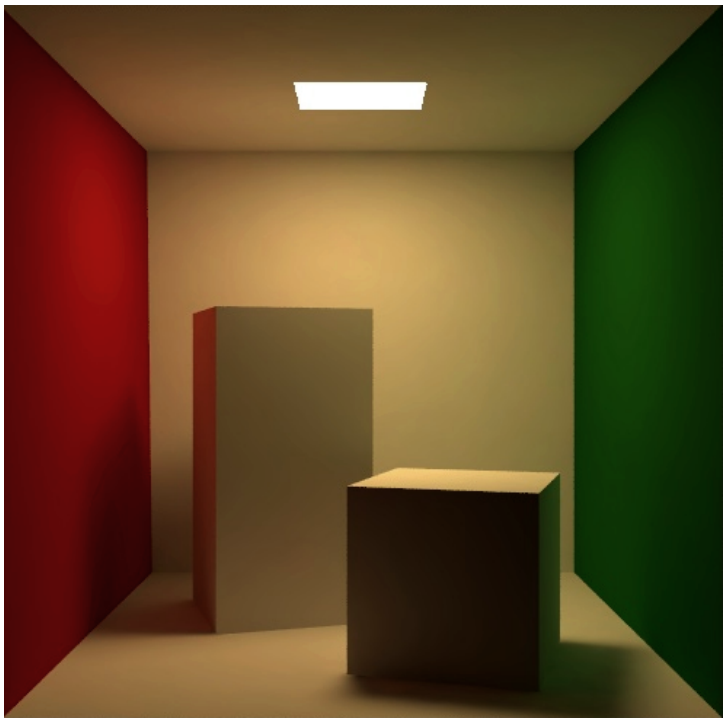


Figura 45: Mallado y solución de la *Caja Cornell* (vista desde arriba).

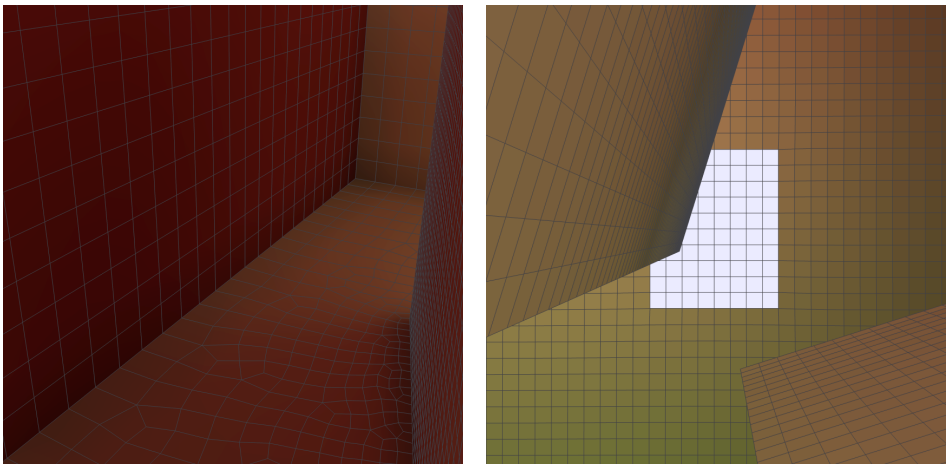


(a) Solución con interpolación lineal.



(b) Imagen sintética de referencia.

Figura 46: Comparativa del análisis con la imagen de referencia de la *Caja Cornell*.



(a) Detalle de la zona de luz creada por la reflexión difusa de la cara trasera del prisma alto.

(b) Vista cenital, similar a la cara superior de los hemicubos de los elementos del suelo.

Figura 47: Detalles de la solución de la *Caja Cornell*.

3.5.3. Modelo del atrio del Palacio Sponza

En [33] se encuentra otro modelo ampliamente utilizado para el ensayo de algoritmos de *radiosidad* e iluminación global: El atrio del Palacio Sponza (en Dubrovnik), modelo realizado por Marko Dabrovic en 2002. Se trata de un palacio construido entre 1516 y 1522, que combina estilos gótico y renacentista.

El modelo realizado por Marko Dabrovic está formado por caras planas, pero no es un mallado para análisis, sino representación de las superficies describiéndolas con el menor número de caras posible.

Por ello, en primer lugar se ha hecho el trabajo de obtener los contornos a mallar, para después pasarlos al algoritmo de mallado cuadrangular y al postproceso mediante muelles.

En este proceso se ha detectado que el modelo original tenía algunas incorrecciones topológicas. En concreto, algunas superficies no eran orientables (por tener aristas compartidas por más de dos caras), y había caras degeneradas (con área nula) y *"vértices T"*[39].

La tarea a realizar en este modelo no es por tanto un mallado propia-

[39] Se conoce como *"vértice T"* la situación producida cuando un vértice de una cara pertenece a la arista de otra sin que dicha arista quede dividida.

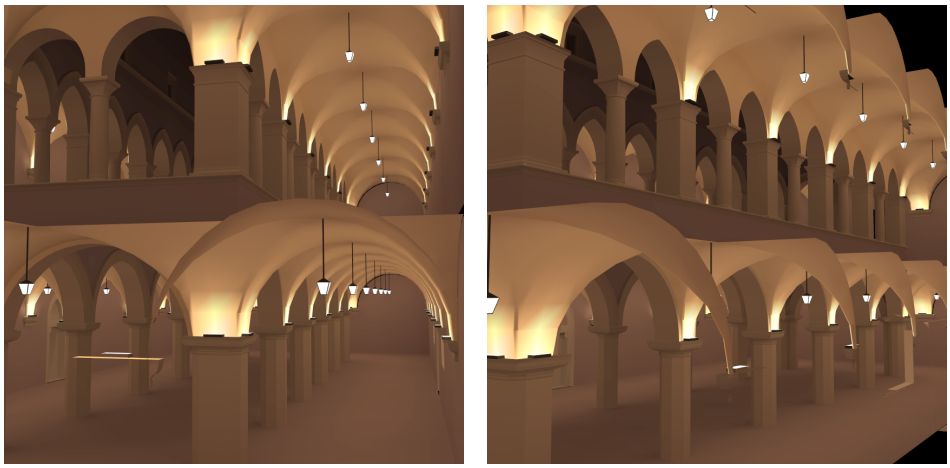


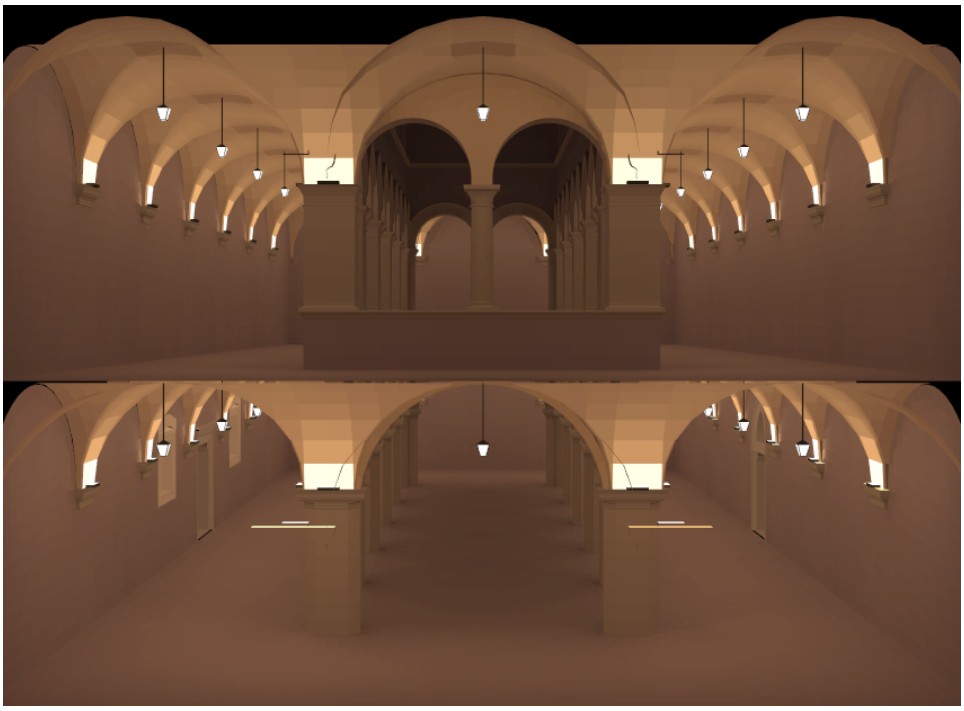
Figura 48: Solución del Palacio Sponza (detalles, con interpolación lineal).

mente dicho, sino un *remallado*, dado que además de mallar había también que interpretar y reparar la topología original.

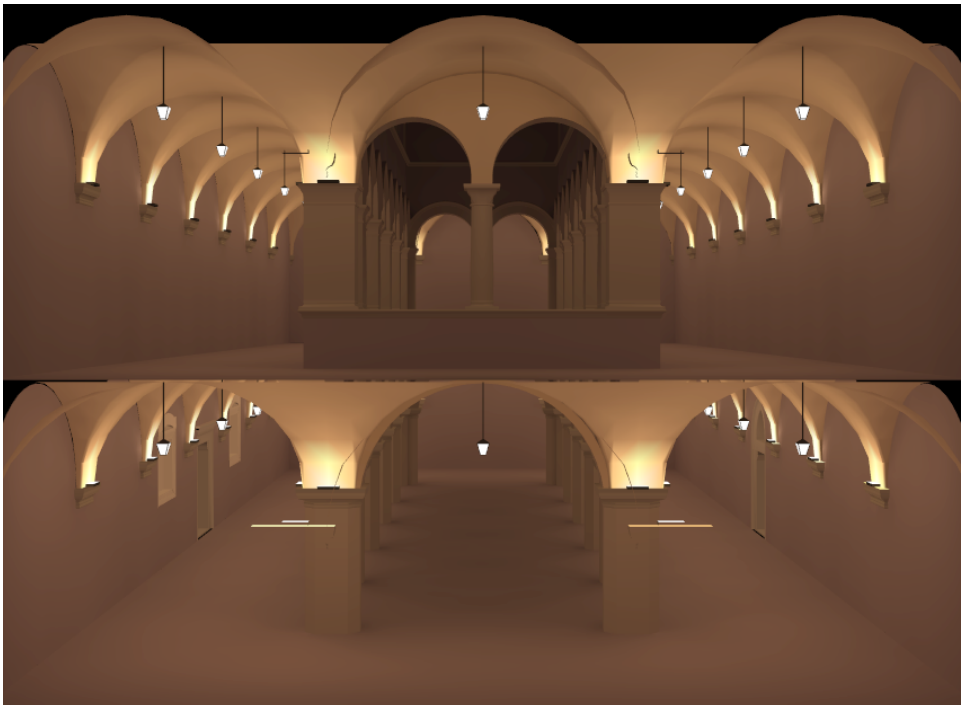
El mallado cuadrangular, con el postproceso de muelles ya aplicado, puede verse en las figuras 50 y 51. En total se han generado 102135 elementos, cuyos factores de forma ocupan cerca de 40 GB (se han almacenado, pese al espacio requerido, por la ganancia de rendimiento que ello proporciona).

Se ha optado por hacer una simulación nocturna[(40)]. El modelo de Marko Dabrovic incluye texturas para las superficies, pero por simplicidad se ha obtenido la solución sin texturas, empleando cromaticidades diferentes a las que realmente posee el Palacio.

[(40)] La *radiosidad* también se puede aplicar a luz solar, mediante un modelo apropiado de la radiación del cielo en cada dirección, pero no obstante a efectos de este análisis se ha tomado iluminación nocturna.



(a) Solución sin interpolar.



(b) Solución con interpolación lineal.

Figura 49: Solución del Palacio Sponza (frontal).

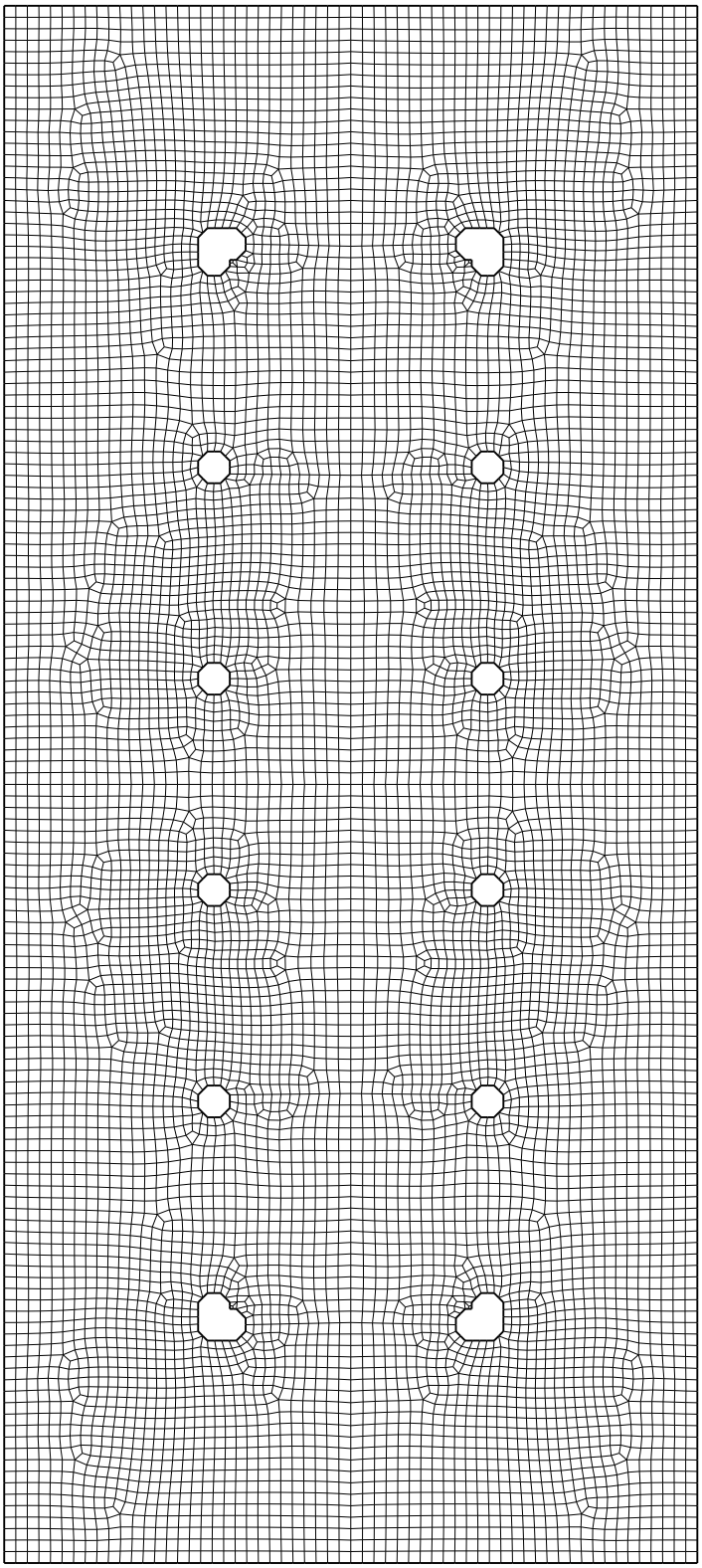
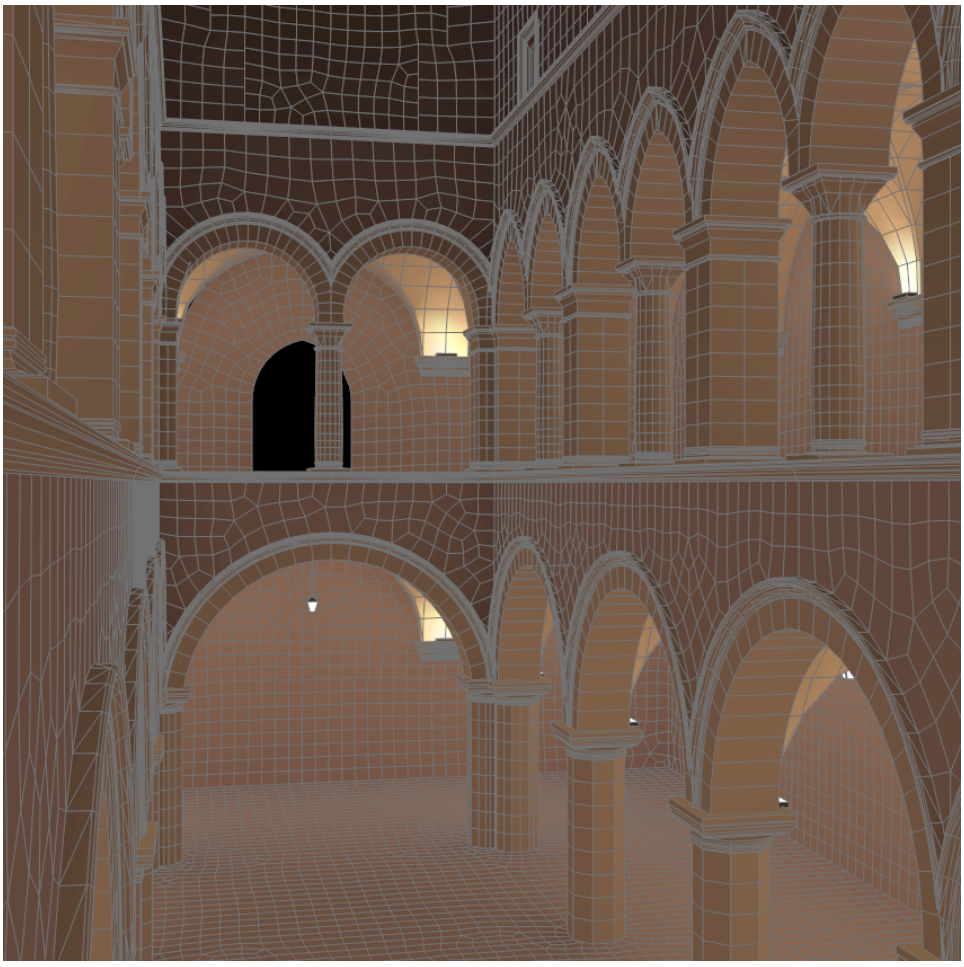
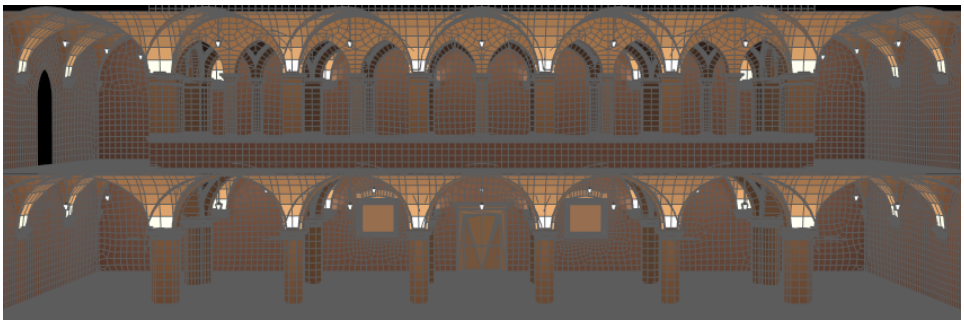


Figura 50: Mallado de la planta del Palacio Sponza.



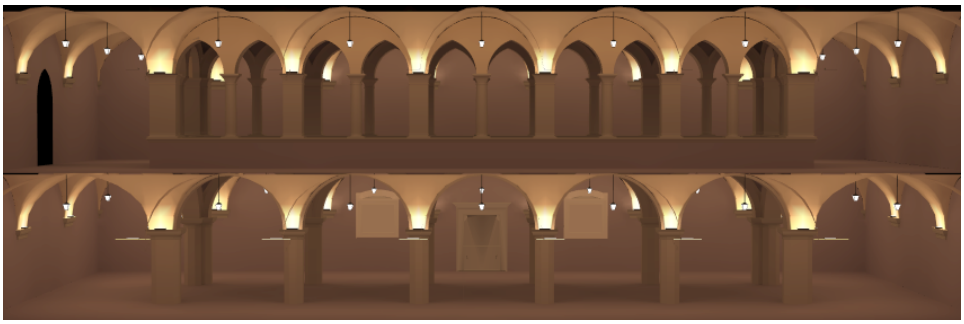
(a) Detalle mallado interior del atrio.



(b) Alzado longitudinal del mallado.

Figura 51: Mallado del Palacio Sponza.

(a) Detalle solución interior del atrio, con interpolación lineal.



(b) Alzado longitudinal de la solución, con interpolación lineal.

Figura 52: Solución del Palacio Sponza.

4. Conclusiones

Un uso eficiente del *hardware* actual, tanto en lo que se refiere a *CPUs* multinúcleo como a *GPUs*, ofrece unas posibilidades que a menudo no están siendo suficientemente aprovechadas. Las mediciones de tiempo obtenidas en el apartado *"Eficiencia en CPU y GPU de los algoritmos propuestos"* así lo indican.

Empleando herramientas como *OpenCL* o *CUDA*, que permiten explotar estos recursos desde una sintaxis estándar de lenguajes de alto nivel como *C* y *C++*, es posible programar dispositivos actuales de muy diversa naturaleza sin necesidad de recurrir a lenguaje de bajo nivel. Un conocimiento de las diferencias del diseño entre una *CPU* y una *GPU* resulta sin embargo necesario.

Se ha observado que el algoritmo de mallado cuadrangular presentado por Sarrate y Huerta en [50] es robusto y produce excelentes resultados, teniendo además un diseño muy sencillo. En el apartado *"Modificaciones al algoritmo de mallado"* se han sugerido algunas modificaciones y mejoras al mismo.

Con todo ello en consideración, la implementación realizada es capaz de mallar de manera fiable y eficiente no sólo una bateria de contornos de prueba, sino todos los dominios de los ejemplos mostrados en el apartado *"Ejemplo de aplicación. Radiosidad"*, aportando un importante banco de pruebas, que la implementación ha mallado muy satisfactoriamente. El trabajo realizado no se encuentra agotado, cabe enfrentar la implementación a más casos prácticos, quizás detectando futuras posibles mejoras.

La ejecución paralelizada del mallado, en *CPU* y *GPU*, ha obtenido incrementos de rendimiento de hasta 4 a 6 veces el del modo secuencial en un núcleo de ejecución. No se ha diseñado una versión específica del algoritmo para arquitecturas *SIMD*, que presumiblemente mejoraría aún más la eficiencia del mallado en *GPU*. Esta particularización se vislumbra como un futuro frente de trabajo, sin perder de vista que implementaciones muy específicas dificultan el mantenimiento del programa, y en algunos casos pueden dejar de ser razonables.

Un escenario que inicialmente no había sido previsto es el caso de mallar modelos partiendo de una definición geométrica que presente incorrecciones topológicas. Dichos problemas son habituales en los archivos generados por programas de modelado 3D. Siendo muy probable que la aplicación práctica del mallado cuadrangular requiera su empleo en modelos que inicialmente

posean errores de topología (como se ha visto en el apartado *"Ejemplo de aplicación. Radiosidad"*), se abre un posible campo de trabajo en el ámbito del *remallado* de geometría, aplicándolo al mallado cuadrangular.

El método propuesto de postproceso para mejora de calidad basado en muelles se ha mostrado igualmente óptimo. Se ha aplicado en todos los ejemplos realizados, siendo el que mayor calidad aportaba de entre todas las alternativas ensayadas. La formulación aquí presentada se basa en la *distorsión de Oddy*, pero sería adaptable a otras medidas de distorsión de cuadriláteros, aspecto que puede ser de interés.

Por último, pero no menos importante, ha resultado especialmente motivadora la aplicación de los algoritmos de mallado y postproceso al método de *radiosidad*. Tratándose de un método originalmente enfocado a elementos cuadrangulares, cuyo uso se ha visto frenado parcialmente por la dependencia de algoritmos de mallado robusto y de calidad, se detecta que el trabajo realizado podría abrir nuevas vías de aplicación en la *radiosidad*. Se sugiere estudiar estas posibilidades, profundizando en detalles que sobrepasan el ámbito de esta Tesis Doctoral.

5. Referencias

[1] Baehmann, P.L., Wittchen, S.L., Shephard, M.S., Grice, K.R., and Yerry, M.A.: *Robust, geometrically based, automatic two-dimensional mesh generation.* International Journal for Numerical Methods in Engineering, 24(6):1043–1078, 1987.

[2] Bastian, M. and Li, B.Q.: *An efficient automatic mesh generator for quadrilateral elements implemented using C++.* Finite Elements in Analysis and Design, 39(9):905–930, 2003.

[3] Blacker, T.D. and Stephenson, M.B.: *Paving: A new approach to automated quadrilateral mesh generation.* International Journal for Numerical Methods in Engineering, 32(4):811–847, 1991.

[4] Bollig, E.F.: *Centroidal voronoi tessellation of manifolds using the GPU.* Master's thesis, Florida State University, 2008.

[5] Canann, S.A., Tristano, J.R., and Staten, M.L.: *An approach to combined Laplacian and optimization-based smoothing for triangular, quadrilateral, and quad-dominant meshes.* In *Proceedings of the 7th International Meshing Roundtable*, pp. 479–494, 1998.

[6] Cao, T.T., Nanjappa, A., Gao, M., and Tan, T.S.: *A GPU accelerated algorithm for 3D Delaunay triangulation.* In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '14, pp. 47–54, New York, NY, USA, 2014. ACM.

[7] Cardano, G.: *Artis Magnæ, Sive de Regulis Algebraicis Liber Unus.* 1545.

[8] Chae, S.W. and Jeong, J.H.: *Unstructured surface meshing using operators.* In *Proceedings, 6th International Meshing Roundtable*, pp. 281–291, 1997.

[9] Clark, J.H.: *The Geometry Engine: A VLSI geometry system for graphics.* In *Proceedings of the 9th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '82, pp. 127–133. ACM, 1982.

[10] Cohen, M., Greenberg, D., Immel, D., and Brock, P.: *An efficient radiosity approach for realistic image synthesis.* IEEE Comput. Graph. Appl., 6(3):26–35, Mar. 1986.

[11] Cohen, M.F., Chen, S.E., Wallace, J.R., and Greenberg, D.P.: *A progressive refinement approach to fast radiosity image generation.* In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '88, pp. 75–84, New York, NY, USA, 1988. ACM.

[12] Cohen, M.F. and Greenberg, D.P.: *The hemi-cube: A radiosity solution for complex environments*. In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '85, pp. 31–40, New York, NY, USA, 1985. ACM.

[13] Cornell University Program of Computer Graphics: *The Cornell Box*. `http://www.graphics.cornell.edu/online/box/`.

[14] D'Amato, J. and Vénere, M.: *A CPU-GPU framework for optimizing the quality of large meshes*. Journal of Parallel and Distributed Computing, 73(8):1127–1134, 2013.

[15] Eckert, E.R.G. and Drake, R.M.: *Heat and Mass Transfer*. McGraw-Hill, 1959.

[16] Findt, A.P. and Clarke, W.A.: *Delaunay triangulation using a parallel architecture*. In *SATNAC 2011 Conference Papers*, 2011.

[17] Gargallo, A., Roca, X., and Sarrate, J.: *A surface mesh smoothing and untangling method independent of the CAD parameterization*. Computational mechanics, 53(4):587–609, Apr 2014.

[18] Giuliani, S.: *An algorithm for continuous rezoning of the hydrodynamic grid in arbitrary lagrangian-eulerian computer codes*. Nuclear Engineering and Design, 72(2):205–212, 1982.

[19] Goral, C.M., Torrance, K.E., Greenberg, D.P., and Battaile, B.: *Modeling the interaction of light between diffuse surfaces*. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, pp. 213–222, New York, NY, USA, 1984. ACM.

[20] Haissam El-Aawar: *Increasing the transistor count by constructing a two-layer crystal square on a single chip*. International Journal of Computer Science & Information Technology (IJCSIT), 7(3), 2015.

[21] Hottel, H.C.: *Radiant heat transmission*. In McAdams, W.H. (ed.): *Heat Transmission*. McGraw-Hill, 1954.

[22] Howell, J., Menguc, M., and Siegel, R.: *Thermal Radiation Heat Transfer, 6th Edition*. CRC Press, 2015.

[23] Joe, B.: *Quadrilateral mesh generation in polygonal regions*. Computer-Aided Design, 27(3):209–222, 1995.

[24] Johnston, B.P., Sullivan, J.M., and Kwasnik, A.: *Automatic conversion of triangular finite element meshes to quadrilateral elements*. International Journal for Numerical Methods in Engineering, 31(1):67–84, 1991.

[25] Kandasamy, V. and König, M.: *Parallel finite element mesh generator using multiple GPUs*. In *Proceedings of the 9th Europeen Conference on Product and Process Modeling 2012*. Moscow State University of Civil Engineering, Publishing House "ASV", 2012.

[26] Kempf, R. and Hartman, J.: *OpenGL on Silicon Graphics Systems*. Silicon Graphics, Inc, 3rd ed., 2005.

[27] Knupp, P.M.: *Algebraic mesh quality metrics*. SIAM J. Sci. Comput., 23(1):193–218, Jan. 2001.

[28] Lee, C.K. and Lo, S.H.: *A new scheme for the generation of a graded quadrilateral mesh*. Computers & Structures, 52(5):847–857, 1994.

[29] Lo, S.H.: *Generating quadrilateral elements on plane and over curved surfaces*. Computers & Structures, 31(3):421–426, 1989.

[30] Lo, S.H.: *Finite Element Mesh Generation*. CRC Press, 2015.

[31] Lohner, R., Morgan, K., and Zienkiewicz, O.C.: *Adaptive grid refinement for the compressible Euler equations*. In *Accuracy estimates and adaptive refinements in finite element computations*, pp. 281–297. Wiley, 1986.

[32] Martínez, J. L., León, J., Martín-Caro, J. A. y Corres, H.: *Análisis de la sección transversal de la Catedral de Palma de Mallorca*. Congreso ACHE "Puentes y Estructuras de Edificación", 2002.

[33] McGuire, M.: *Computer graphics archive*, August 2011. `http://graphics.cs.williams.edu/data`.

[34] Moore, G.E.: *Cramming more components onto integrated circuits*. Electronics, pp. 114–117, 1965.

[35] Nanjappa, A.: *Delaunay Triangulation in R3 on the GPU*. PhD thesis, Department Of Computer Science National University Of Singapore, 2012.

[36] Navarro, C., Hitschfeld, N., and Scheihing, E.: *A parallel GPU-based algorithm for Delaunay edge-flips*. In Hoffmann, M. (ed.): *27th European Workshop on Computational Geometry (EuroCG)*, pp. 75–78, Morschach, Switzerland, Mar 2011.

[37] Nickalls, R.W.D.: *Viète, Descartes and the cubic equation*. The Mathematical Gazette, 90(518):203–208, July 2006.

[38] Nishita, T. and Nakamae, E.: *Continuous tone representation of three-dimensional objects taking account of shadows and interreflection*. In

Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '85, pp. 23–30, New York, NY, USA, 1985. ACM.

[39] Nowottny, D.: *Quadrilateral mesh generation via geometrically optimized domain decomposition.* In *Proceedings, 6th International Meshing Roundtable*, pp. 309–320, 1997.

[40] NVIDIA: *CUDA C Programming Guide*. Version 7.5 ed., 2015.

[41] NVIDIA: *Tesla P100 whitepaper.* Techn. rep., 2016.

[42] Oddy, A., Goldak, J., McDill, M., and Bibby, M.: *A distortion metric for iso-parametric finite elements.* CSME Transactions, 12(4):213–217, 1988.

[43] Owen, S.J.: *A survey of unstructured mesh generation technology.* In *Proceedings, 7th International Meshing Roundtable*, pp. 239–267, 1998.

[44] Owen, S.J., Staten, M.L., Canann, S.A., and Saigal, S.: *Q-morph: An indirect approach to advancing front quad meshing.* International Journal for Numerical Methods in Engineering, 44(9):1317–1340, 1999.

[45] Qi, M., Cao, T.T., and Tan, T.S.: *Computing 2D constrained Delaunay triangulation using graphics hardware.* Techn. rep., 2011.

[46] Qi, M., Cao, T.T., and Tan, T.S.: *Computing 2D constrained Delaunay triangulation using the GPU.* In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '12, pp. 39–46, New York, NY, USA, 2012. ACM.

[47] Rong, G.: *Jump Flooding Algorithm on Graphics Hardware and its Applications.* PhD thesis, Department Of Computer Science National University Of Singapore, 2007.

[48] Rong, G., Tan, T.S., Cao, T.T., and Stephanus: *Computing two-dimensional Delaunay triangulation using graphics hardware.* In *SI3D*, pp. 89–97, 2008.

[49] Sarrate, J. y Coll, A.: *Minimización de la distorsión de mallas formadas por cuadriláteros o hexaedros.* Revista internacional de métodos numéricos para cálculo y diseño en ingeniería, 23(1):55–76, Ene. 2007.

[50] Sarrate, J. and Huerta, A.: *Efficient unstructured quadrilateral mesh generation.* International Journal for Numerical Methods in Engineering, 49(10):1327–1350, 2000.

[51] Scarpino, M.: *OpenCL in Action.* Manning Publications, 2012.

[52] Shewchuk, J.R.: *Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator.* In Lin, M.C. and Manocha, D. (eds.): *Applied Computational Geometry: Towards Geometric Engineering*, vol. 1148 of *Lecture Notes in Computer Science*, pp. 203–222. Springer-Verlag, May 1996.

[53] Shreiner, D., Sellers, G., Kessenich, J.M., and Licea-Kane, B.M.: *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3.* Addison-Wesley Professional, 8th ed., 2013.

[54] Sillion, F.X. and Puech, C.: *Radiosity and global illumination.* Morgan Kaufmann, 1994.

[55] Talbert, J.A. and Parkinson, A.R.: *Development of an automatic, two-dimensional finite element mesh generator using quadrilateral elements and Bezier curve boundary definition.* International Journal for Numerical Methods in Engineering, 29(7):1551–1567, 1990.

[56] Tam, T.K.H. and Armstrong, C.G.: *2D finite element mesh generation by medial axis subdivision.* Advances in Engineering Software and Workstations, 13(5-6):313–324, 1991.

[57] Upstill, S.: *RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics.* Addison-Wesley Longman Publishing Co., Inc., 1989.