Trabajo Fin de Máster:

Modelización de sólidos para el análisis isogeométrico

Las Palmas de Gran Canaria, Junio de 2012

Tutores:
José María Escobar Sánchez
Rafael Alejandro Montenegro Armas

Alumno:
José Iván López González

Índice

1.- Introducción

El análisis isogeométrico (IGA) [1, 2] busca unificar los campos del diseño asistido por ordenador (CAD) y el análisis de elementos finitos (FEM) utilizado para la resolución de problemas en ingeniería. El análisis isogeométrico es una variante del FEM que toma como funciones de base las mismas que describen la geometría (Splines, NURBS, T-spline) [3, 4]. Así, la distancia que separa el CAD de la resolución del problema por elementos finitos desaparece en gran medida.

Los modelos construidos a partir de CAD definen exclusivamente la superficie del sólido, pero la aplicación de IGA requiere una descripción volumétrica del sólido. Uno de los principales problemas del IGA consiste en obtener una representación de un sólido con splines a partir de la descripción de la frontera del sólido proporcionada por un CAD. Tal como señala Cotrell y otros [1] "the most significant challenge facing isogeometric analysis is developing three-dimensional spline parameterizations from surfaces". En este sentido, en la división de Discretización y Aplicaciones, junto con la división de Álgebra Numérica Avanzada, del Instituto Universitario de Sistemas Inteligentes y Aplicaciones Numéricas en Ingeniería (SIANI) de la ULPGC, estamos trabajando en la construcción de un novedoso método para la representación T-spline de sólidos complejos para la aplicación del análisis isogeométrico.

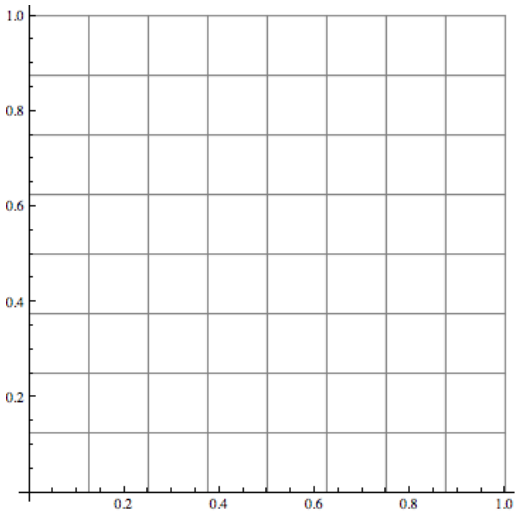
Sólo hay unos pocos trabajos que tratan este problema [5, 6], y todos ellos tienen en común el uso de funciones armónicas para establecer la parametrización volumétrica. En el SIANI hemos realizado un enfoque diferente [7] en el que la parametrización volumétrica se realiza mediante la transformación a una malla de tetraedros desde el dominio paramétrico al dominio físico [7, 8, 9]. Ésta es la característica principal de nuestro método; no tenemos que dar la malla de tetraedros del sólido como entrada, ya que se obtiene como resultado del proceso de parametrización. Otra característica de nuestro trabajo, es que usamos un esquema de interpolación para adecuar una B-spline cúbica a los datos, en lugar de una aproximación por mínimos cuadrados, como hacen otros autores. Esto proporciona una adaptación más precisa de la T-spline a los datos de entrada.

El uso de T-splines en el IGA [3] está motivado por uno de los principales inconvenientes de las NURBS: la necesidad de un espacio paramétrico con una estructura de producto tensorial, haciendo ineficiente la representación de características localmente detalladas. Este problema se resuelve con las T-splines, una generalización de NURBS formulada por Sederberg y otros [4] que permite el refinamiento local. Las T-splines son un conjunto de funciones definidas en una *T-mesh*, esto es, un entramado de prismas rectangurales en R^3 en el que se permiten *uniones en T*.

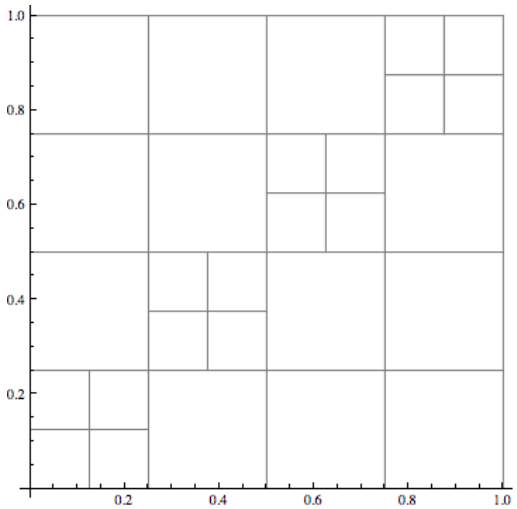
2.- Contextualización del trabajo

Una de las principales dificultades con las que se encuentra el análisis isogeométrico es la representación geométrica del dominio en estudio. La utilización de T-splines permite un refinamiento de características localmente detalladas, a diferencia de NURBS, donde el refinamiento se realiza de forma global, lo que hace ineficiente las representaciones locales.

La construcción de mallas que dan soporte a las T-splines conlleva algunas dificultades a la hora de elaborar el código de forma eficiente y deja abiertas cuestiones de carácter técnico y teórico. En este trabajo se pretende implementar una técnica eficaz para la generación de la *T-mesh*, es decir, del espacio paramétrico en el que están definidas las T-splines, mediante la construcción de estructuras *quadtree* en 2D y estructuras *octree* en 3D. Además, se desarrollará un método de equilibrado o balanceo que permita obtener mallas con transiciones suaves desde zonas con alto refinamiento a zonas poco refinadas, mejorando así la calidad de la malla y por tanto, de la solución numérica obtenida con el método.



Refinamiento global (NURBS)



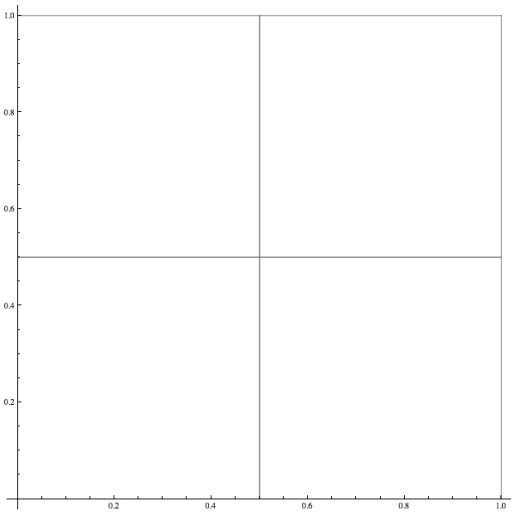
Refinamiento local (T-spline)

3.- Fundamentos Teóricos del Trabajo

Las mallas utilizadas en el análisis isogeométrico con T-spline se construyen mediante divisiones recursivas del espacio conocidas como divisiones *quadtree* en 2D u *octree* en 3D [11]. En esta sección se presentan estas estructuras y su relación con el análisis isogeométrico.

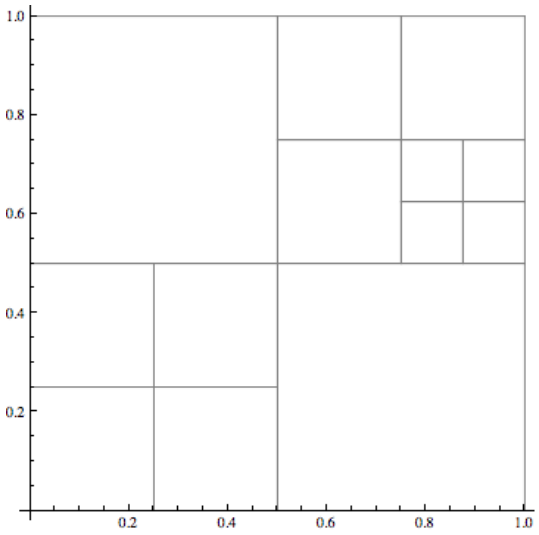
3.1.- Estructura *quadtree*

El término *quadtree* se utiliza para describir clases de estructuras de datos jerárquicas cuya propiedad común es que están basados en el principio de descomposición recursiva del espacio. El primer ejemplo de un *quadtree* se relaciona a la representación de un área bidimensional mediante la subdivisión sucesiva del espacio en cuatro cuadrantes del mismo tamaño.



Quadtree con división inicial en cuatro cuadrantes del mismo tamaño.

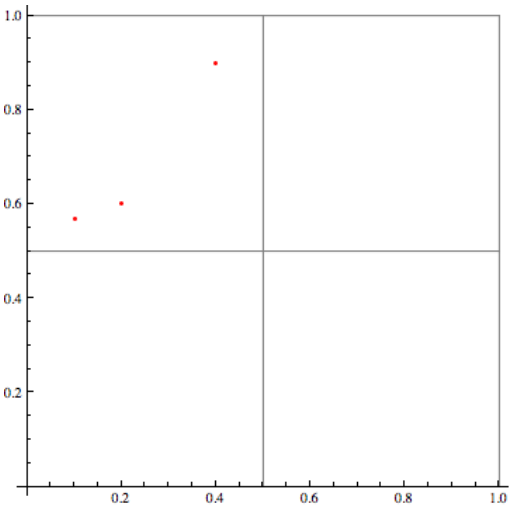
La estructura *quadtree* se representa gráficamente como un cuadrado que es dividido en cuatro cuadrantes de igual tamaño. Cada cuadrante resultante puede ser divido a su vez en otros cuatro, y esta división se realiza de forma sucesiva en función de algún criterio de construcción del *quadtree*.



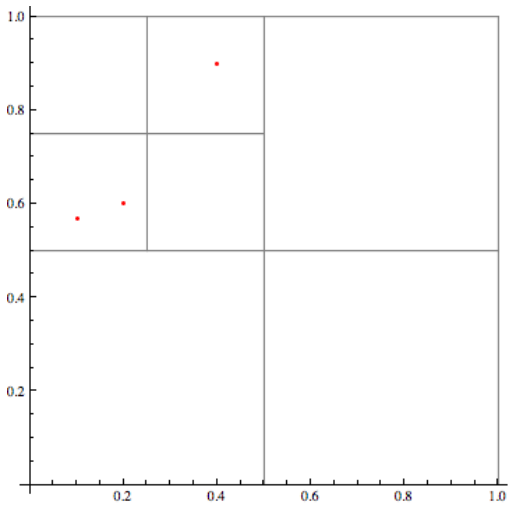
Quadtree con varias subdivisiones.

La división del espacio se realiza conforme a determinados criterios que pueden ser modificados según el caso de estudio. Por ejemplo, si se dispone de una serie de puntos resultantes de la discretización de una superficie, una condición de subdivisión en el *quadtree* puede consistir en limitar el número de puntos que pueden estar contenidos en un mismo cuadrante. Un caso habitual es construir un *quadtree* de modo que cada celda contenga como mucho un punto de la discretización.

Siguiendo ese criterio para la construcción del *quadtree*, en la siguiente figura se aprecia un cuadrante que contiene más de un punto, por lo que se procede a subdividir ese cuadrante.

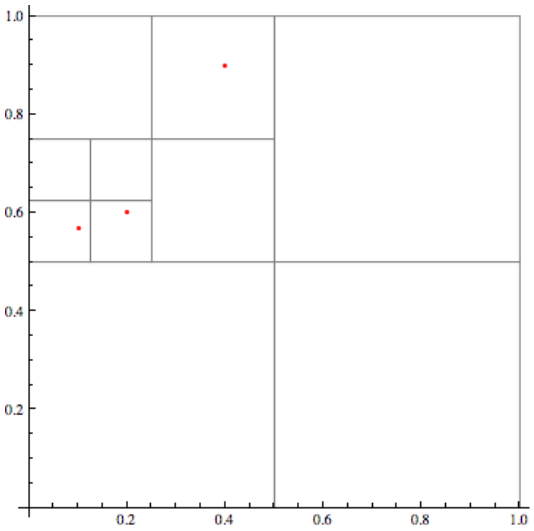


Cuadrante con más de un punto



Subdivisión del cuadrante

Tras la subdivisión, uno de los nuevos cuadrantes resultantes continúa con más de un punto en su interior, por lo tanto debe ser subdividido.



Quadtree resultante.

Tras esta última subdivisión, cada cuadrante contiene como máximo un punto, por lo que ya no es necesario continuar con la subdivisión del espacio.

Según el criterio de construcción seleccionado, la estructura *quadtree* se pude utilizar para construir una malla de cuadriláteros que aproximen la superficie de un sólido a partir de una discretización inicial de puntos.

Desde el punto de vista computacional, un *quadtree*, como su propio nombre indica, se trata de una estructura de árbol en la que cada nodo no hoja tiene siempre cuatro hijos.

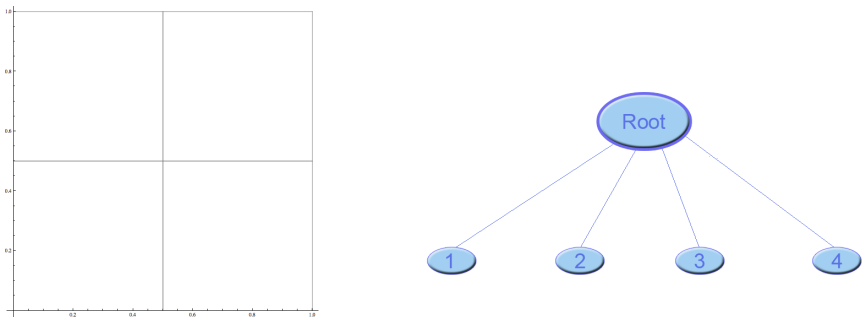
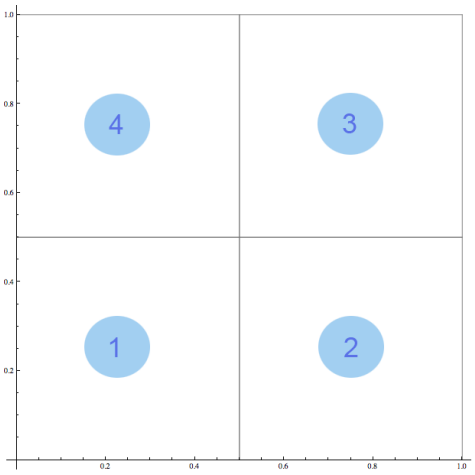


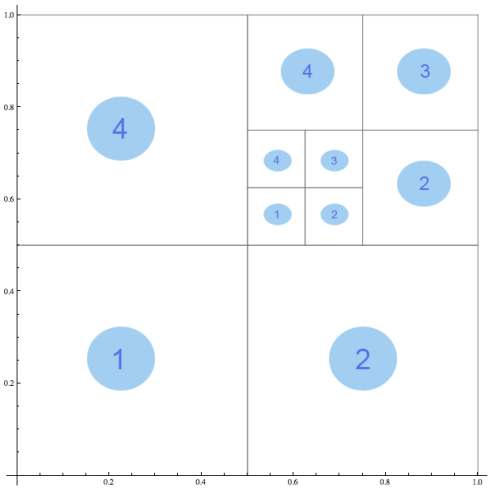
Figura *quadtree* básica y su árbol equivalente.

El cuadrilátero inicial se corresponde con la raíz del árbol (root). Cada celda del *quadtree* está representada por un nodo hoja del árbol y además cada celda es enumerada, de modo que se pueda determinar directamente qué nodo del árbol corresponde con cada celda. La forma de enumerar las celdas es tal que la celda 1 es la de abajo-izquierda, la celda 2 abajo-derecha, la celda 3 arriba-derecha y celda 4 arriba-izquierda.



Quadtree con enumeración de celdas.

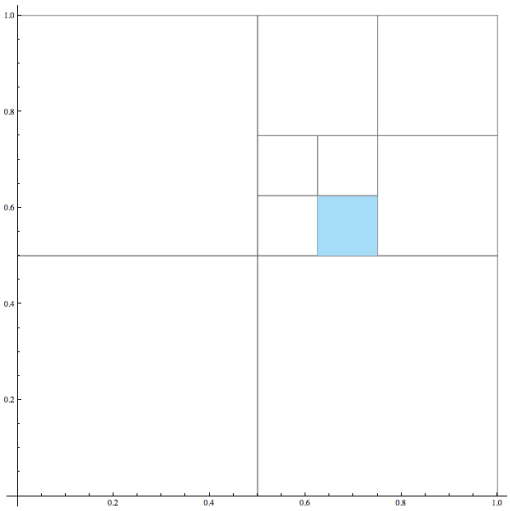
Cuando se realiza la subdivisión de una celda, las nuevas celdas resultantes se vuelven a enumerar. De este modo, cada celda tendrá asociado una secuencia numérica que indicará los descensos que se deben realizar en el árbol para alcanzar el nodo hoja correspondiente a la celda.



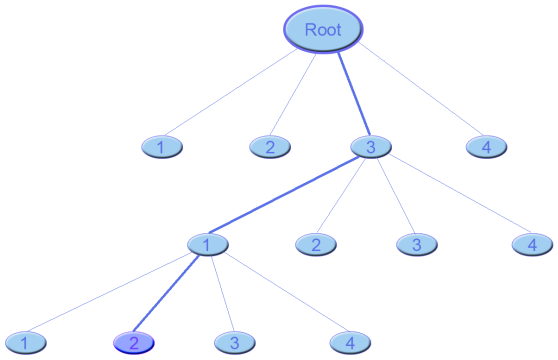
Quadtree con sucesivas enumeraciones.

Cada celda, además, se encuentra en un nivel de altura del árbol. El nivel en el que se encuentra la celda está relacionado con el número de subdivisiones que se han llevado a cabo para obtenerla. De este modo, las cuatro celdas iniciales creadas en la primera división están en el nivel 1 del árbol, las celdas generadas a partir de éstas están en el nivel 2 y así sucesivamente.

La secuencia que identifica una celda indica también el nivel de profundidad al que se encuentra en el árbol el nodo correspondiente a la celda. En la siguiente figura se muestra en azul la celda que corresponde la secuencia {3,1,2}, de la que se puede obtener que el nivel de profundidad en el árbol es 3.



Celda {3,1,2} del *quadtree*.



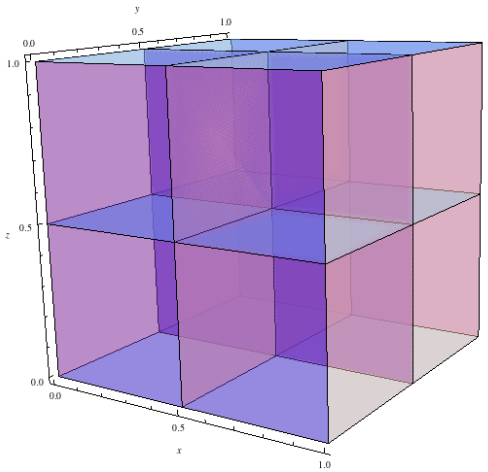
Descenso por el árbol a partir de la secuencia.

Por tanto, cada nodo hoja del árbol *quadtree* está identificado por una secuencia que indica todos sus ancestros.

3.2.- Estructura octree

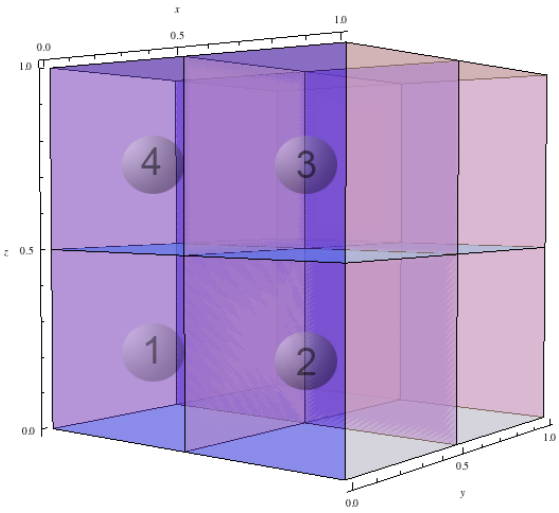
Un *octree* es una estructura en árbol de datos en la cual cada nodo interno tiene exactamente 8 hijos. Las estructuras *octree* se utilizan principalmente para particionar un espacio tridimensional, dividiéndolo recursivamente en ocho octantes. Las estructuras *octree* son las análogas tridimensionales de los *quadtree* bidimensionales, pasando de un cuadrilátero a un cubo.

El principio de construcción del *octree* es exactamente igual que en la estructura *quadtree*. El espacio se subdivide en octantes en función de los criterios establecidos, por ejemplo, cada octante sólo puede contener un punto de la discretización de entrada.

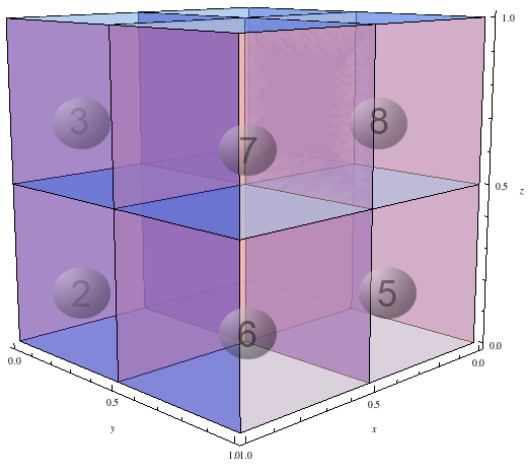


División *octree* inicial.

A cada octante o celda del *octree* se le asocia una numeración similar a la utilizada en la estructura *quadtree*. La siguiente figura muestra la numeración empleada.



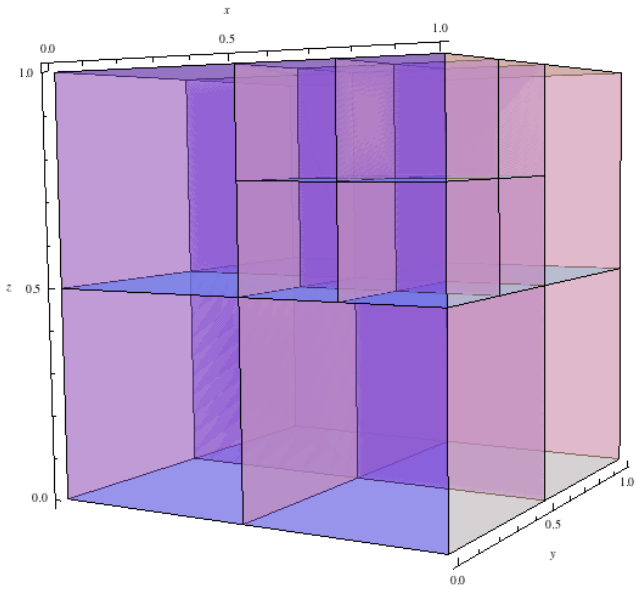
Enumeración de celdas frontales

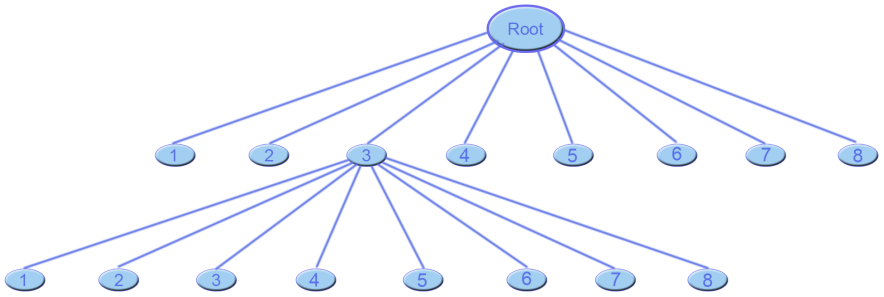


Enumeración de celdas traseras

Las cuatro celdas frontales se enumeran de modo que la celda 1 corresponde con la celda situada abajo-izquierda, la celda 2 abajo-derecha, celda 3 arriba-derecha y celda 4 arriba-izquierda. En cuanto a las celdas de la parte trasera, la enumeración es celda 5 abajo-izquierda, celda 6 abajo-derecha, celda 7 arriba-derecha y celda 8 arriba-izquierda.

Su representación en forma de árbol también es análoga a la estructura *quadtree*, con la única diferencia de que cada nodo no hoja se subdivide en ocho hijos en lugar de cuatro. La raíz del árbol representa en este caso el cubo inicial de partida.


Representación de *octree* con dos niveles

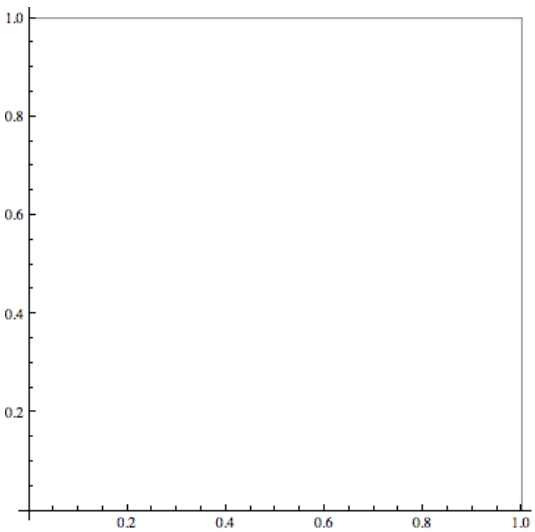


Árbol *octree* de la representación anterior

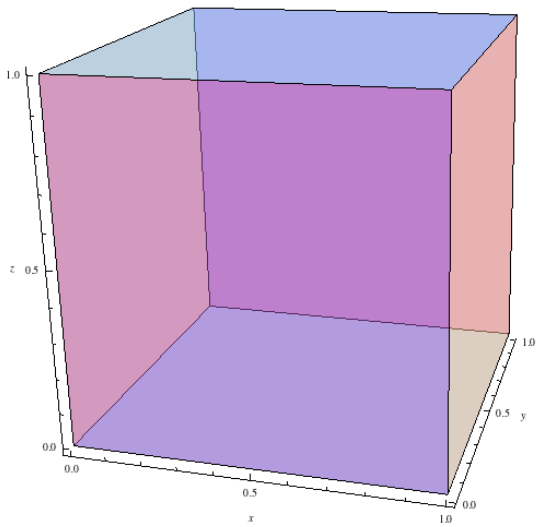
Al igual que en la estructura *quadtree*, cada nodo del árbol está identificado por una secuencia numérica que indica todos sus ancestros, por ejemplo {3, 2, 6, 1}.

3.3.- Mallas en el análisis isogeométrico

El espacio de referencia donde están definidas las funciones T-spline en el análisis isogeométrico, a partir de donde se construye la *T-mesh*, consiste en un cuadrilátero de lado uno con vértices en las coordenadas x e y en (0, 0) y (1, 1) para la versión 2D y un hexaedro de lado uno con vértices en (0, 0, 0) y (1, 1, 1) para la versión 3D.



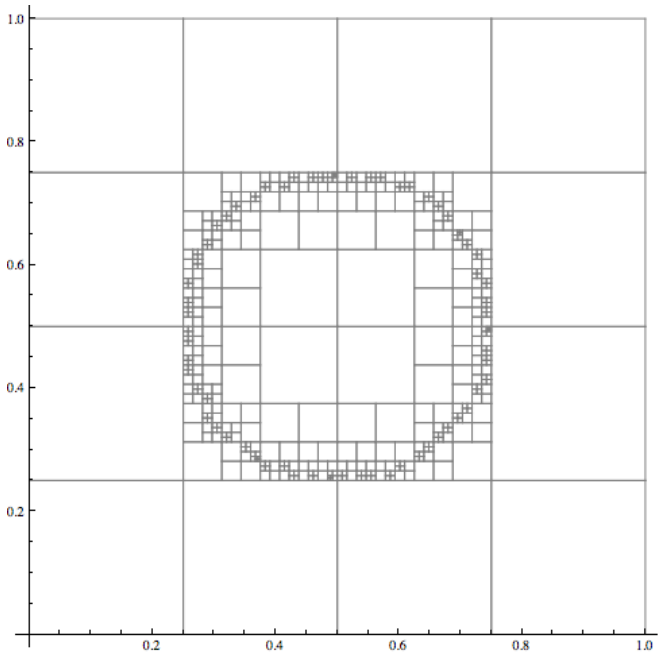
Espacio paramétrico 2D. $C = [0, 1]^2$



Espacio paramétrico 3D. $C = [0, 1]^3$

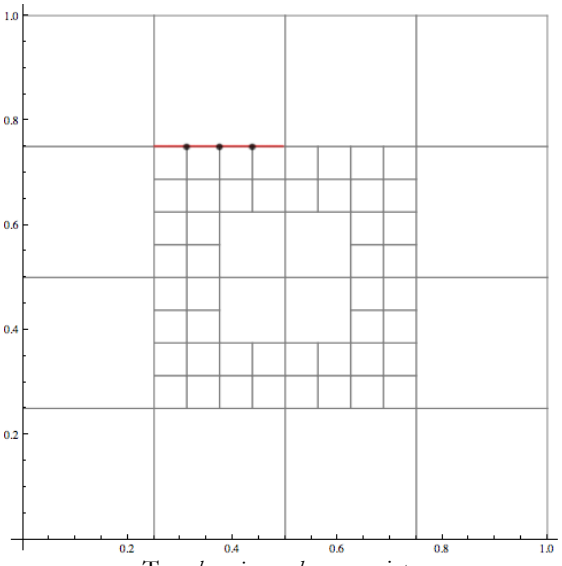
A partir del cuadrilátero o hexaedro de referencia se construye la *T-mesh* para aproximar el objeto o superficie sobre la que se obtiene la solución numérica mediante análisis isogeométrico.

Un condición importante en la generación de una *T-mesh* para el análisis isogeométrico es garantizar como máximo un *hanging node* por arista. Cuando se construye una *T-mesh* puede ocurrir que se produzcan transiciones bruscas desde zonas con un alto refinamiento a zonas poco refinadas. Esto puede llevar a soluciones numéricas con mayor error en la transición de una celda de mayor tamaño a otra de un tamaño mucho menor.



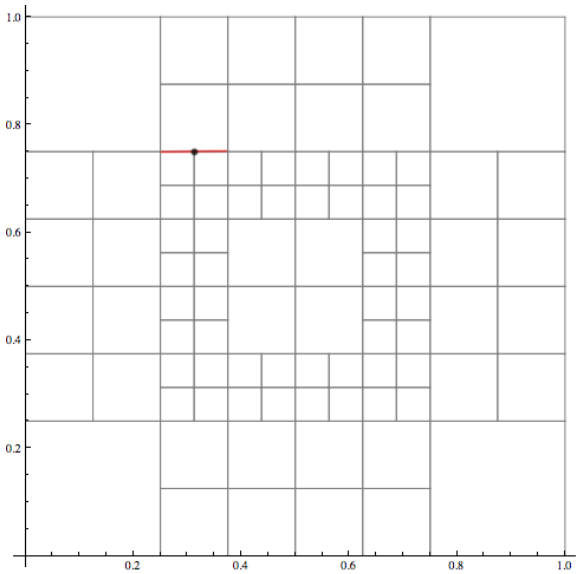
Quadtree con altas transiciones entre celdas

Para evitar este tipo de efectos es conveniente construir la *T-mesh* con un solo *hanging node* por arista. Computacionalmente es equivalente a construir un árbol fuertemente equilibrado o balanceado, donde se garantiza que una celda comparte arista con otra celda como mucho un nivel mayor o inferior.



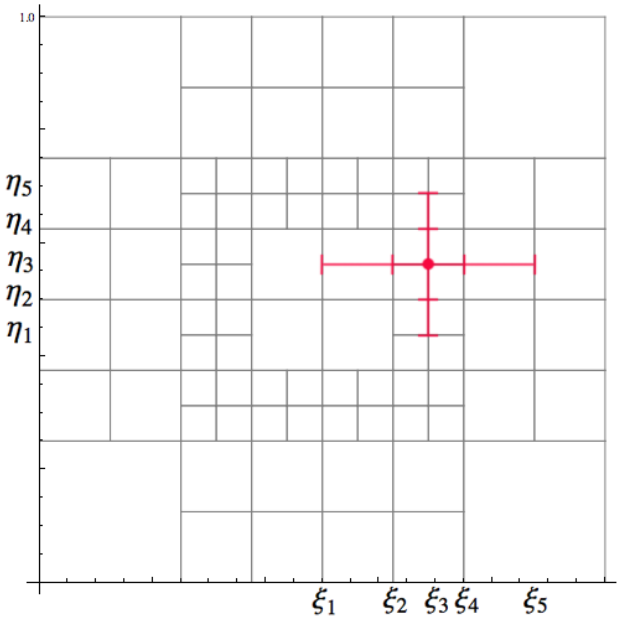
Tres *hanging nodes* por arista

El balanceo del árbol hace que las transiciones entre celdas se realicen de forma suave, por lo que es importante equilibrar el árbol en la generación de mallas *T-mesh* adecuadas para el análisis isogeométrico.



Un solo *hanging node* por arista

Con la *T-mesh* definida, el siguiente paso es obtener los *knots vector* a partir de la malla. Los *knots vector* son quíntuplas que definen las bases de las funciones T-spline. Para cada nodo de la *T-mesh* se obtiene sus dos *knots* o *virtual knots* a la izquierda del nodo, otros dos a la derecha, dos arriba y dos abajo, de modo que se generan dos quíntuplas para el nodo, una con 5 valores en x y otra con 5 valores en y, donde los valores intermedios de las quíntuplas definen las coordenadas x e y del nodo de la *T-mesh*. En 3D se obtiene una tercera quíntupla en la dirección z.



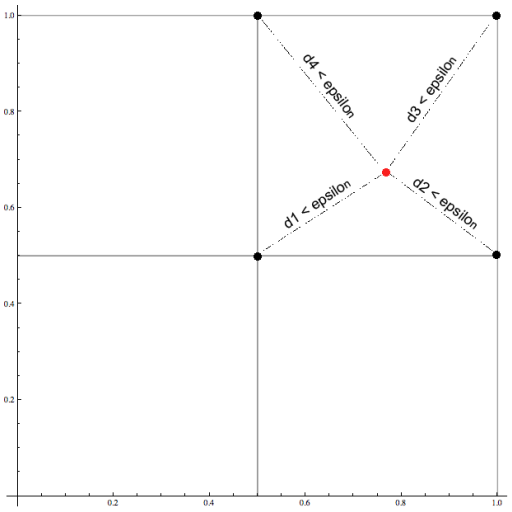
Soporte de una función T-spline: (ξ1, ξ2, ξ3, ξ4, ξ5) × (η1, η2, η3, η4, η5)

4.- Desarrollo

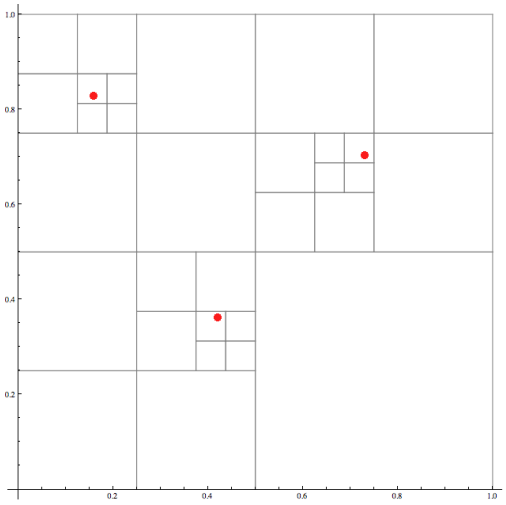
En esta sección se explica el prototipo desarrollado para la construcción del *quadtree* y su versión equilibrada donde sólo se permite un *hanging node* por arista. Luego se expone su extensión a *octree* para la versión 3D.

4.1.- Construcción del *quadtree*

A partir de una discretización del dominio, ya sea a través de una lista de puntos que aproximan el objeto o mediante una triangulación, el objetivo de la estructura *quadtree* es capturar todos los puntos de modo que cada punto quede encerrado en una celda del *quadtree*, logrando así que se aproxime la superficie del objeto mediante una *T-mesh*. Para el caso del análisis isogeométrico se ha considerado que el punto está encerrado en una celda cuando la distancia desde el punto a cada vértice de la celda que lo contiene es menor a una distancia *épsilon* indicada previamente. Esta distancia indica el grado de refinamiento del *quadtree*; cuanto menor es *épsilon* mayor es el refinamiento, ya que las celdas contenedoras serán más pequeñas.



Distancias desde el punto a los vértices


Quadtree que aproxima tres puntos

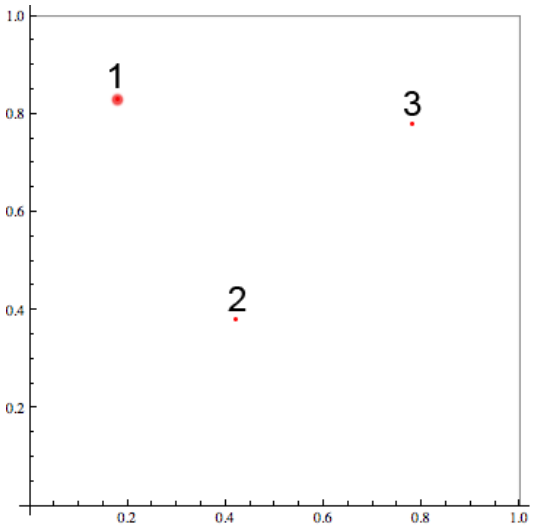
Por tanto, para la construcción del *quadtree* se parte de la discretización inicial del objeto o superficie. Esta discretización será una lista de puntos, coordenadas x e y en 2D, y el algoritmo de construcción del *quadtree* recorre esta lista punto a punto para construir una estructura que encierre todos los puntos de la discretización.

El algoritmo de construcción del *quadtree* consiste en un proceso recursivo en el que se comprueba si el punto cumple las condiciones de cercanía respecto a cada vértice de la celda actual; en caso negativo se decide cuál es el hijo de la celda por el que hay que descender y se vuelve a repetir el proceso. Si no se cumple la condición de cercanía y la

celda actual no tiene descendencia, es decir, se trata de un nodo hoja, entonces se subdivide la celda antes de determinar cuál es la celda por la que se debe descender. Esto se traduce en crear 4 hijos al nodo actual del árbol. El proceso finaliza cuando se alcanza una celda en la que se cumple las condiciones de cercanía del punto respecto a los vértices de la celda.

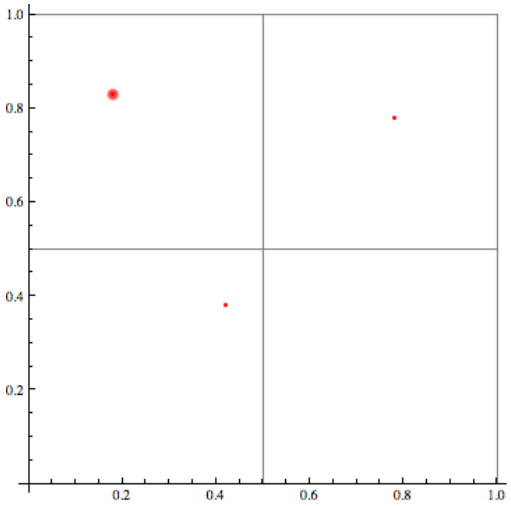
A continuación se muestra gráficamente el proceso de construcción del *quadtree* hasta capturar todos los puntos atendiendo a la condición de cercanía establecida.

Se parte de la lista de puntos inicial que surge de la discretización del objeto o superficie. Se comienza por el primer punto de la lista hasta que quede encerrado en una celda. Para ello se divide recursivamente el espacio.

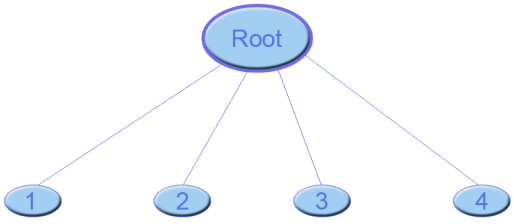


Discretización inicial

Tras la subdivisión se comprueba si el punto queda encerrado en la celda y se repite el proceso hasta que se cumpla la condición de cercanía.

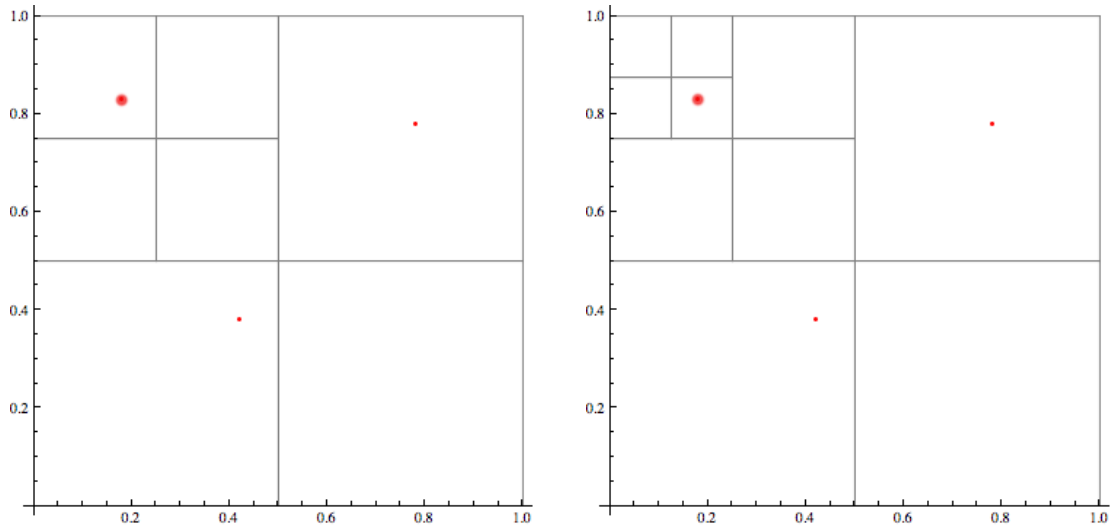


División inicial

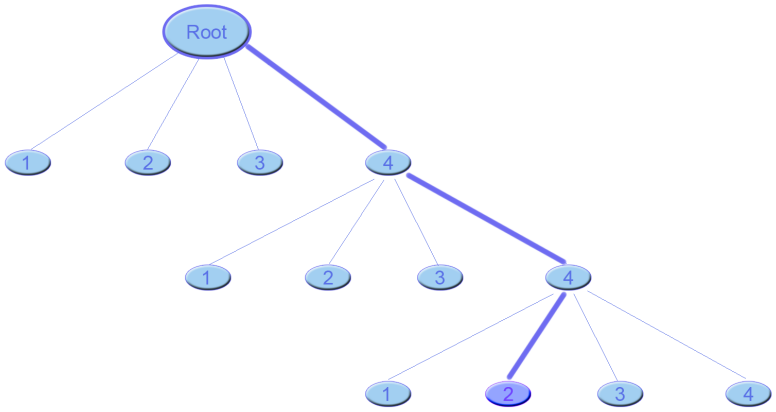


Quadtree tras división inicial

Una vez el punto queda encerrado se procede de igual modo con el siguiente punto de la discretización.



División recursiva del espacio hasta capturar el primer punto



Quadtree tras insertar el primer punto

En el siguiente fragmento de código *Mathematica* se muestra las partes principales del algoritmo de inserción recursiva de un punto para construir el *quadtree*.

```
QTree[points_]:= Module[{root = {}},
   For[i = 1, i <= Length[points], i++,
    root = QInsert[root, points[[i]]];
   ];

   Return[root];
];


(* Función que realiza la inserción *)
QInsert[root_, point_, epsilon_] := Module[{lroot, child, newPoint},
   lroot = root;
   If[QEnclosed[point, epsilon], Return[root]];
   If[Length[root] == 0, lroot = {1, 2, 3, 4}];
```

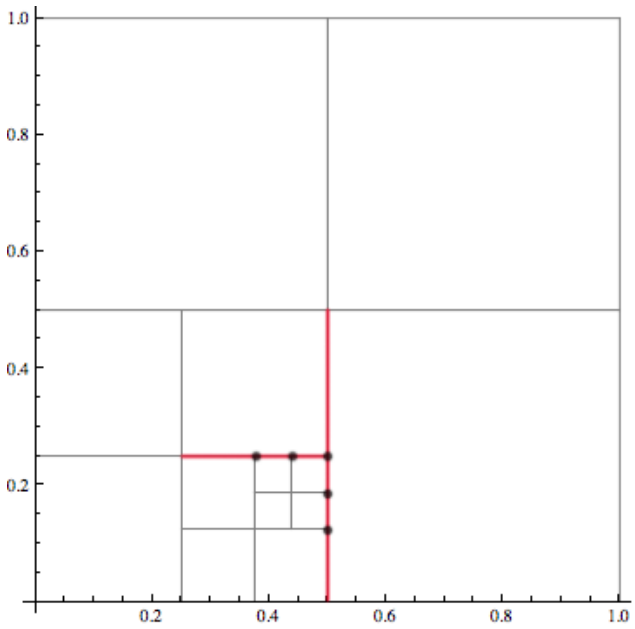
```
    child = QChild[point];
    newPoint = QMovePoint[point, child];
    lroot[[child]] = QInsert[lroot[[child]], newPoint, epsilon*2];
    Return[lroot];
];
```

Código de inserción de un punto

Tras la inserción de todos los puntos se dispone de una *T-mesh* que aproxima el objeto. En esta *T-mesh* generada se producen transiciones bruscas desde celdas grandes a celdas muy pequeñas, por lo que el siguiente paso tras obtener el *quadtree* consiste en equilibrar el árbol de modo que se eviten estas transiciones en la *T-mesh*.

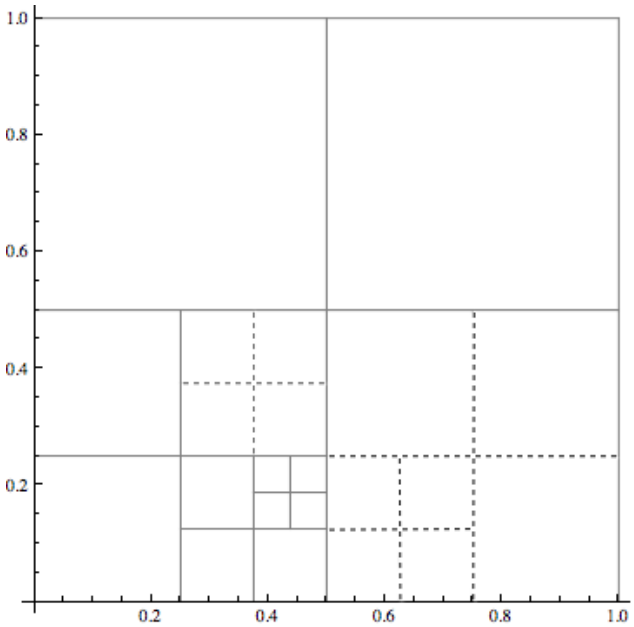
4.2.- Balanceo del quadtree

Balancear o equilibrar el árbol consiste en generar los nodos necesarios, es decir, divisiones en el espacio, de modo que se eviten las transiciones desde celdas grandes a celdas muy pequeñas. Gráficamente se traduce en generar una *T-mesh* donde sólo se permite la existencia de un *hanging node* por arista.



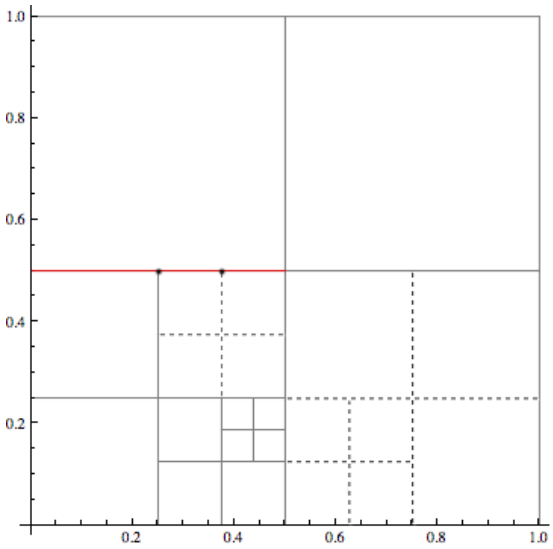
Aristas con más de un *hanging node*

Cuando la *T-mesh* se construye con un único *hanging node* por arista se garantiza que cada celda comparte arista con otras celdas que son como mucho un nivel mayor o inferior, por tanto, se evitan las transiciones grandes y la malla queda suavizada.

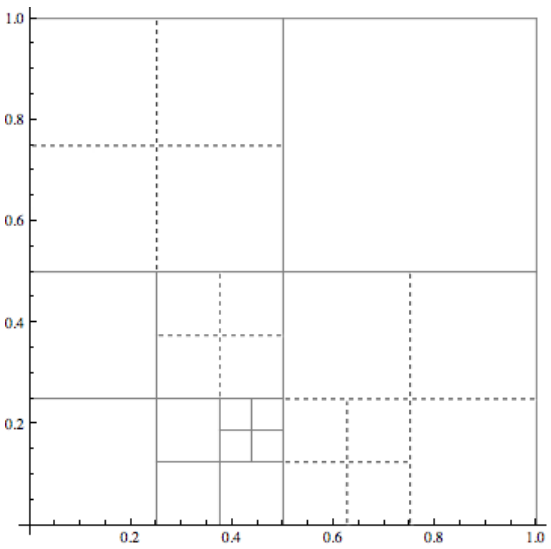


Equilibrado del *quadtree* para evitar varios *hanging node*

Un aspecto importante que hay que tener presente cuando se balancea el árbol es la propagación de *hanging nodes* a otras aristas donde no existía más de un *hanging node*.

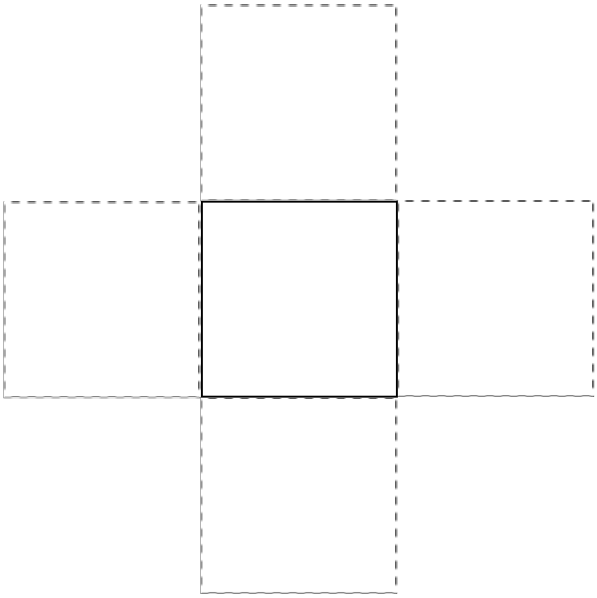


Propagación del *hanging node*



Quadtree balanceado

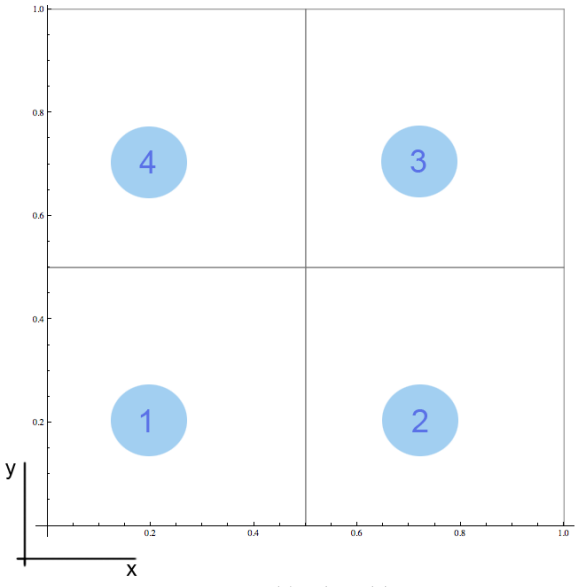
Para resolver el balanceo del *quadtree* es necesario definir una serie de propiedades y relaciones. Cada celda tiene una relación de vecindad con otras celdas de la *T-mesh*, de modo que se puede considerar vecinos de una celda aquellas celdas del mismo nivel de profundidad con las que se comparte una arista.



Celda con sus cuatro vecinos

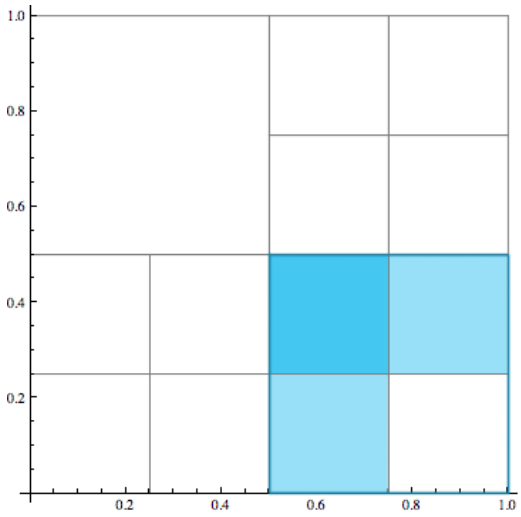
Propiedad 1: Dada una celda C_i siempre existen dos vecinos que comparten ancestros.

Los vecinos que comparten ancestros los podemos denominar *vecinos hermanos* de una celda. En función de la celda se puede determinar automáticamente sus dos *vecinos hermanos*.



Enumeración de celdas

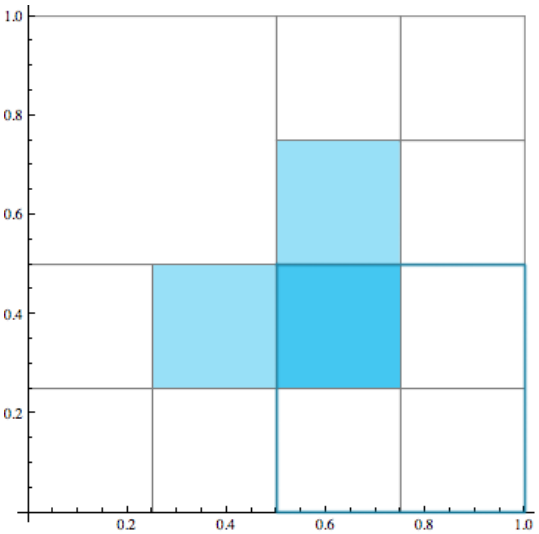
A partir de la secuencia numérica de la celda se obtiene automáticamente sus dos *vecinos hermanos*, uno que comparte arista en la dirección *x* y otro en la dirección *y*. Por ejemplo, si se trata de la celda número 4, en la dirección *x* siempre tendrá como *vecino hermano* a la celda 3 y en dirección *y* a la celda 1. De este modo, para cada una de las cuatro celdas se define cuál es el *vecino hermano* en cada dirección.



Celda {2,4} con sus dos *vecinos hermanos*

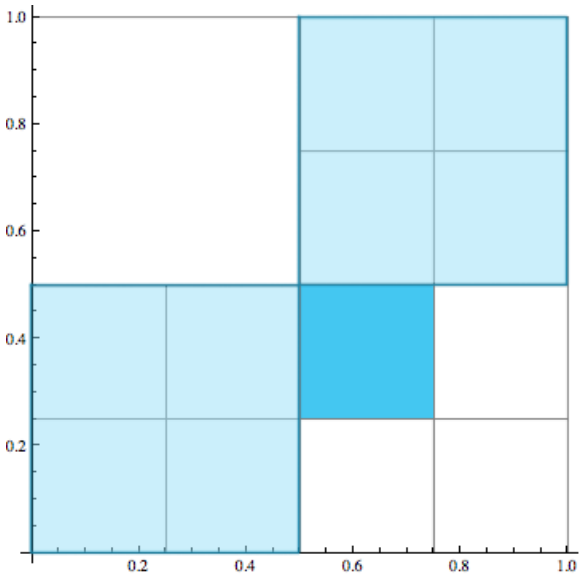
Propiedad 2: El árbol está equilibrado si para cada celda C_i existe las celdas padres de sus *vecinos no hermanos*.

Consideramos *vecinos no hermanos* de la celda C_i aquellas celdas del mismo nivel de C_i que comparten arista con ésta pero no comparten los mismos ancestros.



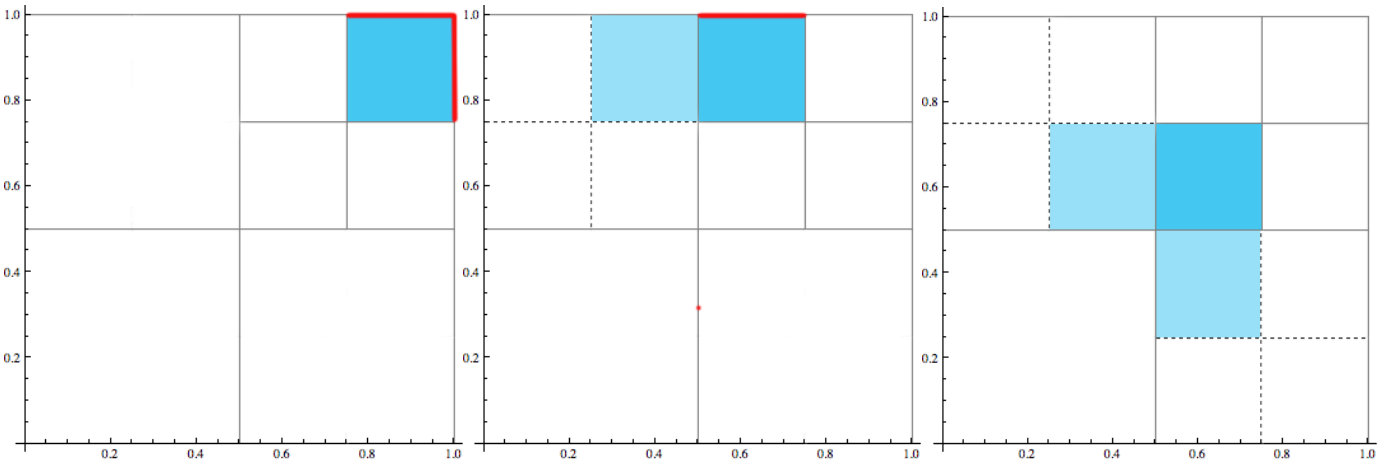
Celda {2,4} con sus dos *vecinos no hermanos*

Una vez detectados los *vecinos no hermanos* de la celda C_i, basta con comprobar si en el *quadtree* existen los nodos correspondientes a las celdas padre de esos *vecinos no hermanos*. En caso de no existir alguna de estas celdas padre, la celda C_i estaría en desequilibrio en el árbol.



Padres de *vecinos no hermanos*

El número de *vecinos hermanos* de toda celda siempre es dos. En cambio, el número de *vecinos no hermanos* de una celda puede variar desde cero hasta dos según la celda que sea. En concreto, las celdas que tienen dos de sus aristas en la frontera de la *T-mesh*, es decir, las cuatro celdas situadas en las esquinas de la *T-mesh*, carecen de *vecinos no hermanos*. Las celdas que tienen una arista en la frontera de la *T-mesh* tienen un único *vecino no hermano*, y las celdas sin aristas en la frontera de la *T-mesh*, es decir, las celdas internas, tienen dos *vecinos no hermanos*.

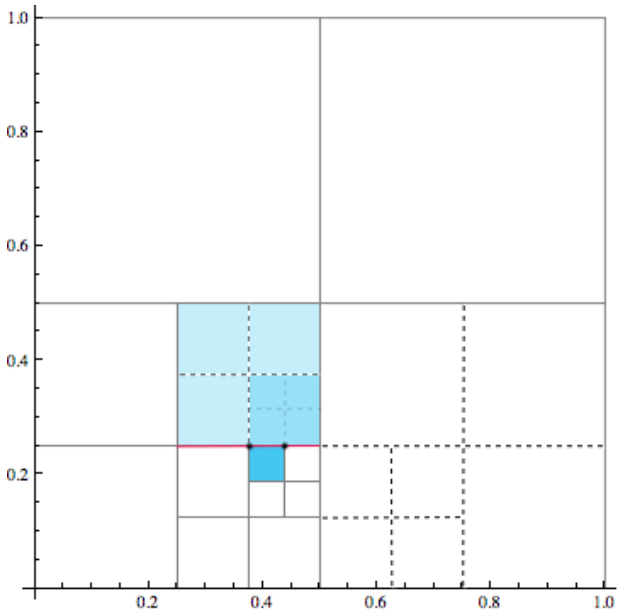


Dos aristas en la frontera | Una arista en la frontera | Sin aristas en la frontera

Definidas las propiedades anteriores y las relaciones entre celdas, el algoritmo de balanceo del *quadtree* consiste en un proceso por el cual para cada celda se comprueba si se cumple la propiedad 2, es decir, si existen las celdas padres de los *vecinos no hermanos*. Nótese que no es necesario que existan las celdas de los *vecinos no hermanos*, tan sólo deben existir sus celdas padres.

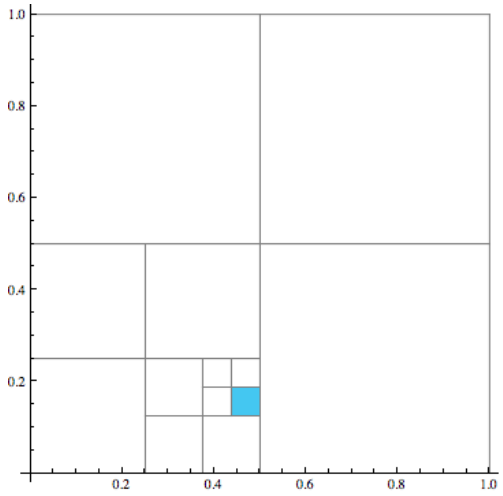
Por tanto, el proceso de balanceo obtiene, a partir de un *quadtree* desequilibrado, la secuencia numérica que identifica a cada una de las celdas; estas celdas se corresponden

con los nodos hojas del árbol. A partir de esta lista de secuencias, se procesará cada una para encontrar las secuencias de sus *vecinos no hermanos* y posteriormente comprobar la existencia de sus celdas padres en el *quadtree*. En caso de no existir alguna de estas celdas padres, se procederá a la inserción de los nodos necesarios en el árbol para que esa celda padre exista. Por último, para corregir la posible propagación de *hanging nodes*, se inserta al final de la lista de secuencias las secuencia correspondiente a la celda padre en cuestión y la de sus tres celdas hermanas.

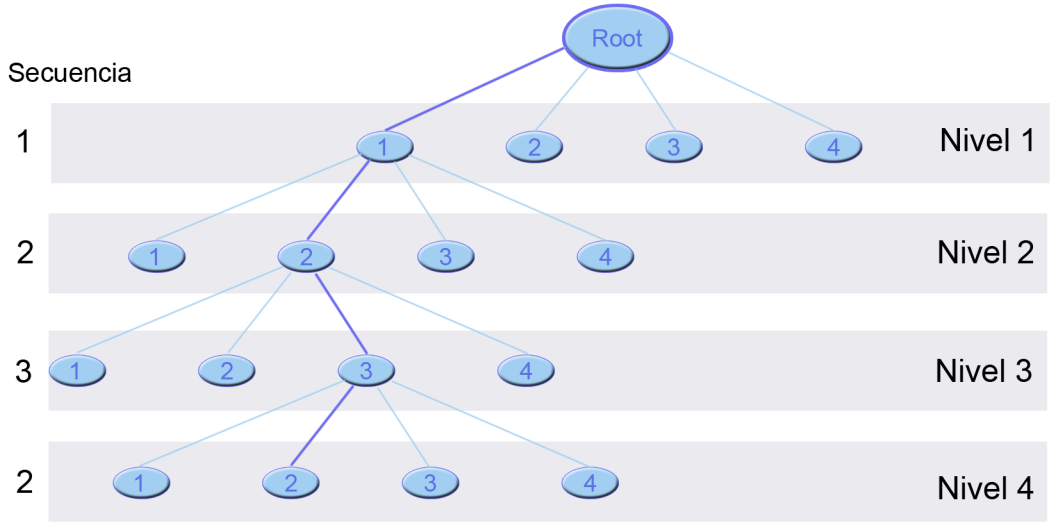


Celda padre del *vecino no hermano* y sus tres celdas hermanas

La secuencia numérica aporta información acerca de la celda. Por un lado identifica completamente a la celda y sus ancestros, de modo que marca los descensos que se necesitan realizar por el *quadtree* para obtener el nodo hoja correspondiente a la celda. La longitud de la secuencia indica el nivel del árbol en el que se encuentra ese nodo y , además, se puede determinar a partir de la secuencia las coordenadas de los vértices de la celda en el espacio paramétrico.

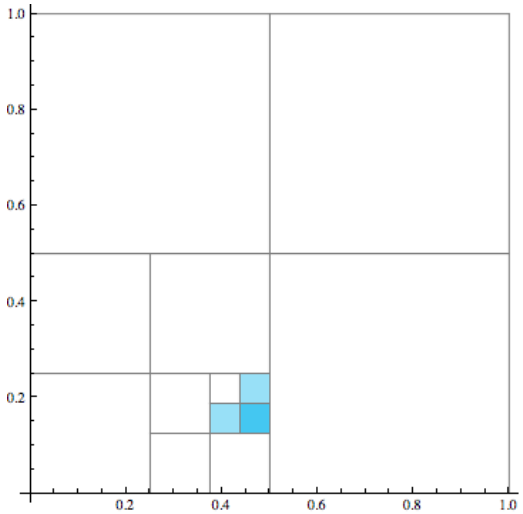


Celda con secuencia {1,2,3,2}



Descensos en el *quadtree* indicados por la secuencia

La información más importante que se puede obtener desde la secuencia numérica es posiblemente la correspondiente a los vecinos de la celda. Dada una secuencia numérica, automáticamente, en función del número de celda final, se obtiene las secuencias de sus *vecinos hermanos*, para los cuales la secuencia es idéntica ya que comparten ancestros y sólo cambia el último valor en función de las vecindades locales de la celda. Por ejemplo, en la secuencia anterior {1,2,3,2}, el último valor es 2, por lo que los *vecinos hermanos* son los que comparten arista en la dirección x e y con la celda número 2, es decir, las celdas 1 y 3. De este modo se determina que las secuencias de sus *vecinos hermanos* son {1,2,3,**1**} y {1,2,3,**3**}.



Vecinos hermanos de {1,2,3,2}. Secuencias {1,2,3,**1**} y {1,2,3,**3**}

Para balancear el *quadtree* se necesita conocer las secuencias de los *vecinos no hermanos* de cada celda. Una característica importante es que a través de la secuencia de la celda también es posible determinar automáticamente las secuencias de los *vecinos no hermanos*. Para lograrlo, se ha desarrollado en este trabajo un mecanismo basado en las relaciones de vecindades de las celdas.

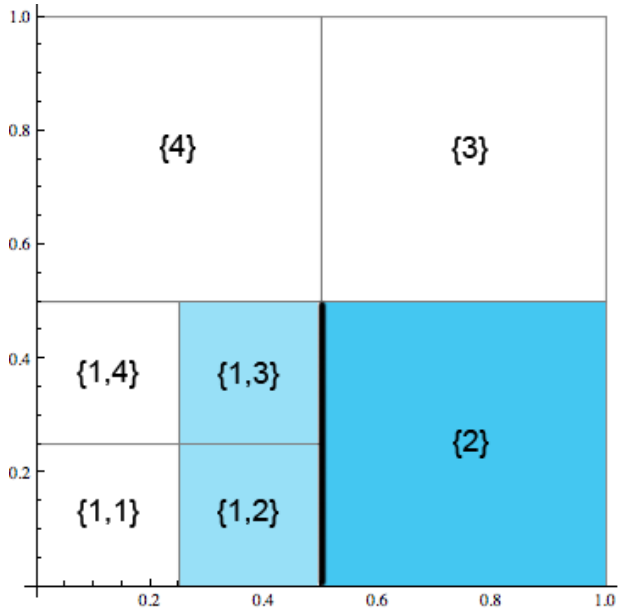
En primer lugar se define una tabla que indica para cada celda cuáles son sus relaciones de vecindad en las direcciones x e y en función del número de celda. A esta tabla se le ha denominado *tabla de vecindades*.

Celda	x	y
1	2	4
2	1	3
3	4	2
4	3	1

Tabla de vecindades

La *tabla de vecindades* se puede leer como "la celda A comparte arista en la dirección x con la celda B y en la dirección y con la celda C". Por ejemplo, la celda 2 comparte arista en la dirección x con la celda 1 y en la dirección y con la celda 3.

En segundo lugar se necesita definir otra tabla que indique los posibles descensos que se pueden realizar por las celdas hijas de una celda vecina, de modo que se continúe compartiendo arista con las celdas por las que se desciende. Por ejemplo, la celda 2 tiene como vecino a la celda 1 en la dirección x, pero las celdas hijas de 1 que comparten arista con 2 son sólo la 2 y 3, por tanto las celdas descendientes de 1 por las que se puede avanzar para continuar compartiendo arista con 2 serán siempre los descendientes 2 o 3.



Restricciones de descenso por la celda 1 respecto a la celda 2

La tabla que establece estas restricciones de descenso se le ha llamado *tabla de restricciones*.

Celda	1	2	3	4
1	--	1, 4	--	1, 2
2	2, 3	--	1, 2	--
3	--	3, 4	--	2, 3

4	3, 4	--	1, 4	--

Tabla de restricciones

La lectura en esta tabla puede ser "para la celda A, los hijos de la celda vecina B por los que se puede descender son …". Por ejemplo, para la celda 1, los hijos de la celda 2 por los que se puede descender son 1 y 4.

En base a la información de estas dos tablas se puede definir un algoritmo para encontrar las secuencias de los *vecinos no hermanos* a partir de la secuencia de la celda. El algoritmo consiste en completar una tabla con la siguiente información:

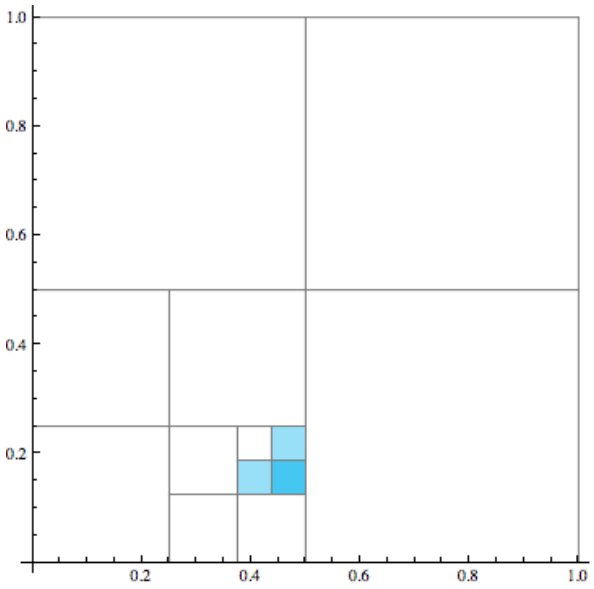
- La primera columna (secuencia) son cada uno de los números de la secuencia de la celda.
- La segunda columna (vecino en *x*) es el número de la celda hermana que comparte arista en la dirección *x* con la celda de la primera columna. Esta columna se rellena con la información de la *tabla de vecindades*.
- La tercera columna (vecino en *y*) es el número de la celda hermana que comparte arista en la dirección *y* con la celda de la primera columna. Esta columna se rellena con la información de la *tabla de vecindades*.
- La cuarta columna (restricciones en *x*) son las celdas por las que se puede descender según la relación de la primera columna con la segunda. Esta columna se rellena con la información de la *tabla de restricciones*.
- La quinta columna (restricciones en *y*) son las celdas por las que se puede descender según la relación de la primera columna con la tercera. Esta columna se rellena con la información de la *tabla de restricciones*.

En el ejemplo de la secuencia {1,2,3,2} esta tabla quedaría de la siguiente forma:

Secuencia	Vecino en *x*	Vecino en *y*	Restr. en *x*	Restr. en *y*
1	2	4	1, 4	1, 2
2	1	3	2, 3	1, 2
3	4	2	2, 3	3, 4
2	1	3		

La última fila de las restricciones, tanto en *x* como en *y*, no es necesario que se añadan. Una vez se ha completado la tabla, el siguiente paso del algoritmo es obtener todas las posibles secuencias candidatas entre las cuales se encontrarán las secuencias de los *vecinos no hermanos* de la celda. Los candidatos se generan manteniendo fija una parte de la secuencia inicial y sustituyendo el resto por los valores de las columnas vecino en *x* y vecino en *y*.

Por ejemplo, los primeros candidatos podrían surgir de mantener fijo toda la secuencia menos en último valor {1,2,3,_} y generar el candidato {1,2,3,**1**} con el valor del vecino en *x* y el candidato {1,2,3,**3**} con el valor del vecino en *y*. Estos dos candidatos que surgen de mantener constante toda la secuencia menos el último valor no es necesario que se realice, ya que estos dos candidatos son precisamente los *vecinos hermanos* de la celda, y por tanto existen con total seguridad.

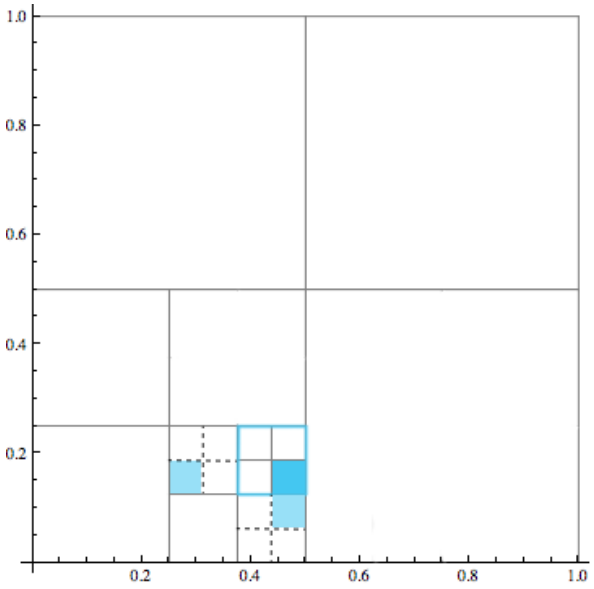


Secuencias candidatas {1,2,3,**1**} y {1,2,3,**3**} correspondientes a *vecinos hermanos*

Las siguientes secuencias candidatas, que son realmente los primeros candidatos a *vecinos no hermanos*, se generan variando los dos últimos valores de la secuencia original. Es decir, en este ejemplo se generarían las secuencias {1,2,**4,1**} y {1,2,**2,3**}.

Secuencia	Vecino en x	Vecino en y
1	2	4
2	1	3
3	4	2
2	1	3

Si se representa gráficamente, se puede ver cómo la secuencia {1,2,**4,1**} no se corresponde con un *vecino no hermano* pero la secuencia {1,2,**2,3**} sí lo es.



Secuencias candidatas {1,2,**4,1**} y {1,2,**2,3**}

El algoritmo, para determinar si una secuencia es válida, utiliza los valores de las columnas de restricciones. El primer valor sustituido en la secuencia marca las restricciones de descenso. En la secuencia candidata {1,2,**4,1**} el primer valor modificado es el 4, el cual indica que se está buscando el candidato en el vecino 4 de la celda original 3, ya que la secuencia original es {1,2,**3**,2}. Por tanto, según la *tabla de restricciones*, los descensos válidos por la celda 4 respecto a la celda 3 serían las celdas 2 y 3, ya que son las que siguen compartiendo arista con la celda original.

Secuencia	Vecino en x	Restr. en x
1	2	1, 4
2	1	2, 3
3	4	2, 3
2	1	

Comprobación del candidato según vecinos en x

Para que el candidato sea válido, todos los descensos que se realicen por el vecino 4 deben realizarse por las celdas hijas 2 y 3. En el momento que un descenso no se realice por una de esas celdas, se dejará de compartir arista con la celda original y el candidato no será válido. Por tanto, en el candidato {1,2,**4,1**}, como el primer descenso por el vecino {1,2,**4**} se realiza por la celda 1, el candidato no es válido.

Se procede de igual forma para el candidato según vecinos en la dirección y.

Secuencia	Vecino en y	Restr. en y
1	4	1, 2
2	3	1, 2
3	2	3, 4
2	3	

Comprobación de candidato según vecinos en y

En este caso, las restricciones son las celdas 3 y 4, y todos los descensos a partir del vecino {1,2,**2**} se realizan por hijos 3 o 4, por lo que la secuencia candidata es válida.

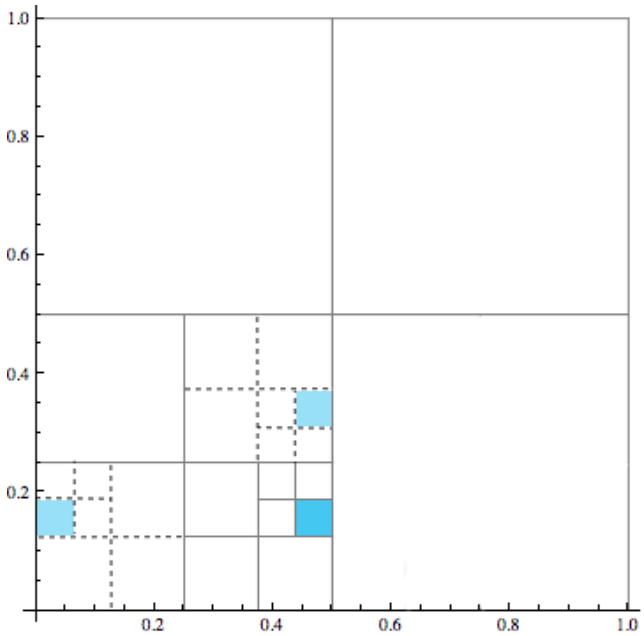
Los siguientes candidatos se obtienen variando los 3 últimos valores de la secuencia, y de esta forma se van generando todos los candidatos hasta haber obtenido dos válidos, ya que una celda sólo puede tener como mucho dos *vecinos no hermanos*, o hasta haber comprobado todos los candidatos.

Secuencia	Vecino en x	Restr. en x
1	2	1, 4
2	1	2, 3
3	4	2, 3
2	1	

Candidato {1,**1,4,1**} no válido

Secuencia	Vecino en y	Restr. en y
1	4	1, 2
2	3	1, 2
3	2	3, 4
2	3	

Candidato {1,**3,2,3**} no válido



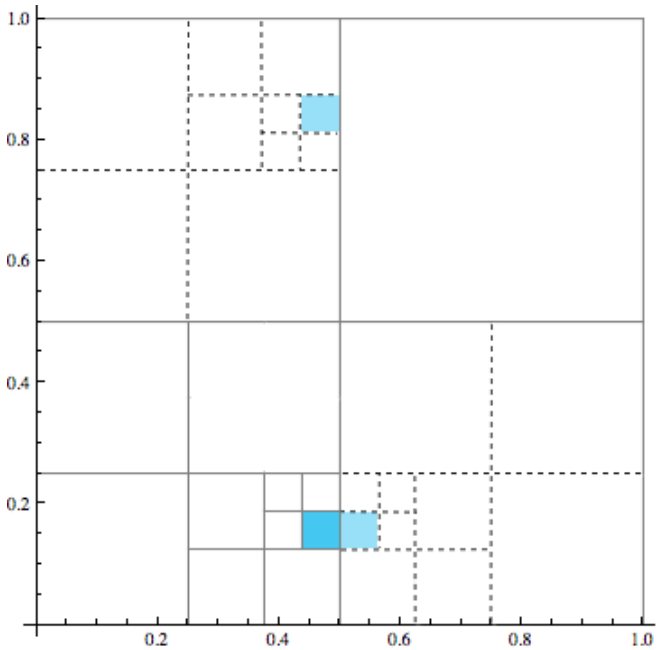
Secuencias candidatas {1,**1,4,1**} y {1,**3,2,3**}

Secuencia	Vecino en x	Restr. en x
1	2	1, 4
2	1	2, 3
3	4	2, 3
2	1	

Candidato {**2,1,4,1**} válido

Secuencia	Vecino en y	Restr. en y
1	4	1, 2
2	3	1, 2
3	2	3, 4
2	3	

Candidato{**4,3,2,3**} no válido



Secuencias candidatas {**2,1,4,1**} y {**4,3,2,3**}

El número de candidatos máximo de cada celda es proporcional a la longitud de la secuencia. Sea *l* la longitud de la secuencia, el número de candidatos máximo que hay que analizar para encontrar los *vecinos no hermanos* de la celda es $2(l - 1)$.

Una vez se dispone de un método para la obtención de los *vecinos no hermanos* a partir de la secuencia de una celda, se puede realizar de forma sencilla un algoritmo para el balanceo del *quadtree*. Para ello habrá que recorrer cada celda comprobando que existan las celdas padres de los *vecinos no hermanos*, y en caso de no existir, insertarlas en el *quadtree*. A continuación se plantea en pseudocódigo el algoritmo para el balanceo de un *quadtree*.

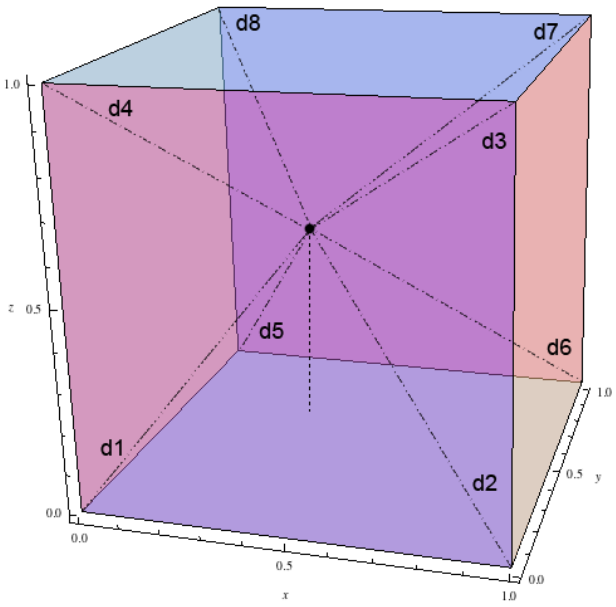
```
secuencias = obtener_secuencias_de_celdas(quadtree);

for secuencia in secuencias do
     vecinos = obtener_secuencias_de_vecinos(secuencia);
     for vecino in vecinos do
          if not existe?(quadtree, padre(vecino)) {
               insertar(quadtree, padre(vecino));
               nuevos = [padre(vecino)] + hermanos(padre(vecino));
               secuencias = secuencias + nuevos;
          }
     end
end
```

Algoritmo de balanceo de un *quadtree*

4.3.- Extensión a 3D. Construcción y balanceo del octree

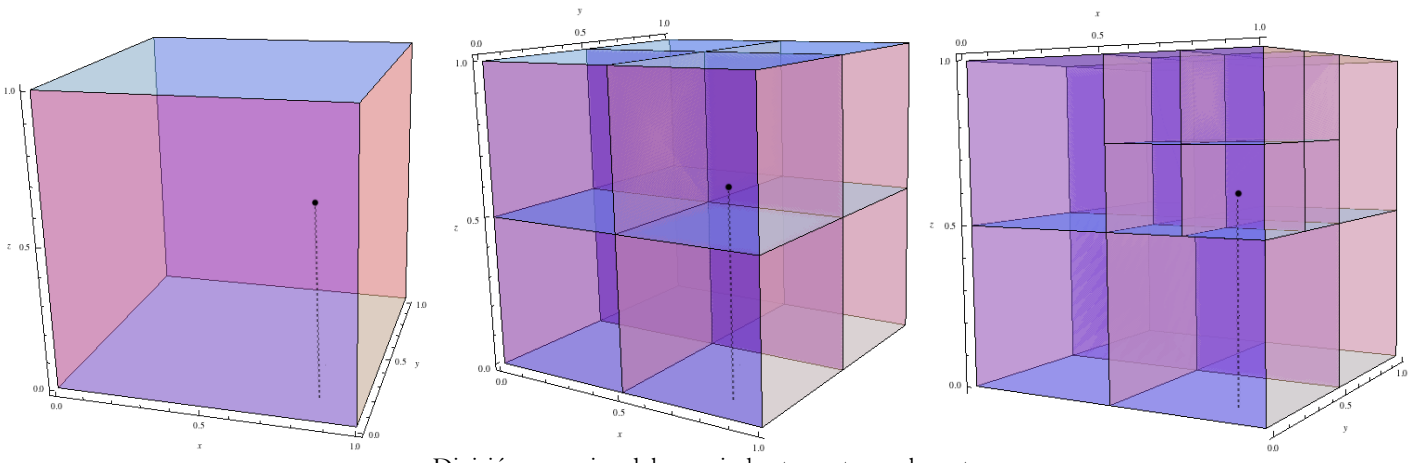
Tanto la construcción como el balanceo de la estructura *octree* es análoga a la estructura *quadtree*.

El proceso de construcción es exactamente el mismo, con la única diferencia de que en el árbol cada nodo pasa a tener 8 hijos en lugar de 4, un nodo por cada octante en el que se subdivide la celda. Un punto de la discretización del dominio se considera que está contenido por una celda si su distancia respecto a cada vértice del cubo que lo contiene es menor a un valor *épsilon* indicado. Además, la discretización de entrada se trata de una lista compuesta por puntos en el espacio con sus coordenadas *x*, *y* y *z*.

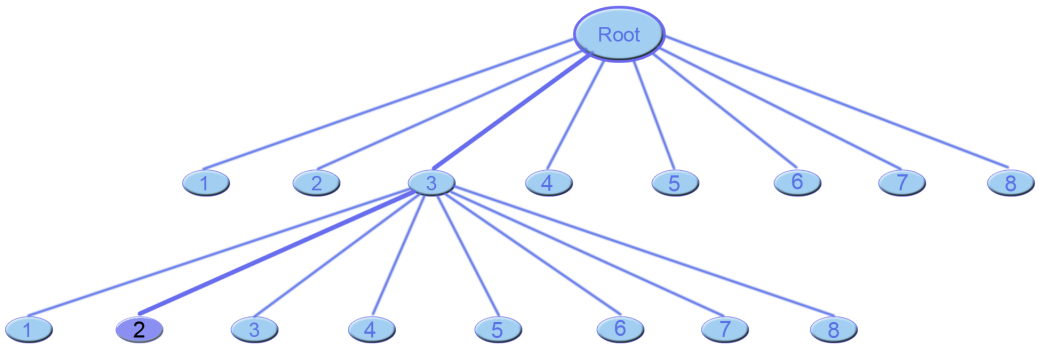


Distancias a cada vértice de la celda del *octree*

La construcción del *octree* se realiza de forma recursiva insertando punto a punto en el árbol hasta garantizar que cada punto está capturado por una celda del *octree* mediante la condición de cercanía a cada vértice.

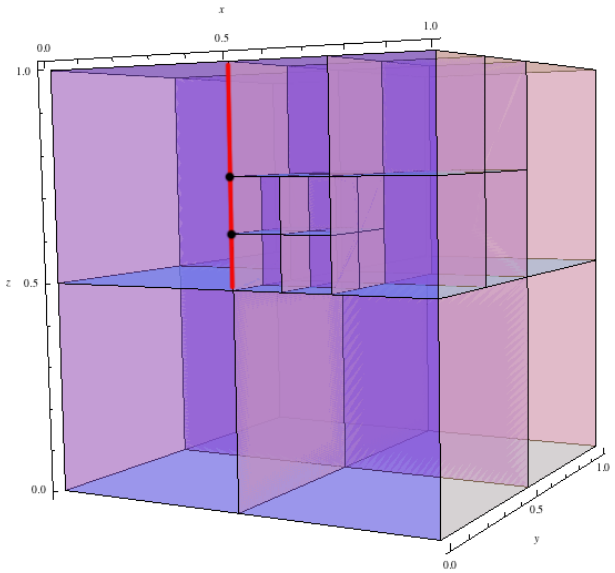


División recursiva del espacio hasta capturar el punto



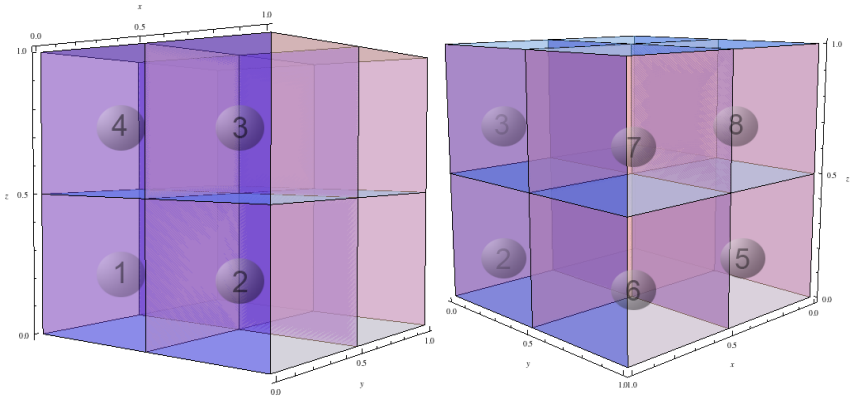
Octree tras insertar el primer punto

En el *octree* resultante puede ocurrir que hayan transiciones de celdas muy grandes a celdas mucho menores. Al igual que el *quadtree*, para evitar estas transiciones bruscas es necesario equilibrar el árbol, permitiendo solamente un *hanging node* por arista.



Octree con más de un *hanging node* por arista

El algoritmo para el balanceo del *quadtree* se puede extender de forma sencilla al *octree*, sólo es necesario redefinir las vecindades entre celdas y las restricciones de descenso por las celdas hijas de una celda vecina. En el caso del *octree*, las vecindades se establecen por caras del hexaedro en lugar de por aristas. De este modo, dos celdas son vecinas si comparten una cara.



Enumeración de celdas del *octree*

Cada celda tiene tres *vecinos hermanos*, uno en cada dirección x, y y z. Por ejemplo, la celda número 1 tiene como *vecino hermano* a la celda número 2 en la dirección x, la celda número 5 en la dirección y y la celda número 4 en la dirección z. En cuanto a los *vecinos no hermanos*, cada celda podrá tener desde cero hasta tres *vecinos no hermanos*, dependiendo del número de caras que tenga la celda sobre las caras del cubo paramétrico. De este modo, las celdas con tres caras sobre las caras del cubo paramétrico, es decir, las ocho celdas de las esquinas, no tendrán *vecinos no hermanos*. Las celdas con dos caras sobre las del cubo paramétrico tendrán un sólo *vecino no hermano*, las celdas con una cara sobre las del cubo paramétrico tendrán 2 *vecinos no hermanos*, y por último, las celdas que no tengan caras sobre las del cubo paramétrico, es decir, las celdas totalmente internas, tendrán 3 *vecinos no hermanos*.

Con estas condiciones de vecindad se puede redefinir las tablas de vecindades y restricciones, manteniendo prácticamente invariable el resto del algoritmo de balanceo del *quadtree*.

Celda	x	y	z
1	2	5	4
2	1	6	3
3	4	7	2
4	3	8	1
5	6	1	8
6	5	2	7
7	8	3	6
8	7	4	5

Tabla de vecindades

Celda	1	2	3	4	5	6	7	8
1	--	1,4,5,8	--	1,2,5,6	1,2,3,4	--	--	--
2	2,3,6,7	--	1,2,5,6	--	--	1,2,3,4	--	--
3	--	3,4,7,8	--	2,3,6,7	--	--	1,2,3,4	--
4	3,4,7,8	--	1,4,5,8	--	--	--	--	1,2,3,4
5	5,6,7,8	--	--	--	--	1,4,5,8	--	1,2,5,6
6	--	5,6,7,8	--	--	2,3,6,7	--	1,2,5,6	--
7	--	--	5,6,7,8	--	--	3,4,7,8	--	2,3,6,7

8	--	--	--	5,6,7,8	3,4,7,8	--	1,4,5,8	--

Tabla de restricciones

Para la localización de las secuencias de los *vecinos no hermanos* en 3D se emplea la misma tabla que en 2D añadiendo la información de las vecindades y restricciones en la dirección z:

Secuencia	Vecino en x	Vecino en y	Vecino en z	Restr. en x	Restr. en y	Restr. en z
--	--	--	--	--	--	--

Tabla para el cálculo de secuencias de *vecinos no hermanos* en 3D

El número de candidatos máximo de cada celda es proporcional a la longitud de la secuencia. Sea l la longitud de la secuencia, el número de candidatos máximo que hay que analizar para encontrar los *vecinos no hermanos* de la celda es $3(l - 1)$.

Tras la adaptación de la localización de *vecinos no hermanos* al caso 3D, el resto del algoritmo de balanceo queda invariante respecto a la versión 2D.

```
secuencias = obtener_secuencias_de_celdas(octree);

for secuencia in secuencias do
     vecinos = obtener_secuencias_de_vecinos(secuencia);
     for vecino in vecinos do
          if not existe?(octree, padre(vecino)) {
               insertar(octree, padre(vecino));
               nuevos = [padre(vecino)] + hermanos(padre(vecino));
               secuencias = secuencias + nuevos;
          }
     end
end
```

Algoritmo de balanceo de un *octree*

5.- Resultados

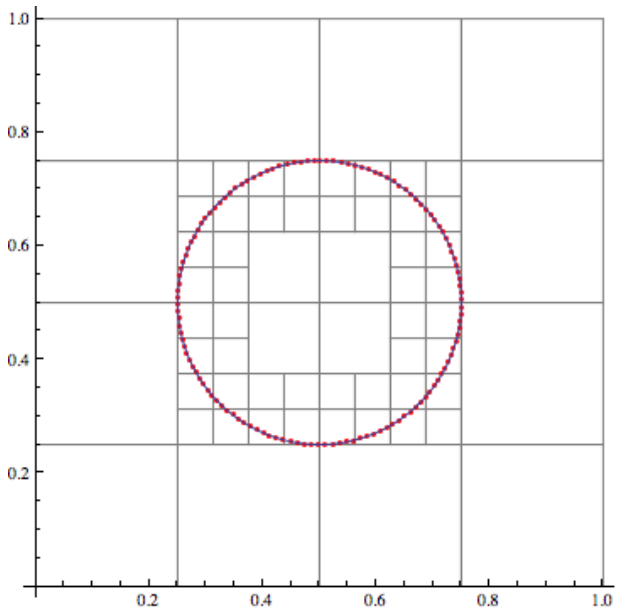
Los prototipos diseñados, tanto para la construcción como para el balanceo del árbol, se han implementado como módulos en lenguaje *Mathematica*. La utilización de *Mathematica* en este proyecto de máster se debe a varias razones. Por un lado *Mathematica* es un lenguaje de alto nivel que permite diseñar de forma rápida un prototipo. Además, disponer de módulos *Mathematica* para el balanceo *quadtree/octree* resultaba interesante para poder ser utilizados directamente en el código *Mathematica* para análisis isogeométrico desarrollado en la división de Discretización y aplicaciones.

A continuación se muestra una serie de resultados obtenidos primero para la versión 2D y posteriormente en 3D.

5.1 Versión 2D

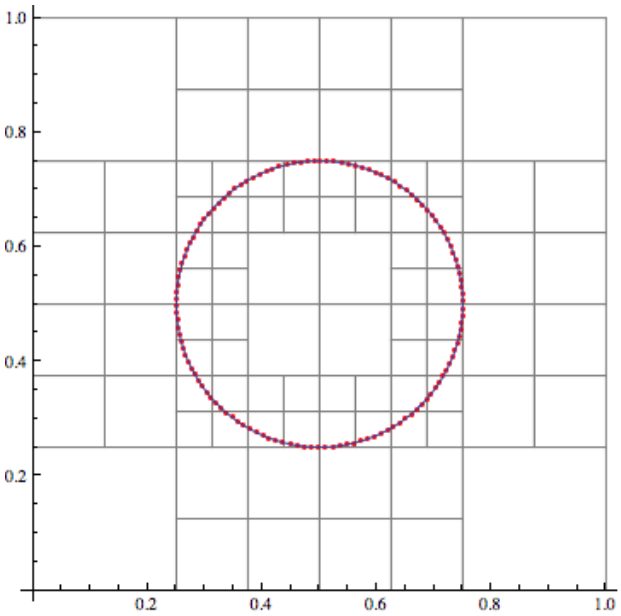
Ejemplo 1

Este primer ejemplo consiste en la construcción de un *quadtree* a partir de la discretización de una circunferencia. A partir de una serie de puntos que caen sobre la circunferencia, se construye el *quadtree* de modo que todos los puntos queden capturados en una celda de la *T-mesh* por la condición de cercanía a cada uno de los vértices de la celda.



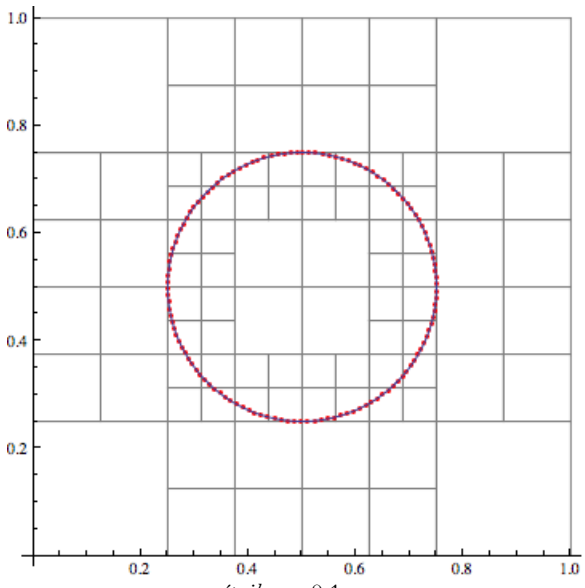
Quadtree que aproxima la circunferencia

El *quadtree* resultante no está balanceado. Tras aplicar el algoritmo de balanceo se obtiene la siguiente *T-mesh*.

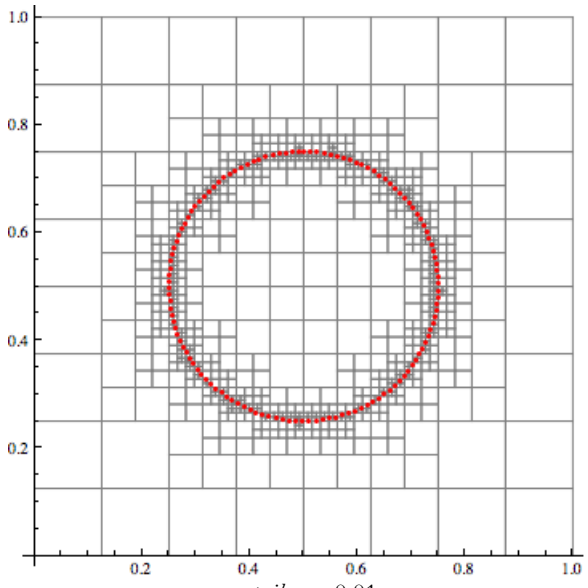


Quadtree balanceado que aproxima la circunferencia

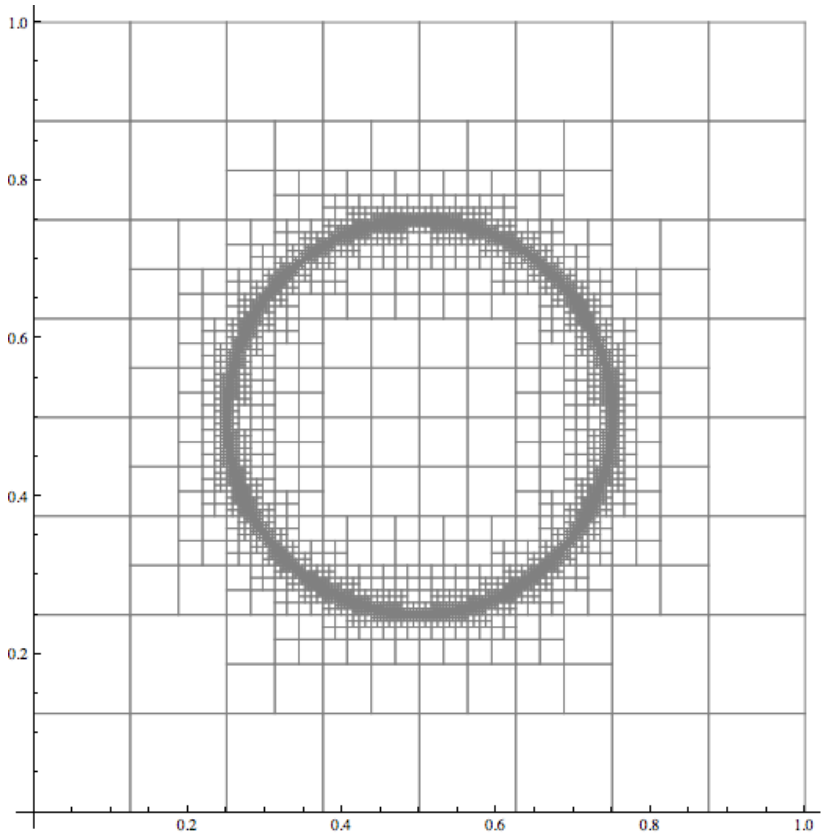
Como se observa, la malla generada es bastante grosera, es decir, el tamaño de celda es muy grande, por lo que no se aproxima la circunferencia con demasiada exactitud. Esto se corrige con un mayor refinamiento de la *T-mesh*, lo que se traduce en utilizar un valor de *épsilon* menor de modo que las celdas contenedoras de un punto sean más pequeñas. En el ejemplo anterior se empleó un valor de *épsilon* igual a 0.1. En las siguientes figuras se muestra cómo se produce el refinamiento de la malla al disminuir el valor de *épsilon*.



épsilon = 0.1



epsilon = 0.01

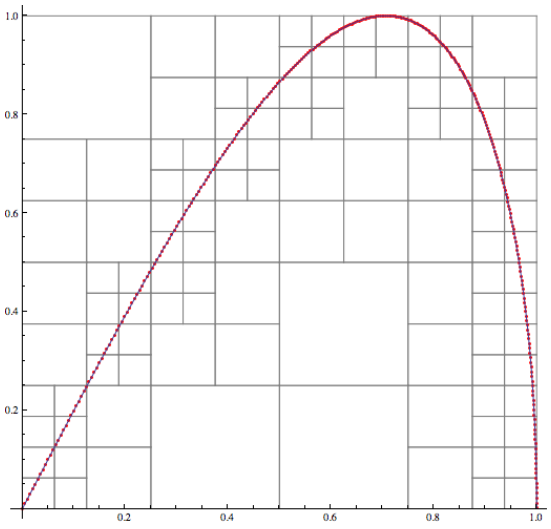


épsilon = 0.001

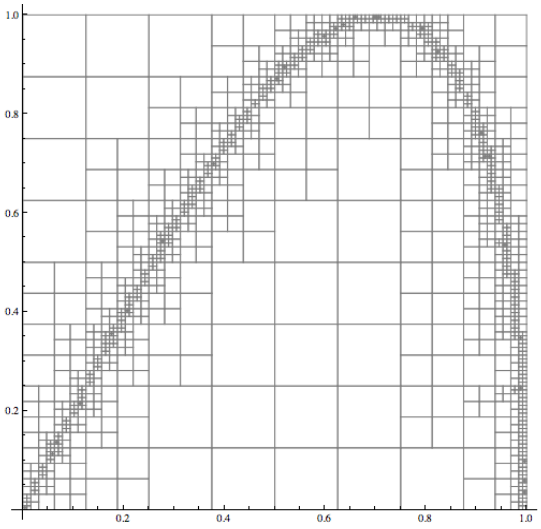
Con un valor de *épsilon* lo suficientemente pequeño se consigue una aproximación muy cercana a la circunferencia real.

Otros ejemplos que se han obtenido en la versión 2D son la aproximación de una parábola o una espiral.

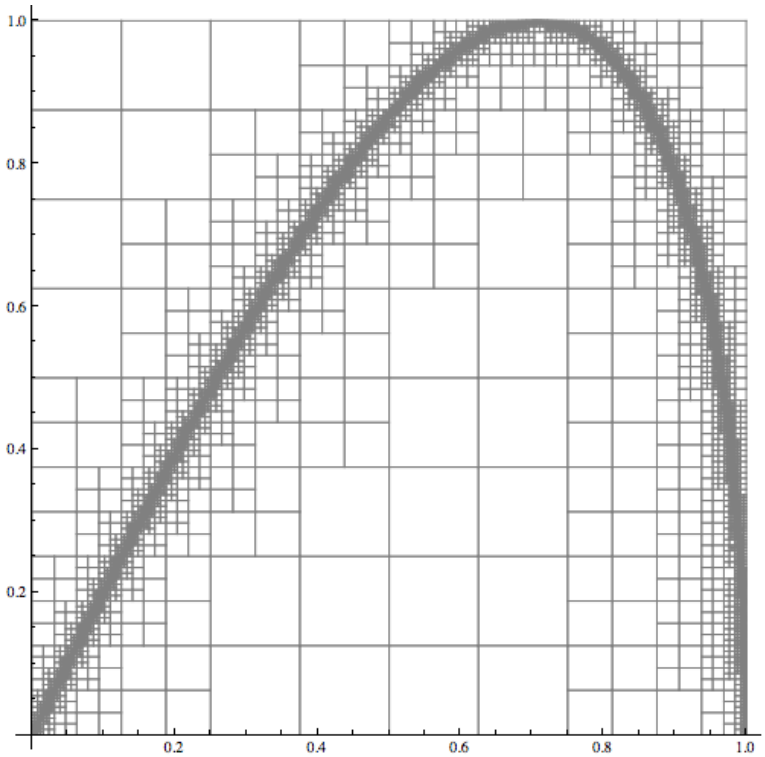
Ejemplo 2



Aproximación de parábola con *épsilon* = 0.1

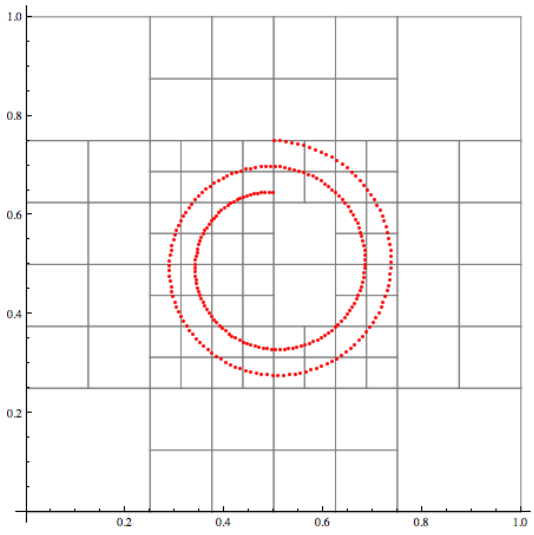


Aproximación de parábola con *épsilon* = 0.01

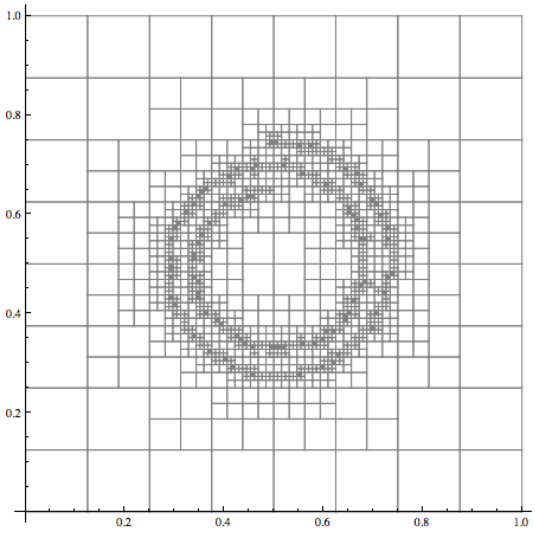


Aproximación de parábola con *épsilon* = 0.001

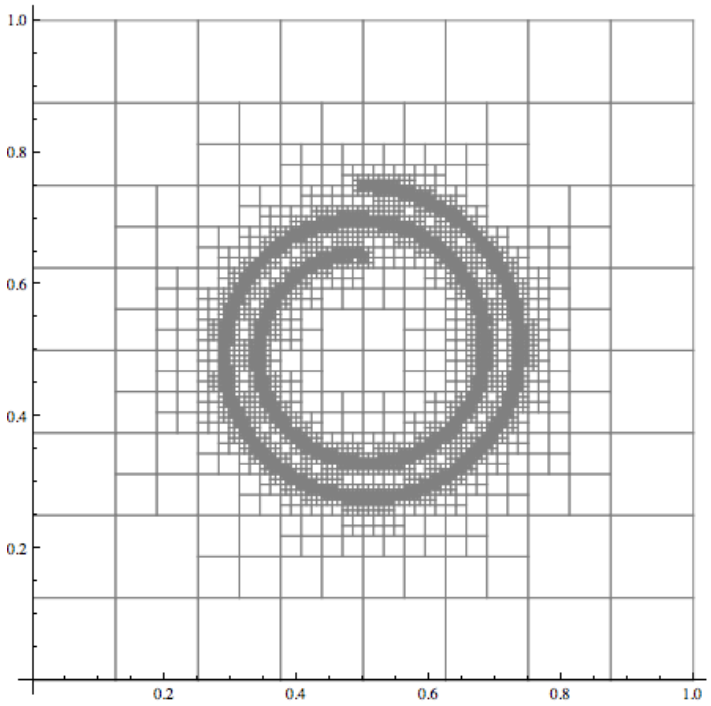
Ejemplo 3



Aproximación de espiral con *épsilon* = 0.1



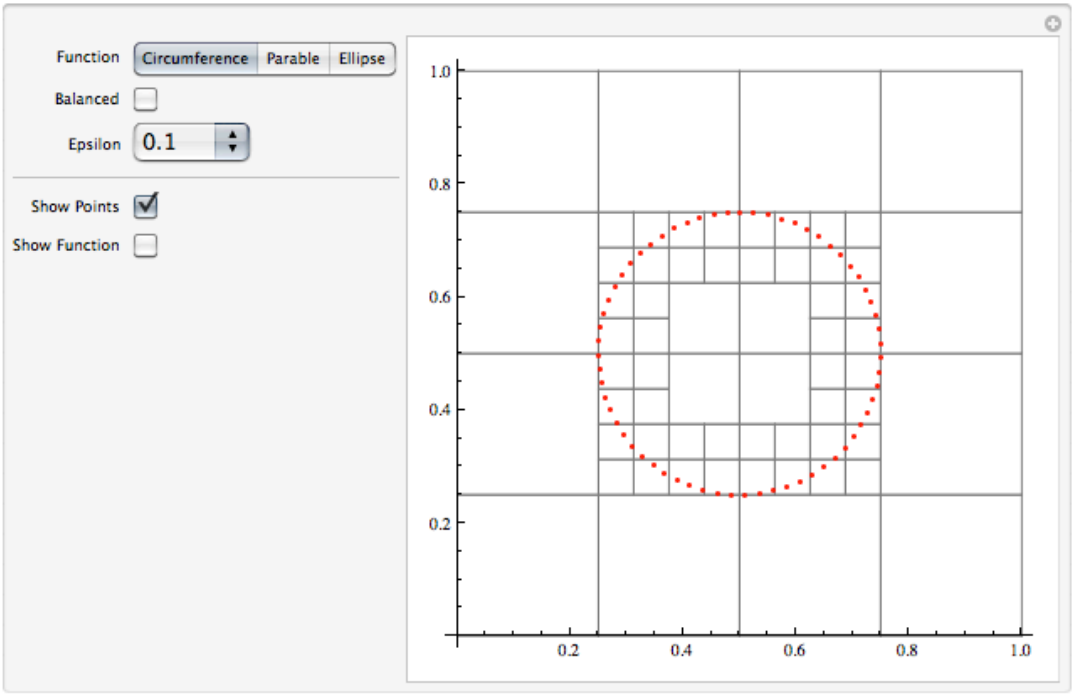
Aproximación de espiral con *épsilon* = 0.01



Aproximación de espiral con *épsilon* = 0.001

Demo Test

Para poder realizar pruebas se ha diseñado un módulo *Mathematica* de test con interfaz gráfica, donde se permite elegir las funciones circunferencia, parábola y espiral para construir una *T-mesh* que se aproxime a dichas funciones, permitiendo obtener la versión balanceada del *quadtree*.

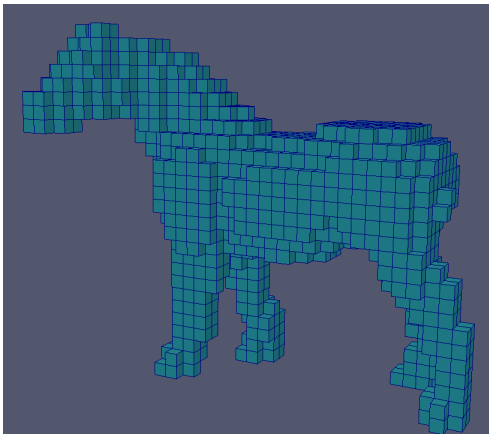


Demo de test

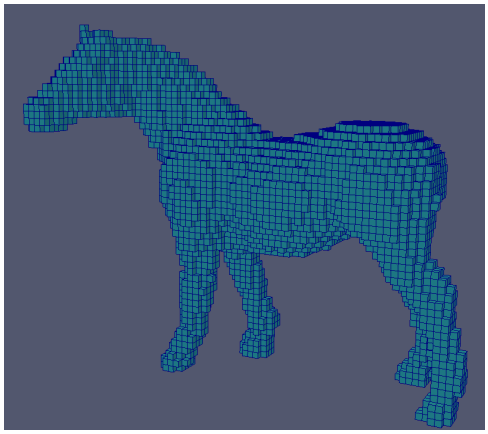
5.2.- Versión 3D

Para poder visualizar los resultados de la aproximación mediante *octree*, se ha representado a través del programa *Paraview* sólo aquellas celdas del *octree* que contienen al menos un punto de la discretización inicial del objeto.

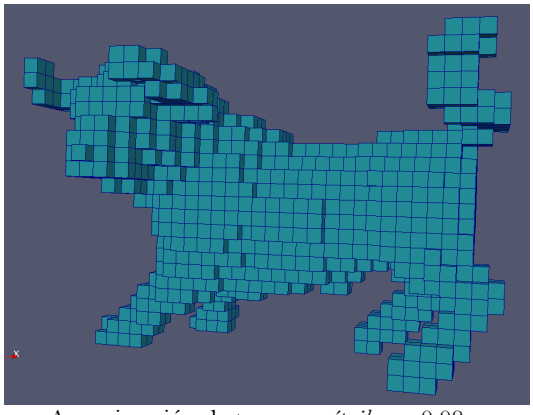
A partir de una triangulación de entrada en un fichero *inp*, se leen los puntos de la triangulación y se construye el *octree* insertando cada punto en el árbol. A continuación se muestran los resultados para la aproximación de distintos objetos.



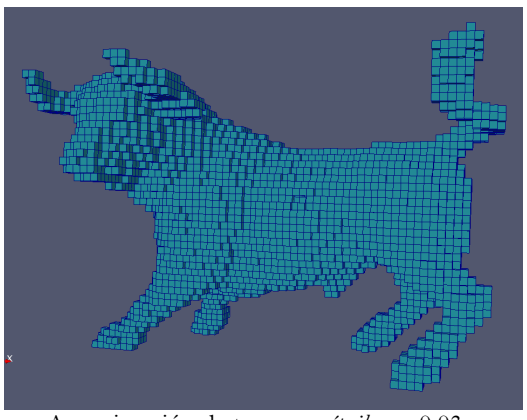
Aproximación de caballo con *épsilon* = 0.08



Aproximación de caballo con *épsilon* = 0.03



Aproximación de toro con *épsilon* = 0.08



Aproximación de toro con *épsilon* = 0.03

6.- Conclusiones y líneas futuras

En este trabajo se ha implementado un prototipo para la construcción automática y balanceo de estructuras *quadtree* y *octree* que aproximan sólidos para el análisis isogeomético con T-splines. El código construido ofrece un comportamiento robusto y con un bajo coste computacional a pesar de estar implementado en un lenguaje interpretado como *Mathematica*. Los módulos desarrollados se han incluido en el código de análisis isogeométrico de la división de Discretización y Aplicaciones, permitiendo aplicar el método con *T-mesh* balanceadas, que era una característica importante de la cual carecía hasta ahora.

El siguiente paso en esta línea sería la implementación del prototipo en código C o C++ para obtener un software eficiente. Con este trabajo se pretende en líneas futuras la utilización de estructuras *octree* para la automatización del método del *meccano* [10].

Referencias

[1] J.A. Cottrell, T.J.R. Hughes, Y. Bazilevs, Isogeometric Analysis: Toward Integration of CAD and FEA, John Wiley & Sons, Chichester, 2009.

[2] Y. Bazilevs, V.M. Calo, J.A. Cottrell, J.A. Evans, T.J.R. Hughes, S. Lipton, M.A. Scott, T.W. Sederberg, Isogeometric analysis: toward unification of computer aided design and finite element analysis, in: Trends in Engineering Computational Technology, Saxe-Coburg Publications, Stirling, 2008, pp. 1–16.

[3] Y. Bazilevs, V.M. Calo, J.A. Cottrell, J.A. Evans, T.J.R. Hughes, S. Lipton, M.A. Scott, T.W. Sederberg, Isogeometric analysis using T-splines, Comput. Methods Appl. Mech. Engrg. 199 (2010) 229–263.

[4] T.W. Sederberg, J. Zheng, A. Bakenov, A. Nasri, T-splines and T-NURCCSs, ACM Trans. Graph. 22 (3) (2003) 477–484.

[5] T. Martin, E. Cohen, R.M. Kirby, Volumetric parameterization and trivariate Bspline fitting using harmonic functions, Comput. Aid. Geom. Des. 26 (2009) 648–664.

[6] T. Martin, E. Cohen, Volumetric parameterization of complex objects by respecting multiple materials, Comput. Graph. 34 (2010) 187–197.

[7] J.M. Escobar, J.M. Cascón, E. Rodríguez, R. Montenegro, A new approach to solid modeling with trivariate T-splines based on mesh optimization, Comput. Methods Appl. Mech. Engrg. 200 (2011) 3210–3222.

[8] J.M. Escobar, E. Rodríguez, R. Montenegro, G. Montero, J.M. González-Yuste, Simultaneous untangling and smoothing of tetrahedral meshes, Comput. Methods Appl. Mech. Engrg. 192 (2003) 2775–2787.

[9] R. Montenegro, J.M. Cascón, J.M. Escobar, E. Rodríguez, G. Montero, An automatic strategy for adaptive tetrahedral mesh generation, Appl. Numer. Math. 59 (2009) 2203–2217.

[10] R. Montenegro, J.M. Cascón, E. Rodríguez, J.M. Escobar, G. Montero, The meccano method for automatic 3-D triangulation and volume parametrization of complex solids, Comput. Sci. Engrg. Technol. Ser. 26 (2010) 19–48.

[11] Hanan Samet, Foundations of Multidimensional And Metric Data Structures, 2006.