Trabajo de Fin de Grado
Grado en Ingeniería Electrónica, Robótica y Mecatrónica

# Entorno interactivo MATLAB para análisis de texturas en imágenes III

Autor: Javier Mérida Floriano
Tutor: Manuel Ruiz Arahal

**Dpto. Ingeniería de Sistemas y Automática**
**Escuela Técnica Superior de Ingeniería**
**Universidad de Sevilla**

Sevilla, 2020

Trabajo de Fin de Grado
Grado en Ingeniería Electrónica, Robótica y Mecatrónica

# Entorno interactivo MATLAB para análisis de texturas en imágenes III

Autor:
Javier Mérida Floriano

Tutor:
Manuel Ruiz Arahal
Catedrático de Universidad

Dpto. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2020

Trabajo de Fin de Grado: Entorno interactivo MATLAB para análisis de texturas en imágenes III

Autor: Javier Mérida Floriano

Tutor: Manuel Ruiz Arahal

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2020

El Secretario del Tribunal

*A mi familia*

*A mis amigos/as*

*A mis profesores/as*

# Agradecimientos

No tengo otras palabras más que de agradecimiento a todas aquellas personas que han estado a mi lado durante estos años de mi vida, ya que, gracias a ellos, los que siguen y los que ya no están, he sido capaz de seguir hacia delante en mi objetivo de ser quien quiero ser.

Agradecimientos individuales a mi tutor, Manuel Ruiz Arahal, por la paciencia mostrada conmigo a la hora de realizar este proyecto; a mis padres y hermanas, por preocuparse y motivarme para poder completar el último empujón; y a mi pareja, por estar junto a mí y apoyarme en los momentos más duros.

*Javier Mérida Floriano*

*Sevilla, 2020*

# Resumen

Este proyecto trata de facilitar, de forma interactiva y visual, el análisis de texturas en imágenes a través de una GUI desarrollada en MATLAB, donde se podrán evaluar distintas texturas con diferentes métodos y algoritmos basados en el descriptor LBP, y relacionarlas, a través de una base de datos modificable y extensible, con la textura modelo más cercana posible.

Mencionar que el objetivo final del proyecto no es la identificación perfecta de las texturas a estudio, sino la facilitación de este proceso, pudiendo comprobar, con los diferentes métodos descritos, la variación de los resultados y quedarse con el más deseado.

# Abstract

This project attempts to facilitate, interactively and visually, the analysis of textures in images through an application developed in MATLAB, where different textures can be evaluated with different methods and algorithms based on the LBP descriptor, and relate them, through a modifiable and extensible database, with the closest possible texture model.

To mention that the final objective of the project is not the perfect identification of the textures to study, but the facilitation of this process, being able to check, with the different methods described, the variation of the results and retain the most appropriate one.

# Índice

# ÍNDICE DE FIGURAS

# Notación

| | |
|---|---|
| GUI | Graphic User Interface |
| LBP | Local Binary Pattern |
| RGB | Modelo de color RVA (Rojo, Verde y Azul) |
| TU | Textura Unitaria |
| ULBP | Uniform Local Binary Pattern |
| VLBP | Volume Local Binnary Pattern |
| LTP | Local Ternary Pattern |
| RLBP | Rotated Local Binary Pattern |
| CMY | Cyan, Magenta and Yellow |
| HSV | Hue, Saturation and Value |
| PPB | Plantilla de Pesos Binomiales |

# 1 INTRODUCCIÓN

El análisis de texturas ha sido objeto de estudio durante décadas debido a su amplia utilidad en múltiples campos científicos, como la detección y caracterización de tumores cerebrales en imágenes de resonancia magnética o el análisis de superficies en robótica móvil.

La textura se define de diferentes formas dependiendo del ámbito en el que se estudie, desde su etimología del latín, donde se define como la disposición y el orden de los hilos en una tela, pasando por su definición general en que se asocia a la sensación táctil que produce un objeto con una cierta superficie, hasta el conjunto de estructuras o patrones que conforman una cierta textura. Es esta última definición la que se va a tomar para explicar las diferencias entre texturas y para aplicar el único algoritmo usado en la GUI, el LBP (*Local Binary Pattern*).

Aún existiendo diferentes métodos y descriptores para el análisis de texturas se ha elegido el LBP debido a su sencillez computacional y a los buenos resultados que ofrece. Más adelante se realiza una comparación de los distintos métodos para así obtener una mejor compresión del porqué de esta elección.

## 1.1 Motivación y objetivo

La rama de automática de la ingeniería trata de facilitar el desarrollo de cierto tipo de actividades llevadas a cabo por el ser humano. En el caso de este proyecto se busca el agilizar la identificación de texturas a través de una GUI que analiza la imagen que contiene la textura bajo estudio y la compara con una base de datos creada por el usuario y tan extensible como éste desee. Aunque este sea el objetivo, es difícil de alcanzar una automatización completa de este proceso, puesto que el análisis de una misma textura puede dar diferentes resultados según la imagen que la contenga y sus propiedades, tales como la iluminación, el enfoque, la orientación, etc.

Es por lo anteriormente expuesto que se busca facilitar el análisis otorgando al usuario todas las herramientas posibles para obtener un resultado conveniente, pudiendo aplicar distintas metodologías dentro de un mismo algoritmo (LBP). Aún consiguiendo esto, se necesitará de la experiencia humana para la toma de decisiones acerca de la imagen analizada teniendo como meta, en un futuro, la automatización completa de este proceso a través de, por ejemplo, redes neuronales.

## 1.2 Estado del arte

Muchas técnicas aparecieron en el siglo XX para tratar de resolver el problema del análisis de texturas. Entre ellas se clasifican por métodos estadísticos, que tienen en cuenta la distribución en el espacio de la imagen de los niveles de gris; métodos estructurales, que se basan en el uso de una estructura básica que se utiliza para construir patrones más complejos que llegan a conformar una textura y utilizando un criterio que limita la variedad de disposiciones posibles de la textura; o métodos espectrales, que utilizan el espectro de Fourier para indicar la dirección de los patrones bidimensionales periódicos o casi periódicos.

En este proyecto se va a seleccionar uno de los métodos más famosos basado en el histograma de la imagen y propuesto por T. Ojala, M. Pietikäinen y D. Harwood en 1994 [1]. Se trata del descriptor LBP (*Local Binary Patterns*), que, partiendo del modelo de análisis de texturas presentado por L. Wang y D. C. He [2], donde se

utilizaba una textura unitaria (TU) representada por 8 elementos donde cada uno de ellos podía identificarse por uno de los tres posibles valores (0, 1, 2) obteniendo así un total de 6561 valores posibles para cada TU, fue modificado para mejorar su eficiencia pasando a tener una catidad total de 256 valores posibles.

Hasta el día de hoy han aparecido nuevas modificaciones del descriptor LBP, tales como el VLBP (*Volume Local Binary Patterns*, [3]), el LTP (*Local Ternary Patterns*, [4]) o el ULBP (*Uniform Local Binary Patterns*). Algunas de ellas y otras modificaciones no comentadas serán de uso en este proyecto para la comparación de resultados.

## 1.3 Estructura de este documento

La estructura de este documento seguirá el siguiente orden: en el capítulo 2 se describe la metodología usada en la GUI, además de la herramienta informática y el *dataset* utilizados para la creación de ésta, y la obtención de resultados; en el capítulo 3 se explican en profundidad todos los conceptos teóricos utilizados para el desarrollo de los algoritmos y de las diferentes metodologías; el capítulo 4 consta de una guía de programación donde se describen los pasos realizados para el desarrollo de la aplicación, así como en el capítulo 5, complementando la información anterior y a nivel de usuario, se explica el correcto uso de la GUI; y en el capítulo 6 se comentan las conclusiones obtenidas del funcionamiento y de los resultados obtenidos, aún no siendo esto último un objetivo per se del proyecto, además de incluir una reflexión sobre los siguientes posibles pasos a realizar en un futuro para la mejora de la GUI.

Se incluye, además, un anexo donde se encuentra la totalidad de código diseñado para la obtención de la aplicación y su funcionalidad con cada uno de los métodos y algoritmos desarrollados, además del *dataset* de imágenes utilizado.

# 2 MÉTODOLOGÍA

Tal y como se explicó en el trabajo fin de grado previo a éste, una metodología típica del análisis y clasificación de texturas podría basarse en el diagrama de la Figura 2-1. De esta metodología se utilizarán ciertos pasos, pero no todos, debido a que la GUI que se desarrolla en el presente proyecto es más específica, ya que se centra en un solo descriptor y no posee una parte de entrenamiento, ya que es el mismo usuario quien establece cuáles son las imágenes que contienen a las texturas utilizadas como patrones.

En la Figura 2-2 se observa el diagrama en el que se basa este proyecto, donde se conserva la primera etapa de adquisición de muestras, que consistirá en la descarga del *dataset* que va a ser utilizado. A continuación, y a partir del *dataset*, se seleccionarán la imagen o imágenes de una misma textura que se utilizarán como patrones de la misma, es decir, se usarán como referencia de una textura.

Una vez elegidas las imágenes patrón, se ha de crear la base de datos que se utilizará en la clasificación, compuesta por una matriz donde las filas correspondan a cada textura patrón y las columnas a cada valor LBP del histograma.

Una vez se ha obtenido la base de datos, se puede pasar al análisis de las imágenes de muestra, cargándolas desde un directorio predeterminado. En este proceso se realiza lo mismo que para la obtención de la base de datos, ya que ambos se componen del histograma LBP de la imagen.

Por último, se realiza la clasificación, que otorgará al usuario la textura asociada, y se evaluan los resultados obtenidos, aplicando diferentes ratios para la determinación de eficiencia y exactitud de algoritmo y método utilizados.

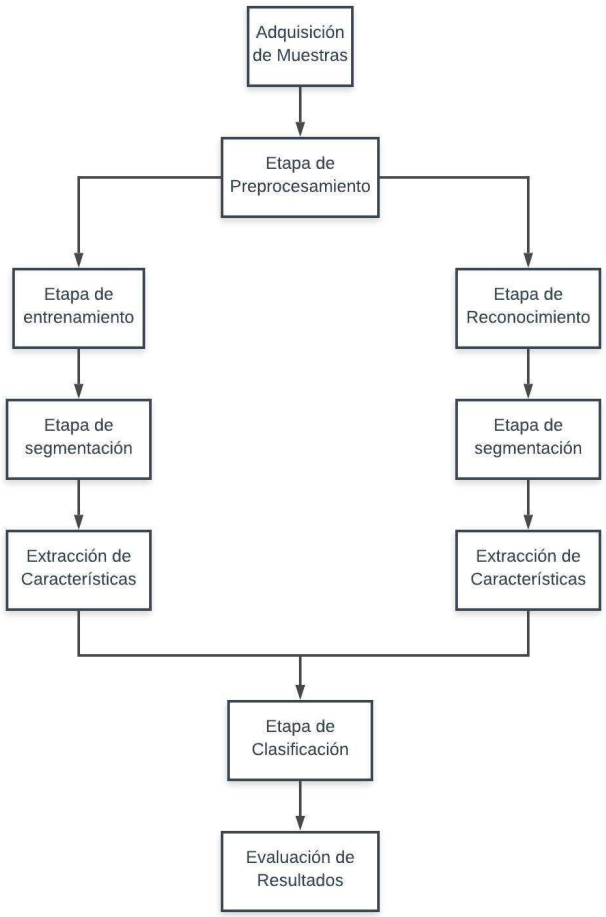



Figura 2-1. Diagrama general de una aplicación clasificadora de texturas

Una vez explicada la metodología que se va a seguir, se puede pasar a explicar la herramienta informática usada para el desarrollo de la misma, así como el *dataset* utilizado para la comprobación del funcionamiento.

## 2.1 Software

Este proyecto se basa en la aplicación desarrollada y su utilidad más que en el algoritmo usado y los resultados obtenidos. Para el diseño de la misma se hecho uso de la aplicación MATLAB de MathWorks [5], en la versión R2016b. Debido a que esta aplicación realiza actualizaciones periódicas y anuales, no se puede asegurar que la GUI aquí desarrollada vaya a funcionar en versiones anteriores a la misma, ya que puede hacer uso de funciones aún no desarrolladas en ese momento.

MATLAB es un sistema de cómputo numérico, con lenguaje propio (lenguaje *M*), utilizado a gran escala en la ingeniería debido a sus utilidades. El paquete MATLAB posee dos herramientas de gran uso, las cuales son *Simulink*, que es una plataforma de simulación de sistemas de control, circuitos electrónicos, etc., y *GUIDE*, que es un editor de interfaces gráficas de usuarios o GUI. Esta última es de la que se ha hecho uso para el desarrollo de este proyecto.

Se ha elegido MATLAB para el desarrollo de est GUI debido a que el lenguaje *M* está orientado a matrices y, como se verá en los conceptos teóricos, las imágenes digitales se tratan como matrices, lo cuál facilita enormemente el análisis de las mismas. Además, MATLAB posee *toolboxes*, o cajas de herramientas, con las que se complementa la funcionalidad del programa y que se van desarrollando a lo largo de los años; una de ellas es el *Image Processing Toolbox*, que posee funciones que facilitan el análisis de propiedades de imágenes, aunque en este proyecto no se vaya a hacer mucho uso de ellas.

## 2.2 Base de datos

La base de datos utilizada en este proyecto se compone por una variedad de imágenes obtenidas de Internet a través de una página de almacenaje y distribución de imágenes sin copyright, por lo que no existe problema alguno con el uso que se le da a las mismas en esta memoria. Dichas imágenes se pueden observer en el apartado 7.2 del Anexo.


Figura 2-2. Diagrama específico del proyecto

# 3 MARCO TEÓRICO

En este capítulo se desarrollarán los conceptos teóricos utilizados para el desarrollo de los distintos algoritmos, basados todos en un mismo descriptor, y de las metodologías utilizadas para el análisis de las imágenes y, por ende, de las texturas.

## 3.1 Conceptos generales

Una imagen digital se considera, a nivel informático, como una matriz *MxN* de datos enteros, donde M indica el número de filas y N el de columnas. Cada elemento de valor numérico y entero que va identificado por los valores *[M, N]* de manera inequívoca se denomina *píxel*, que se entiende también como la unidad mínima de información de una imagen.

El píxel contiene, cuando se trata de una imagen digital, el valor de intensidad que este produce, ya sea en escala de grises o en alguna escala cromática. Como método estándar, el valor mínimo (suele ser cero) corresponde a la ausencia de color, es decir, se corresponde con un píxel completamente negro; y el valor máximo (si se habla de una imagen de 8 bits, que suelen ser las usadas normalmente ya que dicha cantidad es más que suficiente para el ojo humano, este valor es de 255) corresponde al color blanco. Si se está analizando una imagen en escala de grises, los valores de píxel que se encuentren entre el mínimo y el máximo se corresponderán a los niveles intermedios de grises que se peuden obtener con 8 bits, es decir, 254 tipos de gris. Si en vez de escala de grises se encuentra en alguna escala cromática, los valores intermedios corresponderán a la intensidad de color puro existente entre el negro y el blanco, encontrándose su valor "puro" justo en la mitad de la escala, es decir, en el valor 127.

En la Figura 3-1 se puede observar el *zoom* realizado a una imagen digital y como se traduce la matriz de píxeles a una matriz numérica. La imagen se ha sacado de la base de datos que posee MATLAB para poner en práctica los conocimientos adquiridos sobre tratamiento de imágenes.

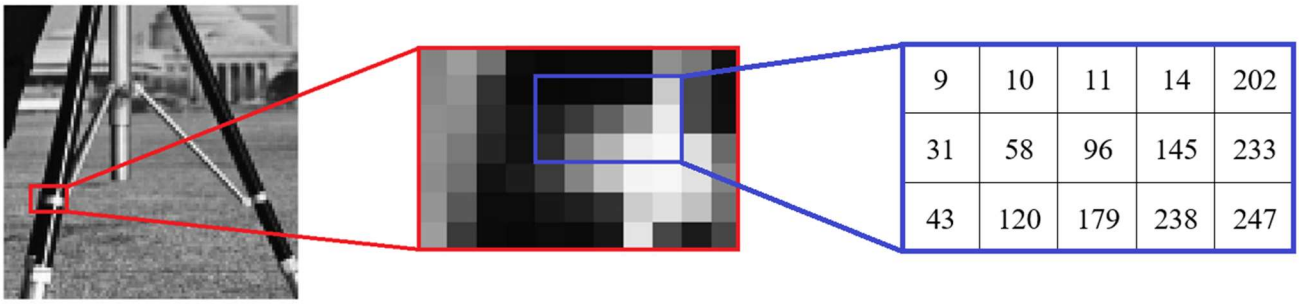



Figura 3-1. Imagen en escala de grises con ampliación y tabla de valores de intensidad

### 3.1.1 Color

El color puede obtenerse a través de distintas representaciones que vienen dadas a través de varios modelos normalizados existentes, entre los que se encuentran el *RGB*, el *CMY* y el *HSV*.

El modelo *CMY* (del inglés, *Cyan, Magenta and Yellow*) es un modelo sustractivo, lo cual quiere decir que se basa en la absorción de la luz a través de los pigmentos de colores, ya que estos pigmentos absorben parte del espectro de luz visible y reflejan la frecuencia con la que se identifican. Se denomina así puesto que la unión entre los tres pigmentos primarios sobre un fondo blanco da lugar al pigmento negro, el cual se caracteriza por la absorción de todas las frecuencias del espectro visible. En la Figura 3-2 se observa el modelo gráficamente.

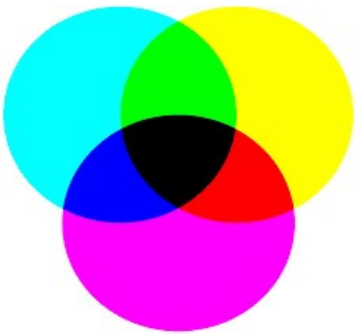


Figura 3-2. Modelo CMY

El modelo *HSV* (del inglés, *Hue, Saturation and Value*) es un modelo que hace uso de los componentes del color en sí y que se puede representar a través de la figura geométrica de un cono. El matiz o *hue* se representa como el ángulo que gira alrededor del eje principal del cono desde 0º a 360º (a veces se encuentra normalizado de 0 a 100%) donde cada grado representa un color u otro. La saturación o *saturation*, que se mide de 0 a 100%, se define como la distancia desde la superficie del cono al eje principal, obteniendo el color una tonalidad más grisácea cuanta menos saturación posea. El valor o *value* hace referencia a la altura del eje principal, o sección circular del cono, donde se encuentra el color deseado: si se hiciese referencia al vértice del cono como el valor 0%, cualquier color determinado por un *Value* de 0 sería negro, pues es un punto por definición; si se utilizase el 100% de *Value*, el color podría ser, dependiendo de la saturación, el color blanco o el color determinado por el matiz más o menos saturado (se podría pensar como la sección circular más grande del cono, es decir, la opuesta al vértice). En la Figura 3-3 se observa el modelo representado gráficamente con forma de cono.

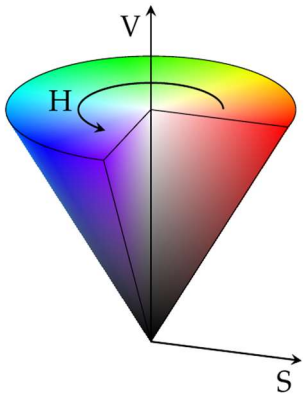



Figura 3-3. Modelo HSV

El modelo RGB (del inglés, *Red, Green and Blue*) es un modelo aditivo y es el más utilizado por norma general. Se denomina así puesto que se basa en la adición de los colores primarios de la luz y en como el ojo humano percibe el color, lo cual se hace a través de los denominados *conos*, que son unas células fotosensibles que posee el ojo y que se encargan de obtener información del color. Estos *conos* son capaces de discriminar la longitud de onda de la señal recibida de forma que existe un tipo de cono por cada color principal, o lo que es lo mismo, por tipo de frecuencia: alta para el color rojo, media para el verde y baja para el azul. En la Figura 3-4 se observa el modelo RGB representado gráficamente.

La adición de estos colores primarios en distinta cantidad da lugar a la escala cromática. En imágenes digitales, esto se consigue utilizando tres plantillas en escala de grises de la misma imagen, donde cada una de ellas corresponde a uno de los colores, que se suman para obtener las distintas tonalidades. Por ejemplo, si el píxel (1,1) de la imagen posee un valor de 255 en todas las plantillas, la suma de los tres colores primarios da lugar al blanco. Si se quisiese conseguir el color amarillo, bastaría con que la plantilla del color azul estuviese a cero y que, tanto la verde como la roja, valiesen lo mismo, observando un color más saturado para el valor máximo de 255 y un tono cada vez más grisáceo para valores más bajos. En la Figura 3-5 se pueden observar las distintas las plantilas RGB en escala de grises y el resultado final tras su combinación.

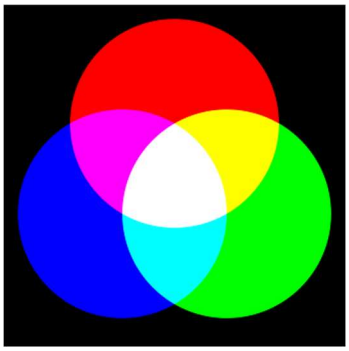


Figura 3-6. Modelo RGB

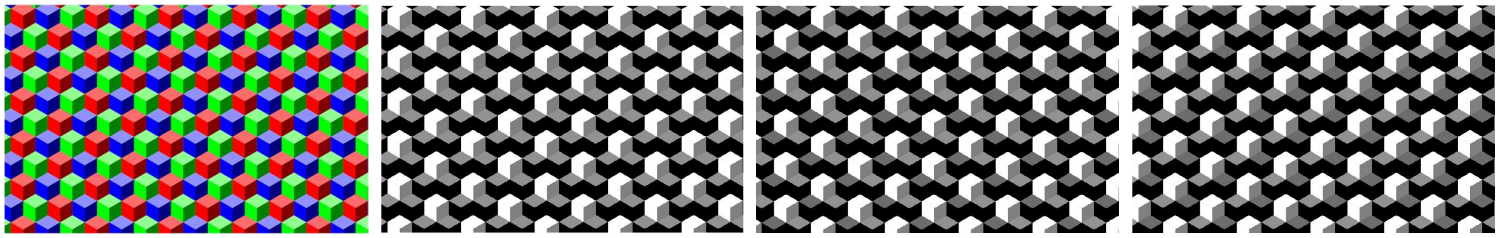


Figura 3-7. Imagen RGB y sus plantillas Red, Green y Blue, respectivamente

### 3.1.2 Histograma

El histograma es una forma de representar la frecuencia con la que se repite un mismo nivel de intensidad en los píxeles de la imagen. Se podría decir que dan información acerca de la población de intensidades luminosas de una imagen.

En el eje de abscisa del histograma se encuentra la variable de la cual determinar su frecuencia, la cual en este caso es la intensidad luminosa que comprende un rango desde 0 a 255, como ya se ha comentado anteriormente. El eje de coordenadas da información acerca de la frecuencia de aparición y esto normalmente se consigue indicando el número de veces que ha aparecido el valor en cuestión de forma bruta. En este proyecto, el histograma representará en el eje de abscisa el valor LBP, que también va de 0 a 255, excepto en sus variaciones, de las cuales se hablará más adelante; y en el eje de coordenadas, la frecuencia de repetición de cada valor LBP normalizada, es decir, está dado en porcentaje (de 0 a 100%), por lo que la suma de todos los valores de frecuencia en cada uno de los valores LBP ha de dar un total del 100%.

En la siguiente figura se puede observar una imagen en escala de grises junto a su histograma de intensidad luminosa.

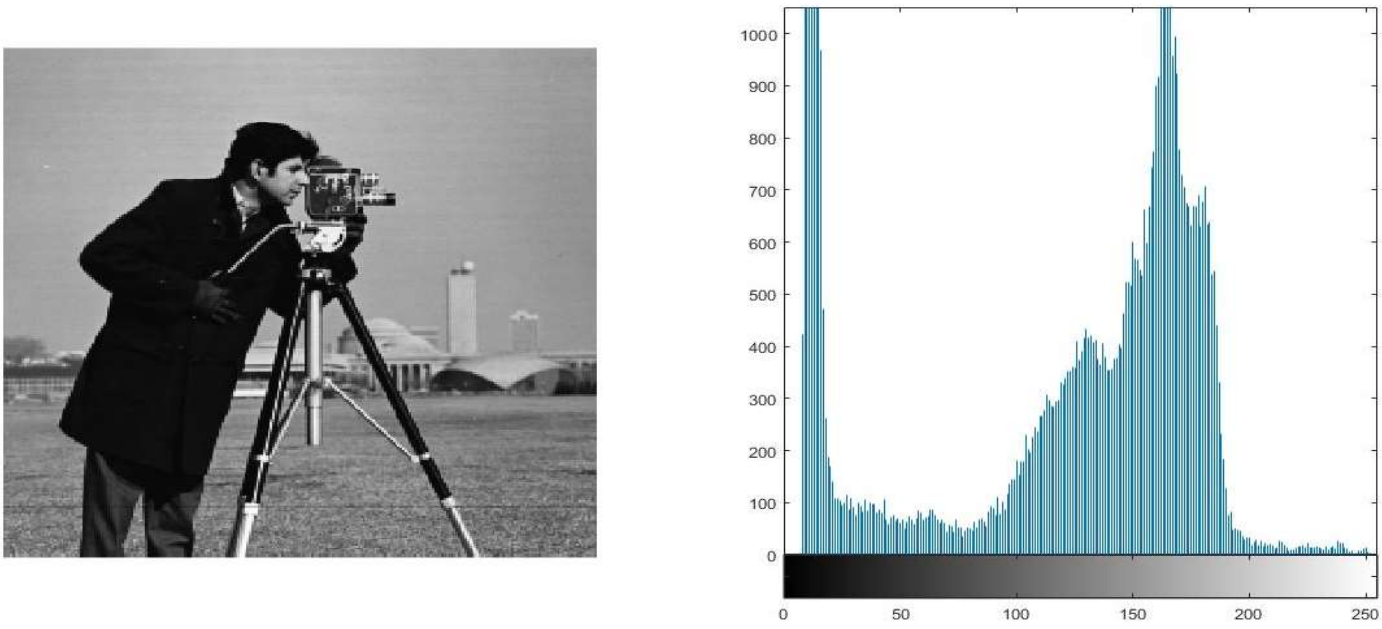



Figura 3-8. Imagen en escala de grises junto a su histograma

## 3.2 Local Binary Pattern (LBP)

El LBP (en español, *Patrón Binario Local*) es un descriptor de texturas basado en histogramas y, tal como su nombre indica, analiza la repetición de patrones binarios de forma local en la imagen. El algoritmo del LBP realiza un análisis alrededor de cada píxel de la imagen observando el valor de sus vecinos y comparándolo con el del píxel en cuestión.

Este análisis se realiza con los píxeles contiguos al que se encuentra bajo estudio, es decir, si se plantea una matriz 3x3, el píxel analizado sería el central y sus vecinos serían los ocho restantes. Lo que se realiza una vez se obtienen los píxeles vecinos, es compararlos valores de estos con el del central, y se crea una matriz binaria del mismo tamaño, donde el píxel central no posee valor o éste es indiferente, pues no se va a utilizar, y las posiciones vecinas adoptan un 1 o un 0 si su valor es mayor o menor que el del píxel central, respectivamente.

Una vez obtenida la matriz binaria resultante de la comparación, ésta se multiplica elemento a elemento por una matriz de pesos binomiales, como la que se puede observar en la Figura 3-7 en la parte inferior de la zona central. Esta matriz puede obtener diferentes formas, en cuanto a la distribución de las potencias de dos alrededor de la misma. Para la aplicación del algoritmo LBP simple se ha utilizado la distribución observada en la figura anteriormente comentada, empezando por la potencia 0 en la posición (1,1) de la matriz y siguiendo con las siguientes en el sentido de las agujas del reloj, aunque ésta será modificable posteriormente, como se verá más adelante. Al multiplicar elemento a elemento ambas matrices, se obtendrá una matriz del mismo tamaño que las anteriores donde solo las posiciones donde el valor del píxel era mayor que el del píxel central serán distintas de cero y tendrán como valor su correspondiente potencia de 2.

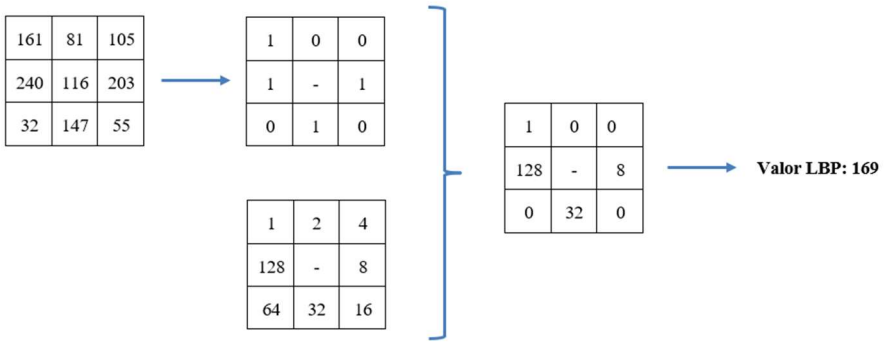



Figura 3-9. Proceso de obtención de valor LBP de un píxel

Una vez obtenida esta última matriz, los elementos de ésta se suman, obteniendo un valor comprendido entre 0 (si todos los elementos de la plantilla binaria son 0) y 255 (si todos los elementos de la plantilla binaria son 1). Este valor obtenido es el denominado valor LBP del píxel al que se ha sometido a estas operaciones, o más bien a sus vecinos. El valor LBP da información acerca del entorno de cada píxel de la imagen, excepto de los píxeles que comprenden el borde de la misma, pues de ellos no se puede sacar una vecindad de 8 y por esa razón se desechan. Esta información puede ser interpretada tal que, si dos píxeles de una imagen poseen el mismo valor LBP, ambos tendrán una estructura parecida alrededor, aunque una de las limitaciones de este descriptor se encuentra en que, en la misma situación, las vecindades de los dos píxeles pueden ser completamente distintas, tal y como se muestra en la Figura 3-8. Esto no quiere decir que sea un mal descriptor, puesto que lo que se busca es que cada textura presente un histograma lo más parecido posible al que se ha determinado como su referencia, y esto sí se consigue aún con dicha limitación.

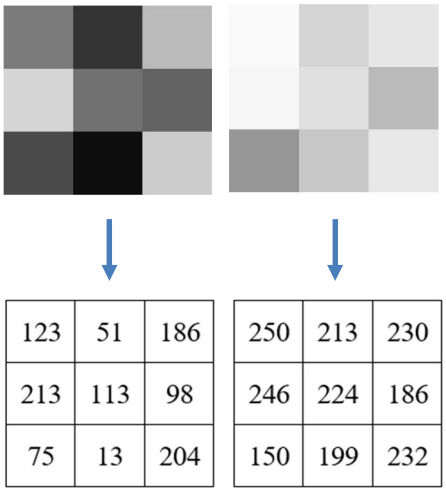



Figura 3-12. Comparación entre dos matrices con mismo valor LBP

Con todos los valores LBP de cada uno de los píxeles de la imagen se puede realizar un histograma a través del cual se identificará la textura perteneciente a la imagen. En la figura 3-9 se puede observar la similitud entre los histogramas LBP de la misma textura obtenidos a partir de dos imágenes diferentes.

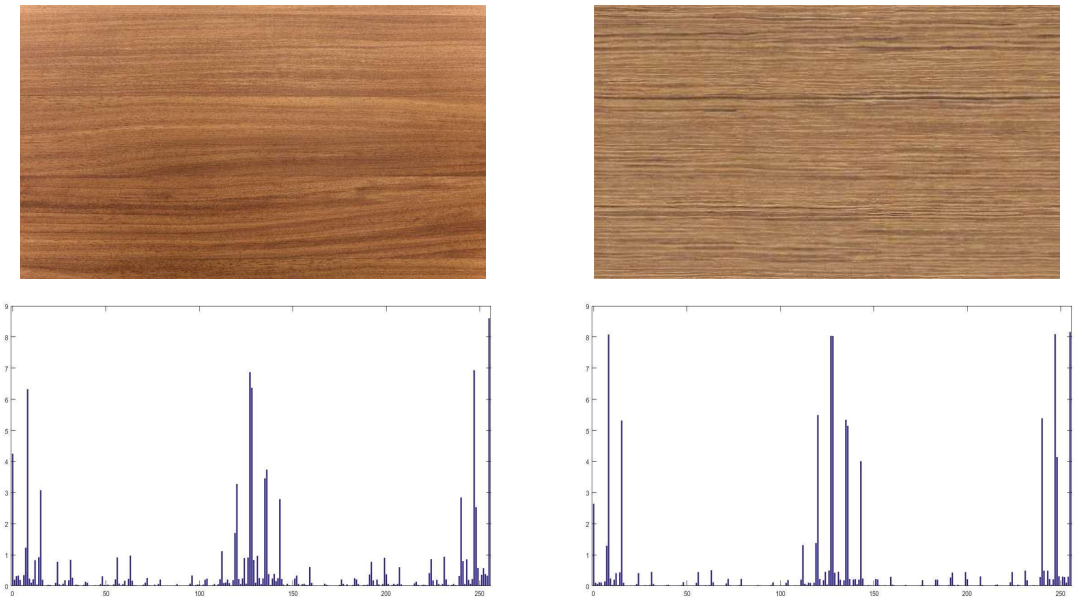


Figura 3-13. Comparación entre histogramas de dos imágenes con una misma textura

Existen varias modificaciones del descriptor LBP que mantienen la misma idea, pero ésta se lleva a cabo con otras consideraciones, simplificaciones o mejoras. En la GUI desarrolada en este proyecto se va a dar la opción de elegir entre distintas de estas modificaciones, por lo que entra dentro de este marco explicar las mismas, lo cual se realiza a continuación.

### 3.2.1 Uniform Local Binary Pattern (ULBP)

La primera modificación del algoritmo LBP es el denominado ULBP, que se diferencia del inicial por hacer una simplificación, la cual consiste en seleccionar solo una serie de patrones binarios cuyas características se mencionan a continuación.

Para comprender esto se ha disponer de la plantilla binaria resultante de la comparación de los valores de intensidad del píxel con sus vecinos y presentarlos como un tren o vector binario, comenzando por la posición (1,1) de la submatriz 3x3 y siguiendo el sentido de las agujas del reloj alrededor del píxel central al cual se le está calculando el valor ULBP. Una vez se obtiene el vector binario hay que observar el número de transiciones de 0 a 1 o viceversa que existen en él. Si este número de transiciones es menor o igual a 2, se dice que el valor LBP resultante es uniforme.

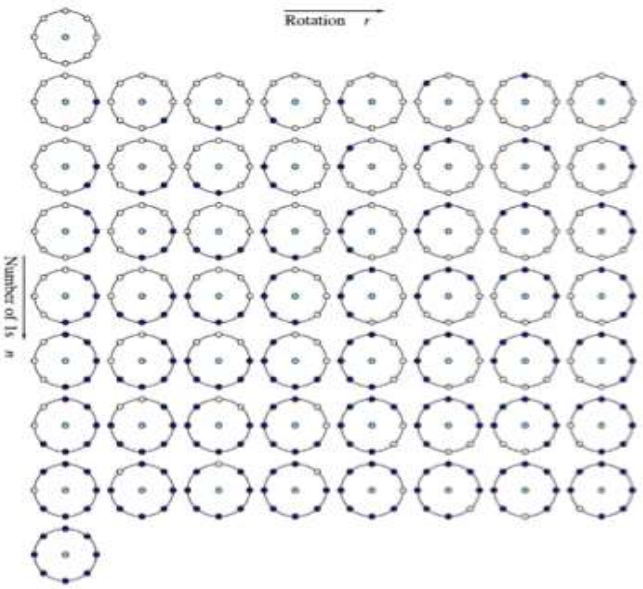



Figura 3-14. Posibles combinaciones percibidas por ULBP (fuente de imagen en [8])

Conforme con lo anterior, la simplificación que realiza este algoritmo se encuentra en seleccionar únicamente aquellos valores LBP procedentes de vectores binarios que sean uniformes. Si se inspeccionan todos los patrones posibles, de los 256 iniciales únicamente existen 58 que son uniformes (las combinaciones posibles se muestran en la Figura 3-10), por lo que el histograma se simplifica de

manera notoria. Lo que se puede ver como una simplificación que empeora el funcionamiento del algoritmo es en realidad una mejora más allá de la eficiencia, puesto que, además de dar información de la textura general de la imagen, el ULBP otorga información extra sobre las partes de las texturas, como se puede observar en la Figura 3-11. En esta se observa como se le aplica el algoritmo ULBP a una imagen (superior izquierda) y se obtiene su histograma (superior derecha), donde una de las zonas viene marcada en verde. Dicha zona representa, en gran medida, una parte de la imagen en concreto, en este caso las franjas horizontales del enlazado mostrado, pudiendo observarse en la plantilla binaria (inferior izquierda) que las zonas en blanco corresponden a las franjas verticales. Finalmente, se puede observar el resultado de superponer la plantilla binaria traducida a RGB, con código (0, 255, 0), con la imagen original.

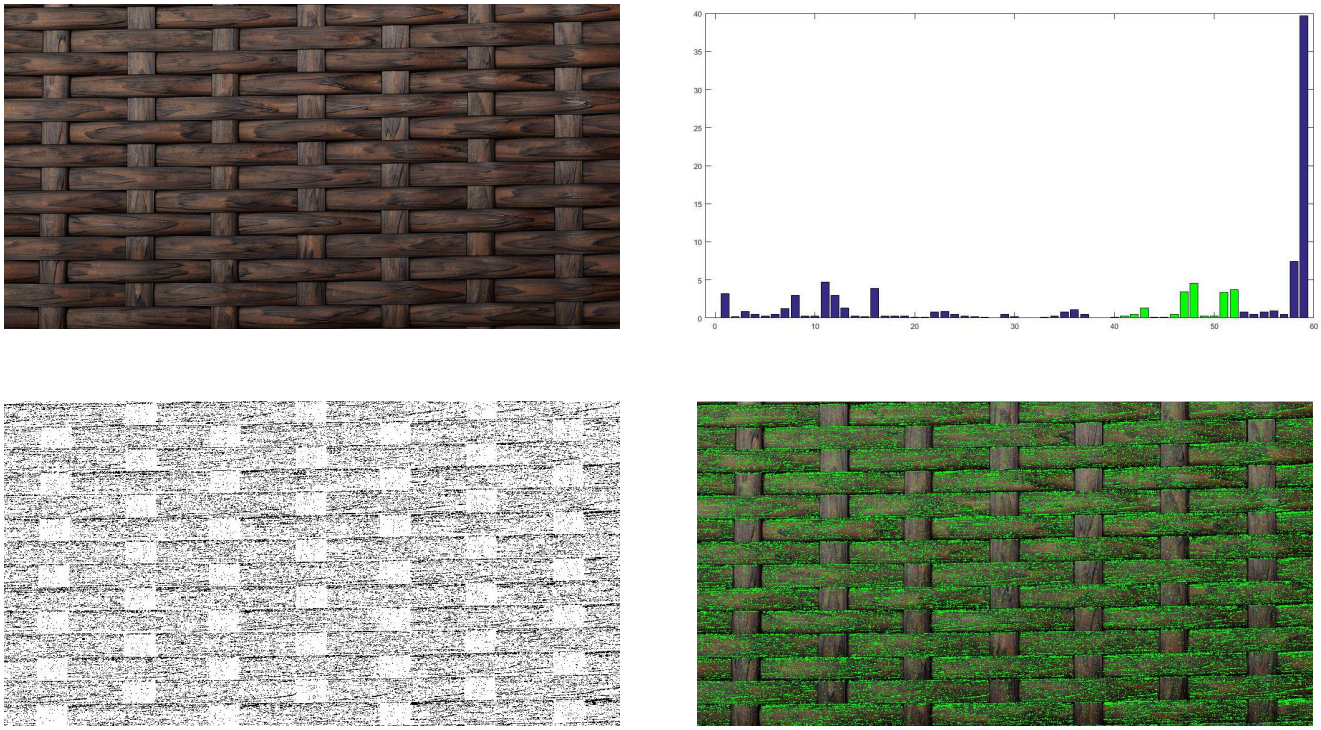


Figura 3-15. Detección de zonas una textura a través de ULBP

Así mismo, uno de los autores del descriptor LBP, Timo Ojala, desmostró que el descriptor ULBP contiene más del 90% de la información de la imagen, por lo que la pérdida por la simplificación comentada anteriormente se puede considerar despreciable. Esto también se puede interpretar como que más del 90% de los píxeles de una imagen poseen valores LBP uniformes, mientras que menos del 10% no.

### 3.2.2 Rotated Local Binary Pattern (RLBP)

La siguiente modificación es tal que se obtiene una invarianza frente a rotación de la imagen, lo cual se puede conseguir con más de una modificación del algoritmo LBP, como RLBP, que es el que se ha implementado, o BRINT_S (del inglés, *Binary Rotation Invariant and Noise Tolerant_Sign*). Se ha elegido el primero de estos porque resulta ser más específico mientras que el otro cumple más especificaciones aparte de la invarianza a rotación, aunque se podría implementar en un futuro para ampliar las capacidades de la aplicación.

En la modificación elegida, el RLBP, se realiza un único cambio con respecto al LBP original, el cual consiste en la orientación de la plantilla de pesos binomiales aplicada sobre la plantilla binaria resultante del primer paso de comparación entre píxel central y sus vecinos. La orientación de esta plantilla viene dada por la posición del píxel vecino con mayor valor entre todos ellos. En la Figura 3-12 se pueden observar las distintas orientaciones

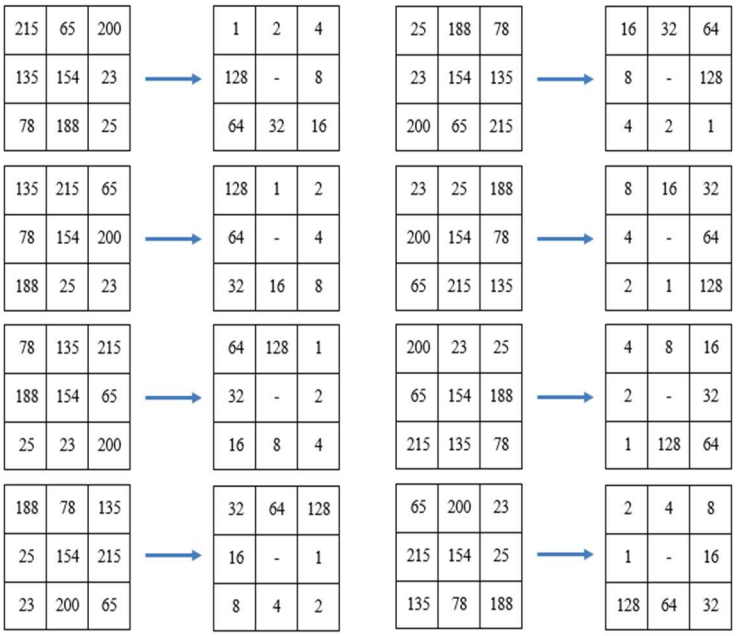



Figura 3-16. Posibles orientaciones de la matriz de pesos binomiales

de la plantilla de pesos binomiales según la posición del píxel de valor máximo.

Una vez obtenida la PPB, el resto del algoritmo se aplica de la misma forma que el LBP original. La elección de la orientación se realiza de esta forma para que un mismo patrón se pueda detectar si la imagen está rotada cierto ángulo, como se observa en la Figura 3-13.

Las limitaciones de este algoritmo se encuentran en que, si hay dos píxeles de una misma vecindad con el mismo valor y ambos son el máximo, solo se podrá elegir uno de ellos y esto dependerá del orden en que aparezcan, ya que la submatriz 3x3 se observa desde su casilla (1,1) y en sentido horario, lo cual puede dar lugar a que un mismo patrón con diferente orientación dé un resultado diferente. Aún así, esta modificación obtiene mejores resultados cuando las imagenes están rotadas respecto a la referencia de textura usada.

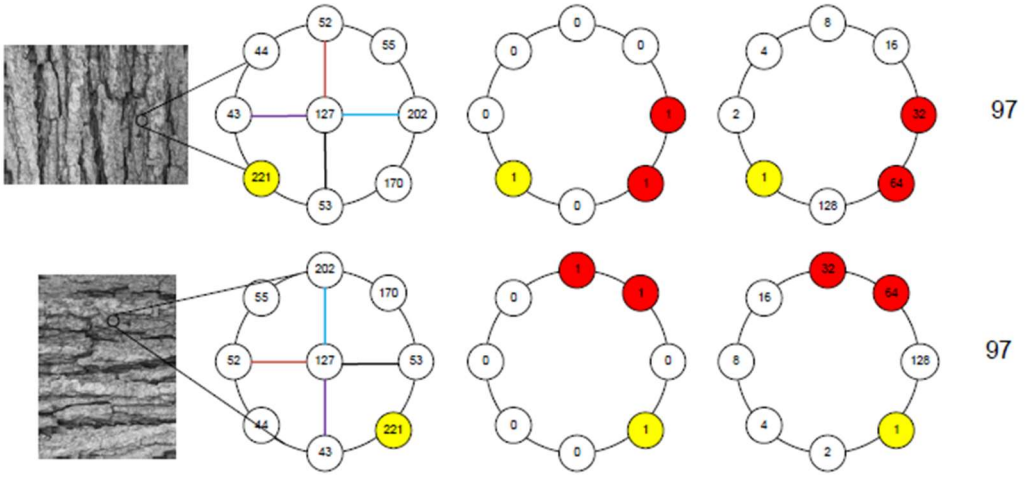



Figura 3-19. Funcionamiento del algoritmo RLBP

# 4 GUÍA DE PROGRAMACIÓN

En este capítulo se describirá el proceso de programación llevado a cabo para la obtención de la GUI, concretando en las funciones relacionadas con la pulsación de botones o la selección de parámetros, así como la representación de resultados y las funciones auxiliares creadas para "limpiar" un poco el código principal. Se va a empezar explicando estas últimas, ya que van a ser usadas en la función principal de la GUI y es preferente entender su funcionamiento antes.

## 4.1 Funciones auxiliares

Las funciones a continuación expuestas son en las que se encuentran la aplicación de los algoritmos usados y una función extra para el tratamiento de la imagen a priori de la aplicación del algoritmo.

### 4.1.1 fun_LBP

La función *fun_LBP*, al igual que las dos siguientes, trata de la aplicación del algoritmo cuyo nombre aparece en su llamada. Todo lo comentado de esta función se puede observar en el propio código de la misma en el capítulo 7.1 del Anexo, Apartado A. Como parámetros de entrada a esta y demás funciones del mismo tipo que se explicarán seguidamente se encuentran la imagen de la cual obtener los valores LBP (en este caso) y un parámetro llamado *clase* que se utiliza para diferenciar entre los casos en los que se está aplicando la imagen de entrada, es decir, si esta viene dada por su formato original (en color) o si viene dada una subimagen en escala de grises. Esta diferenciación se explicará más adelante cuando se comente la función que se aplica para el segundo caso.

Nada más entrar a la función, se ha de comprobar su formato, para lo cual se utiliza la variable comentada, y, en caso de que la imagen venga dada en su formato en color (*clase* igual a 1) se ha traducir ésta a escala de grises para obtener una única matriz de datos. Este cambio de escala cromática se realiza con la función *rgb2gray*. Lo siguiente es determinar el tamaño de la matriz que contiene a la imagen, ya que esto será de uso para recorrer la misma píxel a píxel. Esto se hace con la función *size*, que, aplicada a una matriz, da como resultado el número de filas (variable *M*) y de columnas (variable *N*).

Antes de empezar a estudiar la imagen dada, se inicializan dos matrices: *lbp*, del mismo tamaño que la imagen bajo estudio donde se guardará la imagen de valores LBP; y la matriz fila *v*, donde se guardarán los valores de repetición de cada valor LBP posible, por lo que su tamaño ha de ser de 256. Ambas matrices se inicializan a cero, aunque en el caso de la matriz *lbp* se podría haber inicializado con cualquier valor, ya que éste va a ser sustituido posteriormente. No es el caso del vector fila *v*, que se necesita que esté inicializado a cero porque lleva una cuenta de las veces que se repite cada valor LBP. El hecho de inicializar dichas matrices se apoya en la eficiencia computacional del programa, ya que en el caso contrario se irá incrementando el tamaño de ambas según se avance en el código, cosa que es poco deseable, así que simplemente se inicializan para reservar el espacio en memoria que van a tener al final de la función.

Ahora toca realizar el estudio de la imagen píxel a píxel y obtener el valor LBP de cada uno de ellos, para lo cual se realizan dos bucles *for* anidados, de modo que el primero recorra las filas y el segundo las columnas de la imagen. Estos bucles se realizan entre 2 y el valor máximo de filas o columnas (según el caso) menos 1. Esto se debe a lo ya explicado en el marco teoríco, donde se comentaba la necesidad de eliminar los bordes antes del estudio de la imagen ya que de estos no se va a poder sacar un valor LBP normal puesto que no poseen una vecindad de 8 píxeles. De esta forma se consiguen evitar los bordes de la imagen original.

Una vez dentro del último bucle *for*, las operaciones realizadas se hacen en base al píxel que se encuentra en ese instante dado por las coordenadas de ambos bucles. Lo primero es inicializar una submatriz *3x3* llamada

*umbral* que será utilizada como plantilla final de pesos binomiales. Esta plantilla es importante que se inicialice a cero, puesto que, si un píxel asociado de la submatriz estudiada no cumple la condición de poseer un valor mayor al del central, no cambiará su valor, y éste debe ser cero, pues será su peso binomial asociado. Las operaciones siguientes se basan en la observación de sus píxeles vecinos y la comparación de sus valores con el del píxel central bajo estudio, tal como se explicó en el marco teórico. Para realizar esta comprobación se han utilizado sentencias *if* para cada unos de los píxeles vecinos, por lo que se encuentran un total de 8 sentencias de este tipo. Dentro de cada sentencia, si se confirma la condición, en la que se ha establecido que el valor del píxel vecino es mayor que el del central, se cambia el valor correspondiente de la plantilla *umbral* al peso binomial asociado por su posición. La obtención de la plantilla de pesos binomiales se podría haber realizado también con otra pareja de bucles *for* anidados para recorrer la submatriz *3x3* de una forma más automática, pero como el coste computacional sería mayor y se desea una velocidad lo suficientemente grande como para que la aplicación sea fluida, se intentan evitar.

Una vez obtenida la matriz de pesos binomiales asociada al píxel bajo estudio, se obtiene el valor LBP del mismo sumando todos los valores de ésta, lo cual se realiza con un doble uso de la función *sum*, ya que un solo uso daría como resultado un vector con las suma de los valores de cada columna. El valor resultante se guarda en la matriz *lbp* en la posición del píxel del cual se ha obtenido, es decir, la posición dada por las coordenadas de los bucles. Este mismo valor se utiliza para rellenar *v*, de forma que la posición de este vector dada por el valor LBP hallado (*lbp(i,j)*) más 1 (un vector en MATLAB comienza en 1 y la lista de valores posibles de LBP comienza en 0) se incrementa en una unidad. Una vez realizado esto ya se termina el estudio del píxel y se pasa al siguiente.

Una vez terminados los bucles *for* anidados, se ha obtenido una imagen de valores LBP y un vector de cuenta de repeticiones de los mismos. Este último posee valores absolutos de repetición, y realmente se desea exportar dicha información en tanto por ciento. Por esto, a los valores contados en *v* se les realiza la operación de escalado a porcentaje, donde el denominador corresponde al número de píxeles analizados (número total de píxeles de la imagen oiginal menos los bordes) y el numerador es el 100 correspondiente al número total de píxeles. Por último, para poder visualizar la imagen de valores LBP, la matriz *lbp* se transforma a *uint8*, ya que su formato original era *double* y no podría mostrarse como imagen utilizando *imshow*. Estos dos valores son los que devuelve la función.

### 4.1.2 fun_ULBP

Esta función contiene una estructura similar a la anterior, la cual se puede comprobar en el Apartado B del capítulo 7.1 del Anexo. Las bases del algoritmo son las mismas, pero posee diferencias notables que han de ser explicadas. El comienzo es idéntico en cuanto a obtención de la imagen e inicialización del vector *v*, que en este caso tiene una longitud de 58, y de la matriz en este caso llamada *ulbp*, que cumple la misma función que *lbp* en el caso anterior.

La primera diferencia se da al inicializar un vector columna auxiliar, *pesos_ULBP*, que es utilizado para identificar qué valores LBP obtenidos son uniformes, finalidad de este algoritmo. Este proceso de identificación se ha probado de distintas formas, obteniendo como más eficiente la utilizada finalmente, la cual consiste en guardar en un vector de longitud 256 los valores ULBP correspondientes a cada valor LBP posible, caracterizando a los que son verdaderamente uniformes desde el 1 al 58, y los que no lo son con un 59. Gracias a esto se podrán desechar todos los valores que den como resultado más de 58.

Una vez definido lo anterior, se pasa a un doble bucle *for* anidado idéntico al del apartado anterior, en cuyo interior aparece también la matriz *umbral* y las sentencias *if* que hacen cambiar la plantilla de pesos binomiales. Esto es así debido a que, hasta este momento, el ULBP y el LBP no difieren en cuanto a procedimiento, ya que lo único que hace es seleccionar ciertos valores LBP y desechar los demás. Se vuelve a obtener el valor LBP, pero ahora se almacena en una variable auxiliar *val* y, una vez obtenido, se utiliza para obtener su correspondencia ULBP del vector descrito anteriormente. Si este valor ULBP es menor de 59, se procede a incrementar su contador en el vector *v*. Independientemente de su valor, éste se guarda en la matriz *ulbp* que, cuando se vaya a representar, es predecible que la imagen va a tener una tonalidad bastante oscura, puesto que, dentro de la escala de grises, solo se estarán utilizando los valores de 0 a 58. Finalmente, se vuelve a realizar la conversión a tanto por ciento de los valores de *v* y se pasa a formato *uint8* la matriz *ulbp*.

### 4.1.3 fun_RLBP

Como en las anteriores, se puede observar el código de esta función en el apartado C del capítulo 7.1 del Anexo. Esta función ha de conseguir un vector *v* de contador de repeticiones del valor RLBP y una matriz *rlbp* que contenga la imagen en valores RLBP de la imagen original. Es por ello que las similitudes, en cuanto a las inicializaciones, con la función *fun_LBP* siguen existiendo, pero también aparece un nuevo apartado en el que se inicializan 8 tipos distintos de plantilla binaria. Todas estas plantillas son crecientes en sentido horario y entre ellas solo cambia la posición de inicioy, por ende, la posición del resto de pesos. La matriz *plant_1* sería la predeterminada que se ha utilizado en las dos funciones anteriores. Se definen con la finalidad de decidir que orientación aplicar según la posición del mayor valor de píxel vecino, como ya se ha explicado en el marco teórico.

A continuación, con todas las variables necesarias ya inicializadas, se realiza el doble bucle *for* anidado de las funciones anteriores, pero en este caso el interior del mismo cambia bastante. Lo primero que se realiza es declarar un vector *bin* de longitud 8, donde se guardarán los valores de los píxeles vecinos para su posterior comparación y detección del máximo de ellos. De nuevo se han observado diferentes formas de recorrer la submatriz de vecindad, llegando a realizar un recorrido con un bucle *for*, aunque finalmente se ha optado por introducir cada valor de forma manual introduciendo el valor del píxel correspondiente a cada una de las posiciones del vector, puesto que se ha vuelto a observar una mejora en el rendimiento de la aplicación, concretamente en la espera producida por la actualización de la base de datos en este modo.

Acto seguido se busca el valor máximo de los píxeles vecinos aplicando la función *max* al vector *bin*, la cual devuelve tanto el valor como la posición de este en el vector. Como se han asociado los valores al vector *bin* de tal forma que correspondiese con la plantilla *plant_1*, es decir, la posición 1 del vector corresponde con la celda (1,1), la segunda posición con la celda (1,2), y así con los demás en sentido horario, la posición donde se encuentre el valor máximo de los píxeles vecinos, almacenada en la variable *ind*, otorga a su vez la plantilla de pesos binomiales que ha de ser utilizada para el cálculo del valor LBP. Esto último se ha conseguido con la serie de sentencias *if/elseif* que se puede observar a continuación del cálculo del valor máximo.

Una vez calculada la plantilla de pesos binomiales, se ha de calcular la plantilla binaria correspondiente a la comparación de valores entre píxeles vecinos y píxel central, por lo que se vuelve a utilizar la matriz *umbral* y se realizan las mismas sentencias *if* para la asignación del valor, con la diferencia de que esta vez no se incluirán directamente los pesos binomiales, sino los valores resultantes de la comparación, es decir, 0 ó 1. Una vez obtenidas la plantilla de pesos binomiales y la binaria, ambas se multiplican punto a punto, es decir, celda a celda, para obtener la matriz *mat_rlbp*, que realizará la misma función que la matriz *umbral* en las dos funciones anteriores. El valor LBP con el método de invarianza a rotación se consigue entonces de la suma de todas las celdas, asociando cada uno al píxel correspondiente de la matriz de valores RLBP, y se suma en el vector de repeticiones *v* una unidad más en el correspondiente valor. Una vez finalizado el análisis de la imagen se realizan, como en los casos anteriores, la conversión a *uint8* de la matriz *rlbp* y la normalización del vector *v*.

### 4.1.4 fun_class_3_prot

Esta última función es la utilizada cuando se realiza un pretratamiento a la imagen original y se obtienen tres subimagenes de la primera a modo de prototipos. Este procedimiento se basa en crear, de forma aleatoria, 3 imágenes cuyos píxeles proceden de la imagen original y que no se repiten en las otras subimágenes. Este concepto de aleatoriedad hace que el proceso de detección de textura mejore, ya que, aunque parezca que se van a eliminar los patrones que caracterizan a cada una de ellas, estadísicamente y con la cantidad de píxeles que existen en una imagen estándar (del orden de $10^5$), se comprueba que dicha repetitividad en la imagen no se pierde, si no que se acentúa más y, bajo la metodología aplicada, se obtienen 3 imágenes para comparar, lo que genera menor margen de error.

Una vez explicado el por qué de esta metodología, se procede a explicar el modo de programación de dicha funcionalidad, la cual se puede observar en el apartado D del capítulo 7.1 del Anexo. Como ya se ha ido observando, lo primero a realizar es la conversión de RGB a escala de grises con la función *rgb2gray*. Es aquí donde se encuentra la explicación de la existencia del segundo parámetro en las funciones de llamada a los algoritmos LBP: como en estafunción se realiza la conversión a escala de grises, si se está utilizando esta

metodología para el cálculo de la base de datos o el análisis de una imagen de prueba, las subimágenes generadas que se analizarán con los algoritmos previamente explicados ya se encuentran en escala de grises, por lo que si se intenta de nuevo la conversión *rgb2gray*, el programa generará un fallo y no continuará. Es por esto que, si se usa este método, en las funciones de los algoritmos se omite la conversión, pues ya está realizada.

A continuación, se obtiene el tamaño de la imagen (número de filas y columnas), y este se divide entre 3, redondeando al entero más cercano. Los valores obtenidos serán las dimensiones de las subimágenes a calcular, que se definirán en plantillas numeradas del 1 al 3 y se inicializarán a cero. Además, se inicializan a 1 una serie de variables auxiliares que se corresponderán al lugar del píxel de cada subimagen que se está rellenando dentro del siguiente bucle. Se realiza ahora un doble bucle *for* anidado pero que esta vez sí tiene en cuenta los bordes de la imagen original, pues no se está aplicando ninguna vecindad y se quiere aprovechar la mayor cantidad de información que posee la imagen original. La funcionalidad dentro de este bucle es la distribución aleatoria de los píxeles de la imagen original a las 3 subimágenes deseadas, por lo que debe aparecer la función *randi*, que genera un número entero aleatorio según una distribución uniforme discreta dentro del rango indicado, que en este caso es de 1 a 3. Este valor aleatorio indica a qué subimagen va a ir a para el píxel observado en dicho instante. Una vez seleccionada la subimagen, se introduce el valor del píxel de la imagen original en la posición que indiquen las coordenadas de dicha subimagen, que se habían inicializado a (1,1), y acto seguido, se incrementa la fila si su indicador no ha llegado al número máximo posible, caso en el que se volverá a la primera fila y se incrementará en uno la columna. Con este método, la generación de subimágenes se realizará columna a columna. Una vez se han repartido todos los píxeles de la imagen original y se hayan obtenido las 3 subimágenes de tamaño similar, se convierten a *uint8*, paso que no es necesario porque no se a optado por representar dichas plantillas, pero que puede ser útil si se desea hacerlo.

## 4.2 Función principal (GUI_LBP)

Una vez se han explicado todas las funciones auxiliares que se han utilizado en la GUI, se puede pasar a explicar el programa principal de ésta, donde se realizan las operaciones asociadas a la interación con la interfaz.

A la hora de crear una GUI en MATLAB, el primer paso a realizar es el de crear una nueva plantilla de interfaz gráfica, lo cuál se realiza con el comando *GUIDE*. Este comando hace que se abra una ventana donde se puede elegir crear una nueva GUI, otorgando como opciones una plantilla en blanco (vacía) o con algunas funcionalidades predeterminadas. En este caso se ha elegido la primera opción y a partir de ahí se han ido distribuyendo todas las funcionalidades requeridas según se ha deseado.

Una vez se selecciona la GUI predeterminada que se desea crear, aparece un programa por defecto donde se encuentra la función de inicialización de la misma, la cual no debe ser modificada. La siguiente función que aparece es la llamada *GUI_LBP_OpeningFcn*, la cuál se ejecuta justo antes de que la GUI se haga visible y donde se inicializan las variables creadas por el programador. Como se puede observar en el Anexo, Capítulo 7.1, Apartado E, las variables inicializadas van desde las creadas estrictamente en el programa hasta las que definen propiedades de los objetos de la GUI. Todas ellas se encuentran dentro de la estructura *handles* y para modificarlas se les puede asignar un valor directamente, si se trata de variables de programa como *Samples*, *Method* o *Classifier*, que definen el tratamiento de la imagen a priori (se ha explicado en el apartado anterior), el tipo de algoritmo y el tipo de clasificador , respectivamente; o, si se trata de variables de la GUI, como propiedades de los objetos de la misma, se ha de utilizar el comando *set* para cambiar su valor, como se observa en las variables *Actualizar* o *Resultado*, que son de tipo string y muestran un mensaje estático por pantalla, o como *panel_ini* y *panel_options*, que representan las dos pantallas de las que se compone la GUI y que, a través del parámetro *'visible'*, hacen efectiva o no la visualización de las mismas. Es por eso mismo que en esta inicialización una de ellas está *'on'* y la otra *'off'*, para que solo una de ellas se muestre. Esto último se realiza con el fin de simular el efecto de "pestañas" en una misma ventana. Hay una función más, con nombre *GUI_LBP_OutputFcn*, en la que sus salidas se devuelven a la línea de comandos y que no ha sido utilizada ya que no se ha considerado su utilidad más allá del testeo de la aplicación.

A partir de aquí, las funciones que siguen han sido creadas a partir de los elementos añadidos a la GUI y definen el funcionamiento de estos.

### 4.2.1 Actualizar_Base_Callback

Esta es la primera función de usuario que aparece en el programa, aunque hay que recalcar que el orden de las mismas es indiferente, ya que se activan por llamada desde la GUI, es decir, las funciones se ejecutarán cuando el elemento de la GUI al que están asociadas sea activado. Esta función se activa al pulsar el botón *Actualizar Database* de la pestaña *Opciones* de la GUI y ha de ser la primera en activarse si se utiliza la aplicación por primera vez, ya que es la que crea o actualiza la base de datos de los prototipos de texturas, por lo que, si no hay base de datos previa, no se podrá comparar ninguna imagen de prueba.

Lo primero a obtener una vez se quiere actualizar la base de datos es el directorio donde se encuentran las imágenes que van a comprender la misma. Esto se hace a través de la función *dir*, a la cual se le introduce el nombre del directorio del cual se desea conocer su contenido y además se le pueden indicar archivos en concreto que se desean, ya sea a través de su nombre específico o de un conjunto de archivos definidos por su extensión. En este caso se ha optado por obtener todos los archivos de una misma extensión (*.jpg*), lo cual se hace tal como se ha indicado en el código, es decir, designando un asterísco (*) en el lugar donde se define el nombre del archivo. Si en dicho directorio nunca van a haber otros archivos que no sean las imágenes para la creación de la base de datos, no haría falta especificar extensión. La función devuelve una estructura de los N ficheros que existan en el directorio con las especificaciones descritas con campos como el nombre, la fecha de modificación o si el nombre encontrado corresponde a otro directorio o no. De todos estos solo se utilizará el campo del nombre de cada archivo para poder identificar cada textura.

Una vez almacenada la estructura de ficheros contenidos en la carpeta de base de datos, se inicializa un vector vacío *n* donde se guardarán más adelante los nombres de cada imagen de textura prototipo para la base de datos. El siguiente paso es inicializar y dimensionar la variable *vector*, que almacenará más adelante el contenido del algoritmo LBP, es decir, en cada columna de esta matriz, que posee tantas filas como imágenes haya en el directorio de base de datos, se guardará la cantidad de veces que se repite cada valor LBP en cada imagen para poder realizar el histograma. Es por esto que la inicialización del mismo se ha de realizar en referencia al algoritmo utilizado, lo que se consigue con la sentencia *if/else*, ya que, si se utiliza el algoritmo ULBP, el número de columnas ha de ser de 58, ya que será la cantidad total de posibles valores ULBP. En caso de que no se trate del algoritmo ULBP, la matriz tendrá 256 columnas. Para determinar qué algoritmo se está utilizando para el cálculo de la base de datos se ha de comprobar el valor del *Radio Button* asociado a la variable *Method*, lo cuál se hace observando su valor en la estructura *handles*, tal como aparece en el código. Un valor de 2 indica que se está utilizando el ULBP, mientras que el 1 y el 3 se corresponden al uso del LBP y RLBP, respectivamente.

Lo que viene a continuación es un bucle *for* en el cuál se va iterando entre 1 y la cantidad de imágenes de la base de datos, longitud que viene dada por *length(d)*, para tratar cada imagen prototipo por separado. En cada iteración se almacena el nombre de la imagen que se está tratando en dicho instante en su correspondiente columna del vector *n*. Además, como aviso al usuario, se imprime en la GUI, utilizando la variable *Actualizar* explicada al comienzo del apartado, un mensaje de aviso que muestra el número de imágenes restantes por tratar, por lo que se obtiene información aproximada del tiempo restante que queda para terminar la actualización de la base de datos. Para cambiar el mensaje que muestra la variable se debe utilizar la función *set*, comentado previamente, seguido del comando *drawnow*, ya que sin este último el texto no cambia. Acto seguido se almacena en la variable *im* la imagen a tratar haciendo uso de la función *imread*, que abre un fichero de extensión de imagen y la traduce a una matriz tridimensional *3xMxN*, donde el 3 hace referencia a los tres canales de colores que componen la imagen y el *MxN* hace referencia al tamaño, en filas por columnas respectivamente, de la imagen.

Una vez se tiene la imagen almacenada hay que observar de nuevo que tipo de algoritmo y el número de muestras obtenidas de la imagen. Dentro del número de muestras solo existen dos modos, siendo el primero (una muestra) en el que se aplica el algoritmo sobre la imagen tal cual se recibe con *imread* y el segundo (tres muestras) en el cual se obtienen tres subimagenes procedentes de la original a través de un proceso de selección de píxeles aleatorio, como ya se ha explicado anteriormente.

Para cada tipo de algoritmo disponible se designa una sentencia *if*, dentro de la cuál se realiza la correspondiente llamada a la función deseada según el caso en el que se encuentre. Cabe destacar que este paso se realiza independientemente del número de muestras de la imagen, puesto que, a la hora de presentar resultados en el caso de haber obtenido 3 muestras, se hace con el vector *v* de la imagen original, aunque la clasificación se haya realizado con el trío de subimágenes. Una vez utilizado el algoritmo pertinente sobre la imagen original, se comprueba el contenido de la variable *Samples*, y en caso de que esta indique el uso de 3 muestras, se aplica la función *fun_class_3_prot* a la imagen bajo estudio, obteniendo sus 3 plantillas, y se calculan los vectores *v* de

cada una de ellas, indicando en la llamada a la función de ejecución del algoritmo que no ha de aplicar la conversión a escala de grises, utilizando un valor distinto de 1 en el segundo parámetro de ésta.

Una vez se ha calculado tanto el vector de cuenta *v* como la imagen de valores LBP/ULBP/RLBP, con el algoritmo y el número de muestras de imagen solicitados, se guarda el primero en la fila correspondiente de la matriz *vector*, y si se hn utilizado 3 muestras, también se incluyen sus correspondientes vectores en las matrices *vector1*, *vector2* y *vector3*. Esto es todo lo que se debe realizar para obtener la base de datos, puesto que lo anterior se va a repetir para cada imagen de la carpeta, y cada una de ellas acabará el bucle *for* con un vector asociado.

Antes de dar por finalizado el proceso de creación o actualización de la base de datos, se han de almacenar los valores obtenidos en su correspondiente archivo. Se ha hecho uso de la función *save* para conseguir dicho almacenaje y además obtenerlo de forma permanente, ya que esta función crea un archivo de extensión *.mat* que se guarda en el directorio de trabajo y no se elimina con el cierre de la aplicación o del programa MATLAB. Esto se ha hecho así porque, de lo contrario, el proceso de inicialización de la base de datos habría que hacerlo cada vez que se accediese a la aplicación y cada vez que se cambiase el método o número de muestras según los que se obtiene la base de datos. Es por ello que se crea una base de datos diferente para cada posible combinación entre algoritmo usado y número de muestras, que además obtendrán una longitud y una lista de nombres características, por lo que se puede cambiar la base de datos de una combinación y mantener las demás intactas. Antes de guardar los diferentes valores obtenidos, se definen la variable *Tam*, que contiene el tamaño de la base de datos, y el vector *Names*, con los nombres de todas las imágenes almacenadas, sin la extensión *.jpg*, para lo cual se somete al vector *n* a la función *erase*. Una vez obtenidas estas dos variables, con uso similar a la variable *vector*, se pasa a distinguir entre cada combinación posible con las sentencias *if/else* observadas en el código, y, dependiendo de la misma, se guardan las tres variables comentadas con el nombre que las identifique. Los formatos utilizados para los nombres de los archivos *.mat* que conforman la base de datos son el nombre del algoritmo utilizado seguido de *_x*, donde *x* es el número de muestras utilizadas, para almacenar la variable *vector*; el nombre del algoritmo usado seguido de *_3x*, donde aquí *x* es una letra (a, b ó c) que identifica al vector obtenido de cada subimagen en el caso de haber usado 3 muestras; el formato *Length_(Nombre del Algoritmo)_x* para el almacenaje del tamaño de la base de datos correspondiente dado por la variable *Tam*, donde *x* realiza la misma función que en el primero de los casos; y el formato *Names_(Nombre del Algoritmo)_x* para el vectorde nombres dado por la variable *Names*, donde *x* indica también el número de muestras. Como se puede observar en el código, la función *save* funciona introduciéndole como primer parámetro el *string* con el nombre que se desea ponerle al archivo *.mat* y como segundo parámetro el nombre (en formato *string*) de la variable a almacenar.

Después de todo el proceso descrito, cada base de datos viene identificada como mínimo por su vector de cuenta, su tamaño y su vector de nombres característico, además de por las diferentes vectores de cada subimagen en el caso de haber usado 3 muestras.

Antes de finalizar esta función con la sentencia *guidata(hObject,handles)* para actualizar los valores de las variables *handles* usadas, se cambia el valor de la variable *Actualizar*, donde se pasa a mostrar un mensaje de finalización del proceso de actualización, a través de la función *set*.

### 4.2.2 LoadImage_Callback

Esta función es la que se ejecuta cuando se pulsa el botón *Seleccionar Imagen* de la pestaña *Opciones* y su función es la de elegir la imagen que se desea estudiar y de la cual se quiere obtener su textura. Además de la elección de la imagen de prueba, en esta función se realiza la aplicación del algoritmo a dicha imagen y la comparación de su vector *v* característico con los existentes en la base de datos, obteniendo a su vez el más cercano a través de un clasificador de distancia, y, por tanto, su textura asociada.

Lo primero a realizar en esta función, como se puede observar en el apartado E del capítulo 7.1 del Anexo, es la apertura de la ventana donde se encuentran las imágenes de prueba entre las que se puede elegir. Esto se realiza con una llamada a la función *uigetfile* con parámetro *'*.jpg'*, apareciendo como resultado una ventana del directorio de trabajo donde se encuentra la aplicación, que en este caso será el mismo que donde se encuentran las imágenes de prueba. Se utiliza el parámetro anterior para que solo aparezcan los archivos con dicha extensión y no los programas con extensión .m de las funciones definidas anteriormente. La llamada a *uigetfile* se ha

realizado dentro de la función *imread*, para que, nada más se seleccione una imagen, ésta sea guardada dentro de la variable *im* definida en ese mismo instante.

A continuación, se aplica el algoritmo correspondiente (el que indique la variable *Method*) a la imagen seleccionada y almacenada en *im*. Dentro de la sentencia *if* que indica qué algoritmo se va a aplicar, también se definen la variable *lim_x*, de valor el tamaño del vector de repeticiones esperado (256 ó 58) que será utilizada para la representación visual del histograma; y el vector *x*, que definirá el eje de abscisas del histograma y cuyos valores serán los correspondientes a los posibles dependiendo del algoritmo utilizado (de 0 a 255 ó de 0 a 57).

Una vez se ha llegado a este punto, se han obtenido tanto la base de datos como la imagen a estudiar, por lo que a continuación se debe realizar la comparación entre ambos campos y obtener la relación de semejanza. Antes, se ha de cargar la base de datos correspondiente que viene indicada por los valores de las variables *Method* y *Samples*. Aquí se vuelve a utilizar la sentencia *if* para la determinación de la combinación de variables usada, como ya se vio en el apartado anterior, y dentro de cada una de ellas se obtiene la base de datos con la que se va a comparar, utilizando la función *load*, que carga dentro de una estructura el valore almacenado en el archivo *.mat* que se le indique. Dentro de esta estructura, el nombre del número, vector o matriz obtenido, vendrá dado por el nombre de la variable que se guardó en su momento con la función *save*. Las estructuras utilizadas para el almacenaje de estos datos vienen dadas por *datos* para las matrices de los vectores de cuenta, *Length* para el número de archivos que posee la base de datos y *Tags* para el vector de nombres identificativos de cada imagen. Además, se utilizan las estructuras *datos1*, *datos2* y *datos3* para las matrices de vectores de cuenta en el caso del uso de 3 muestras.

Una vez cargada la base de datos que se va a usar para la comparación, se ha de comparar el vector de cuenta obtenido de la imagen a estudio con los vectores de la base de datos. Para ello se ha hecho uso de un clasificador de distancia básico, representado por la función *pdist2*, que realiza un cálculo de distancia entre un par de observaciones, representadas en este caso por los vectores *v* de la base de datos y de la imagen de prueba. Si no se especifica nada dentro de la llamada a esta función, la distancia calculada será la euclídea o del espacio euclídeo, la cual consiste en el cáculo de la norma de la resta elemento a elemento entre ambos pares de vectores. En este caso se ha hecho uso de un parámetro extra, representado por la variable *Classifier* de tipo *string*, que indica el tipo de cálculo de distancia entre ambos vectores, pudiendo ser euclídea como en el caso por defecto, *City Block*, que consiste en el cálculo de la suma de los valores absolutos resultantes de las restas elemento a elemento entre dos vectores, o de *Minkowski*, cuyo resultado es el valor al aplicar la ecuación de la Figura 4-1, donde *p* es un valor a elegir, *n* es el número total de datos de los vectores, y *x* e *y* son los dos vectores a comparar. Si se observa detenidamente, los tres tipos de cálculo anteriores son resultado de aplicar la misma fórmula, siendo el caso euclídeo cuando *p* es igual a 2 y el caso de *City Block* cuando *p* es igual a 1. Para tener un clasificador extra distinto, se utiliza el valor de *p* igual a 3 para definir el clasificador *Minkowski*, que es po lo que se ha de diferenciar entre su uso o el de los otros dos clasificadores, ya que se ha de estipular el valor de *p* a través de un cuarto parámetro, que consistirá en un 3. Antes de calcular la distancia entre los vectores de repetición de valores LBP, se ha de inicializar una variable *suma* a un valor inalcanzable por la operación que realiza *pdist2*, el cual se corresponde con 100, ya que sería el valor máximo entre una imagen cualquiera y una en la que no aparece ningún valor LBP, caso imposible en esta aplicación, pero que se considera como el peor posible, y es por esto que la variable *suma* se inicializa a un valor de 1000. Además de la variable *suma*, se inicializa la variable *Database_Length*, correspondiente al tamaño de la base de datos y obtenido su valor a través de *Length.Tam*, y la variable *Database_Tags*, correspondiente al vector de nombres de las imágenes de la base de datos y obtenido su valor a través de *Tags.Names*.

$$d = \sqrt[p]{\sum_{j=1}^{n} |x_j - y_j|^p}$$

Figura 4-1. Ecuación para el cálculo de la distancia Minkowski

Acto seguido se crea un bucle *for* que recorre todos los posibles enteros entre 1 y el valor de la variable *Database_Length*, el cual se corresponde con la cantidad total de imágnes de la base de datos, por lo que este bucle se realiza para obtener el vector *v* de cada una de las imágenes prototipo para poder realizar la comparación pertinente. Dentro del bucle hay que diferenciar entre el número de muestras de la imagen a través de la variable *Class*, haciendo que, en caso de que solo exista una muestra, la obtención de la distancia entre vectores de repetición se realice entre el obtenido de la imagen de prueba, almacenado en esta función como *v*, y el valor del vector correspondiente a la imagen de la base de datos que se está observando en el momento, dado por la variable *datos.vector(k,:)*, donde *k* es la variable incrementada del bucle *for*. En caso de que sean tres las muestras obtenidas de la imagen, se ha de obtener el resultado del cálculo de la distancia entre los vectores de cada una de las plantillas, almacenados

como ya se había visto en las variables *datos1.vector1*, *datos2.vector2* y *datos3.vector3*. De los tres resultados obtenidos se escoge el menor de ellos como representación del parecido a dicha textura. Una vez obtenido el valor de distancia, almacenado en la variable *suma_aux*, se comprueba si éste es menor que el último observado, que, en caso de ser la primera imagen de la base de datos, siempre lo será. Esto se realiza con una sentencia *if* añadiendo la comprobación de si *suma_aux* es menor que *suma*, inicializada a 1000. Si esto es cierto, el valor de *suma_aux* se pasa a *suma* para que se actualice el nuevo umbral mínimo que se ha de mejorar para obtener un mayor parecido con otra textura. Además, se guarda en la variable *res* el valor del entero *k*, ya que éste identifica inequívocamente a la textura asociada.

Una vez realizado en bucle *for* por completo, se ha obtenido en la variable *res* la identificación de la textura más parecida a la que posee la imagen de prueba, lo cual se interpreta como el resultado final de la identificación de la textura, por lo que ahora solo queda representar histogramas e imágenes de valores LBP, así como presentar en la GUI la textura final asociada a la imagen de prueba. Todo esto se realiza a continuación, como se puede observar en el código del Anexo, y se representa en la pestaña *Resultados* de la GUI.

Lo primero que se hace es mostar la imagen de prueba, para lo cual se ha de habilitar el cuadro de imagen identificado por la variable de la GUI llamada *axes1*. Esto se consigue llamando a las función *axes* e introduciendo como parámetro de la misma el marco que se desea modificar. Una vez llamada esta función, los siguientes pasos que se realicen contarán exclusivamente para el marco indicado, funcionalidad similar a la utilización de *figure* en el entorno MATLAB. Una vez habilitado este primer marco, se muestra la imagen con *imshow* y se le establece un título con la llamada *title*. La siguiente imagen que se representa es la de valores LBP/ULBP/RLBP en el marco dado por *axes2*, volviendo a utilizar la función *imshow* esta vez con la matriz *LBP* obtenida al realizar el algoritmo especificado sobre la imagen de prueba. El título asociado a este marco se decide según una sentencia *if/elseif* que determina el valor de la variable *Method* y, por ende, el algoritmo utilizado.

A continuación, se pasa a la representación de los histogramas de la imagen de prueba y de la imagen prototipo de la base de datos asociada, en los marcos *axes3* y *axes4*, respectivamente. La representación de histogramas en MATLAB se puede realizar de diferentes formas y en esta aplicación se ha optado por el uso de la función *bar*, que representa un gráfico de barras a través de los parámetros que se les entrega, donde están los valores del eje de abscisas y lo del eje de ordenadas, que son, respectivamente, el vector *x* definido anteriormente y el vector *v* obtenido del análisis de la imagen de prueba, en el caso del *axes3*, y por *x* y *datos.vector(res,:)* en el caso de *axes4*. A ambos histogramas se les modifican los límites superior e inferior del eje de abscisas, ensanchándolo en una unidad por cada límite, para que el histograma se observe mejor. Es aquí donde se hace uso de la variable *lim_x* definida al principio de la función. También a ambos histogramas se les añade un título y *labels* o etiquetas en ambos ejes, para tener cada magnitud definida.

Para terminar, se actualiza la variable de tipo *string* denominada *Resultado*, hasta ahora vacía, la cual imprimirá un mensaje con la textura final asociada, haciendo uso de la última variable que faltaba y del identificador *res*, tal que *Database_Tags(res)* dará el nombre de la imagen de la base de datos la cual ha sido elegida como mejor aproximación a la imagen de prueba. Se observa que no se ha concluido con una sentencia *guidata* ya que no se ha modificado el valor de ninguna variable de la estructura *handles*, solo se ha observado el valor de éstas.

### 4.2.3 tabOpciones_Callback y tabResultados_Callback

Estas dos funciones se explican de manera conjunta por que van relacionadas la una con la otra. Cada una de ellas se activa cuando se pulsa el botón de pestaña asociado, es decir, los botones situados en la parte superior izquierda de la interfaz. Estas funciones cumplen el objetivo de conseguir una funcionalidad similar al uso de pestañas tales como en el editor de MATLAB, y para ello lo que se ha hecho es construir diferentes paneles o ventanas superpuestas con los objetos gráficos que se desean presentar en cada una de ellas. El objetivo de cada función es hacer visible la pestaña con su nombre y hacer lo contrario con la otra.

Una vez creados los paneles e introducidos los objetos gráficos en ellos, se puede hacer uso de una propiedad de los mismos: la visibilidad. Este parámetro es intrínseco a cada panel y su puesta a *on* conlleva una visualización completa del panel y una puesta a *off* lo contrario. Es por ello que, por ejemplo, cuando se pulsa el botón *Opciones*, se ha de poner el parámetro *visible* del panel de opciones, identificado por la variable *panel_options* de la estructura *handles*, a *on* y el del panel de resultados, identificado por *panel_resul*, a *off*. En

caso de pulsar el botón *Resultados* se asignarán los valores de los parámetros*visible* de forma contraria. Nunca se debe tener ambos paneles visibles a la vez, pues el programa puede dar lugar a fallos de funcionalidad, y se debe actualizar la estructura *handles* al final de la función.

Esta forma de integración de las pestañas es muy útil, ya que la inclusión de pestañas extras es tán sencilla como incluir una línea de comandos en cada función de llamada de pulsación de los correspondientes botones y asignar su valor de parámetro *visible* correspondiente en cada caso.

### 4.2.4 muestras_SelectionChangedFcn, metodo_SelectionChangedFcn y clasificador_SelectionChangedFcn

Estas funciones también se explican de manera conjunta porque, aunque no se encuentren relacionadas entre sí, la estructura de programación existente en todas ellas resulta ser la misma. Son las encargadas de obtener los datos acerca del número de muestras obtenidos, del algoritmo utilizado y del tipo de clasificador de distancia usado, lo cual se realiza en la interfaz a través de grupos de *Radio Buttons*, que se encontran dentro de paneles en los que solo se podrá elegir uno de ellos a la vez. Una vez introducidos tantos *Radio Buttons* como opciones se deseen dentro de un mismo panel, se observará que el panel pasa a ser un denominado *ButtonGroup*, y se creará una función de respuesta frente al cambio del botón seleccionado dentro de él, que será como las que se están estudiando en este instante.

Pues bien, dentro de estas funciones de respuesta, se utiliza la función *get* con parámetros *(hObject,'String')* para guardar en la variable *opt* el texto asociado al botón que se encuentra pulsado en dicho instante. Es importante entonces que el texto asociado a cada *Radio Button* sea distinto de los demás, al menos en un mismo grupo. Son funciones que se activan por interrupción, como las anteriores, siendo en este caso, en vez de la pulsación de un botón, el cambio en la selección de uno de ellos lo que la genera. Una vez se ha obtenido el *string* de la selección de dicho instante, basta con realizar una sentencia *switch-case* en la que se plantean todos los posibles resultados de *opt* y, dependiendo de este, se ajustará el valor de la señal *Samples,* en el caso del número de muestras a obtener con el algoritmo, o el de la señal *Method*, para la elección del algoritmo en sí. El valor de la señal *Classifier* se obtiene directamente de la función *get*, ya que ésta devuelve un tipo *string* que contiene el nombre del clasificador en minúculas (uso de la función *lower*) que se usa en la función *pdist2*.

Finalmente, como en la mayoría de funciones, se actualiza la estructura *handles* a través de la función *guidata* para poder utilizar los valores de *Samples, Method* y *Classifier* en las funciones principales.

# 5 GUÍA DE LA APLICACIÓN

En este apartado se encuentra la guía de usuario, en la cual se explican los pasos a seguir para quien desee hacer uso correcto de la aplicación. Además, se explicará de forma detenida el proceso de obtención de la interfaz gráfica, es decir, qué elemento se escogieron y sus funcionalidades.

## 5.1 Guía de Usuario

En este apartado se describirán todas las funcionalidades y el orden en el que estas han de realizarse para obtener un funcionamiento correcto de la aplicación.

### 5.1.1 Descarga y preparación

Una vez se ha descargado el fichero *.zip* que contiene la GUI, se ha de descomprimir el mismo en el directorio donde el usuario estime pertinente. Una vez realizado este paso, se pasa a tener todos los archivos y carpetas necesarios para la ejecución de la aplicación en un mismo directorio.

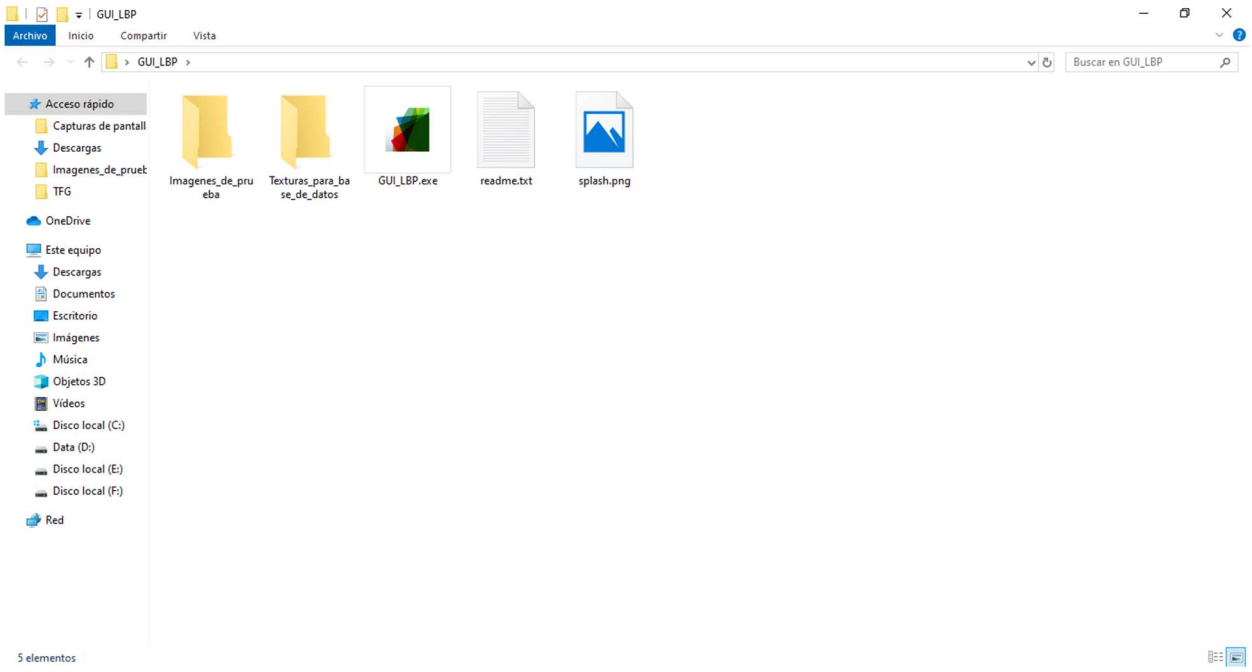



Figura 5-1. Carpeta de archivos descomprimidos

Entre estos archivos se encuentran la carpeta *Texturas_para_base_de_datos*, que contiene las imágenes por defecto que se utilizan como base de datos de la aplicación; la carpeta *Imagenes_de_prueba*, donde se almacenan las imágenes que van a ser analizadas en la aplicación; el ejecutable *GUI_LBP*, el cual contiene a la aplicación en sí; un fichero de texto *readme*, que contiene indicaciones sobre el software necesario para la correcta ejecución de la aplicación; y una imagen con nombre *splash* en formato *.png* que se utiliza como imagen de presentación durante la carga de la aplicación. De estos, únicamente se puede borrar el fichero de texto sin cambiar el funcionamiento de la aplicación, puesto que los tres primeros son indispensables para su desarrollo y la imagen restante, más opcional, es necesaria para la espera durante la carga de la GUI.

Una vez llegado a este punto, se puede pasar directamente a utilizar la aplicación, aunque, normalmente, ya que el uso de la misma está caracterizado por el que le vaya a dar el usuario, se modifica el contenido de ambas carpetas, personalizando las imágenes que comprenden la base de datos y las que comprenden las imágenes de prueba.

En cuanto a la base de datos, el usuario es libre de introducir tantas imágenes como quiera, en formato *.jpg* preferiblemente, teniendo en cuenta que éstas van a ser las que definan cada textura, identificando cada una de ellas por el nombre de la misma imagen que la contiene. Como se puede observar, la carpeta contiene por defecto 16 imágenes que definen un total de 15 texturas, ya que la textura *grass* (*hierba*, en inglés) posee dos imágenes que la definen. Este es un paso que se recomienda hacer, puesto que a mayor número de imágenes diferentes que definan a una misma textura, más posibilidades hay de que la detección de la misma sea correcta, ya que el algoritmo utilizado (*LBP*) para la obtención de la misma no es perfecto.

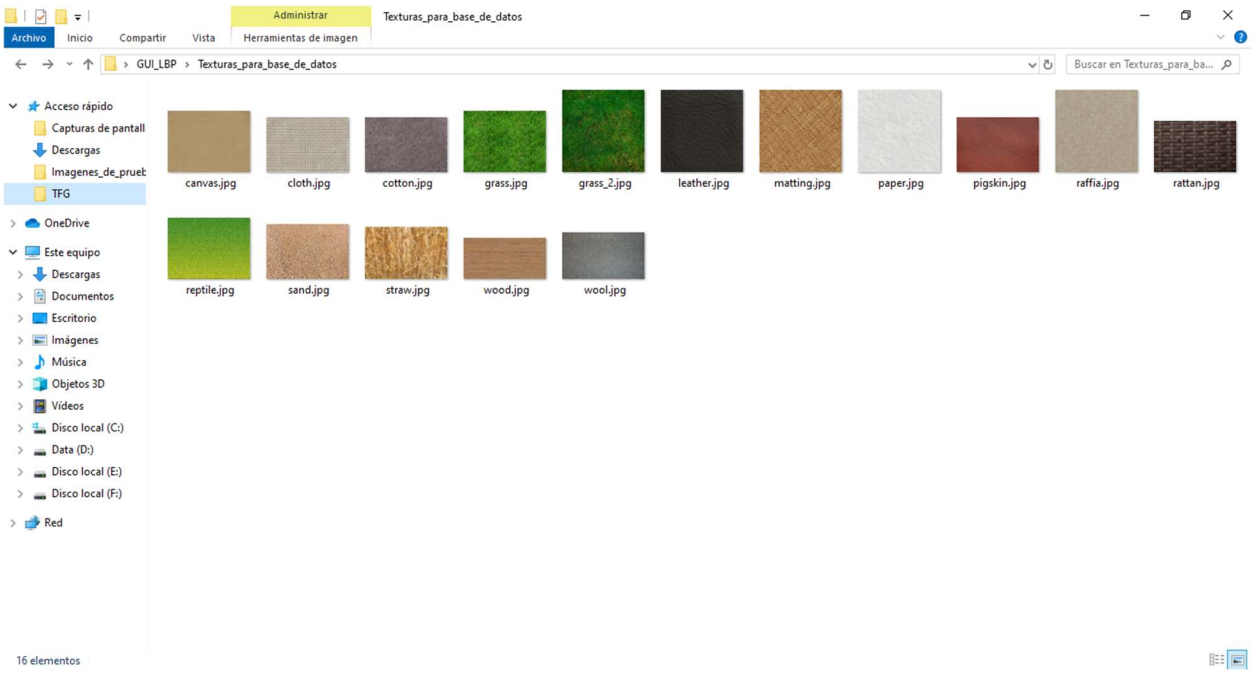



Figura 5-2. Contenido de la carpeta de base de datos

La carpeta de imágenes de prueba contiene 7 imágenes que se han utilizado para el testeo de la aplicación, y por eso se encuentran por defecto. Los nombres de estas imágenes son indiferentes para el programa, puesto que no los utiliza para nada, por lo que el usuario es libre de identifcarlos como desee, aunque se recomienda utilizar una identificación sencilla, como la de nombrar cada imagen como la textura que se cree que se encuentra en la misma.

Como último paso antes de comenzar con el uso de la aplicación, se debe comprobar que el software indicado en el fichero de texto *readme* se encuentra instalado en el dispositivo en el que se esté utilizando la aplicación, puesto que, de lo contrario, no funcionará de forma correcta. Dentro de los requisitos indicados en dicho fichero se encuentra únicamente que el software de *MATLAB Runtime* se encuentre instalado en la versión indicada, que es en la que se ha diseñado la aplicación, pudiendo utilizar versiones posteriores siempre y cuando no se haya suprimido el servicio de diseño de GUIs en dicha versión.

Si fuese de utilidad para el usuario, siempre se puede crear un acceso directo al ejecutable de la aplicación y disponerlo donde más se desee (siempre y cuando no se cambie la ubicación del ejecutable original) o anclar dicho ejecutable a la barra de tareas para tenerlo siempre disponible.

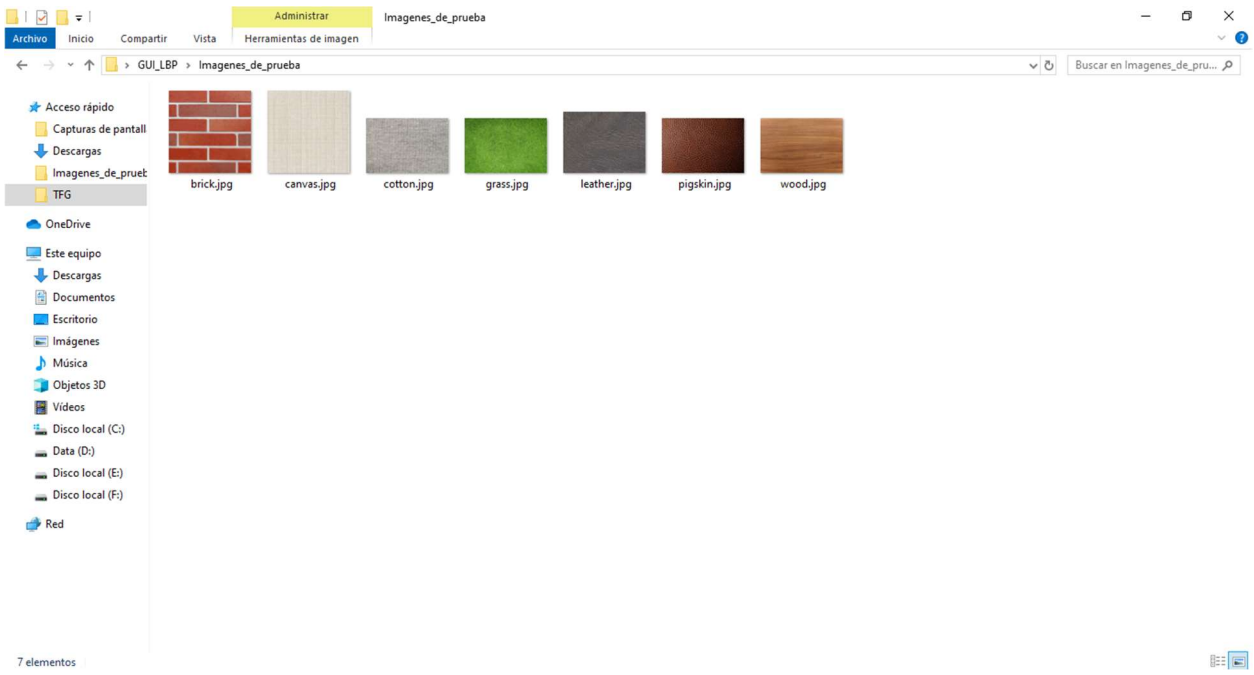



Figura 5-3. Contenido de la carpeta de imágenes de prueba

### 5.1.2 Uso de la aplicación

Ahora se va a pasar a explicar el correcto uso de la aplicación, paso por paso. El primero de ellos es clicar en el ejecutable, lo cual da paso a la carga de la GUI y cuyo proceso se puede validar observando la aparición de la imagen mostrada en la Figura 5-4, donde se muestra el nombre de la aplicación y su eslogan.

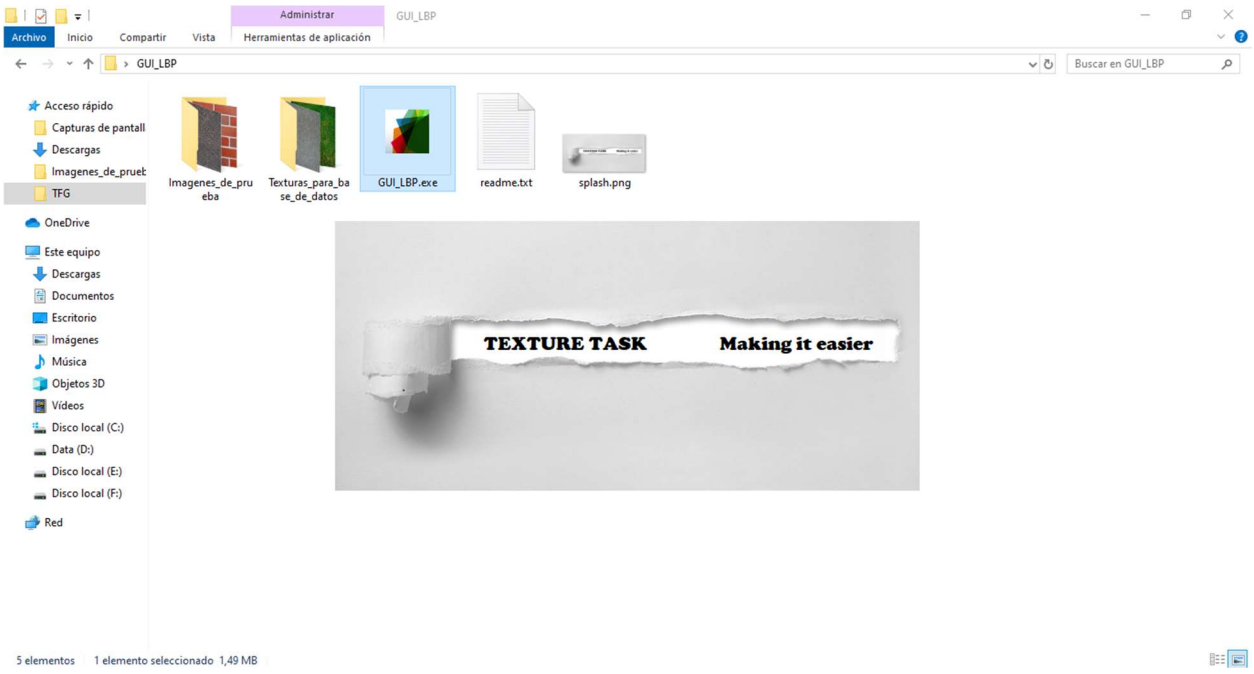



Figura 5-4. Período de carga de la aplicación

Una vez pasado el período de carga de la GUI, aparece ésta en una ventana nueva, con una primera pestaña de nombre *Opciones*, donde se encuentran los primeros pasos a realizar que se describen a continuación.

Lo primero a llevar a cabo, si se está utilizando la aplicación por primera vez, es la actualización de la base de datos, para lo cual se ha de pulsar el botón *Actualizar Database* indicado en rojo en la Figura 5-5. Este es un paso que, si no es la primera vez que se utiliza la aplicación y ya se ha creado una base de datos previamente, no hace falta realizar en usos posteriores de la GUI, a no ser que se haya modificado el contenido de la carpeta de imágenes para base de datos, puesto que la información sobre la base de datos previa queda guardada en el directorio donde se encuentre el ejecutable. Se ha de tener en cuenta que la base de datos se crea en función a ciertos parámetros, los cuales se pueden encontrar en la parte izquierda de esta misma pestaña. De estos parámetros, únicamente importan para este aspecto los englobados en los grupos *Seleccionar número de muestras* y *Seleccionar método*, puesto que son los únicos que afectan al resultado obtenido tras el análisis de la imagen. El resto de parámetros, englobados en el grupo *Seleccionar clasificador*, son útiles a la hora de comparar imágenes, por lo que se tiene en cuenta a la hora de realizar dicho paso.

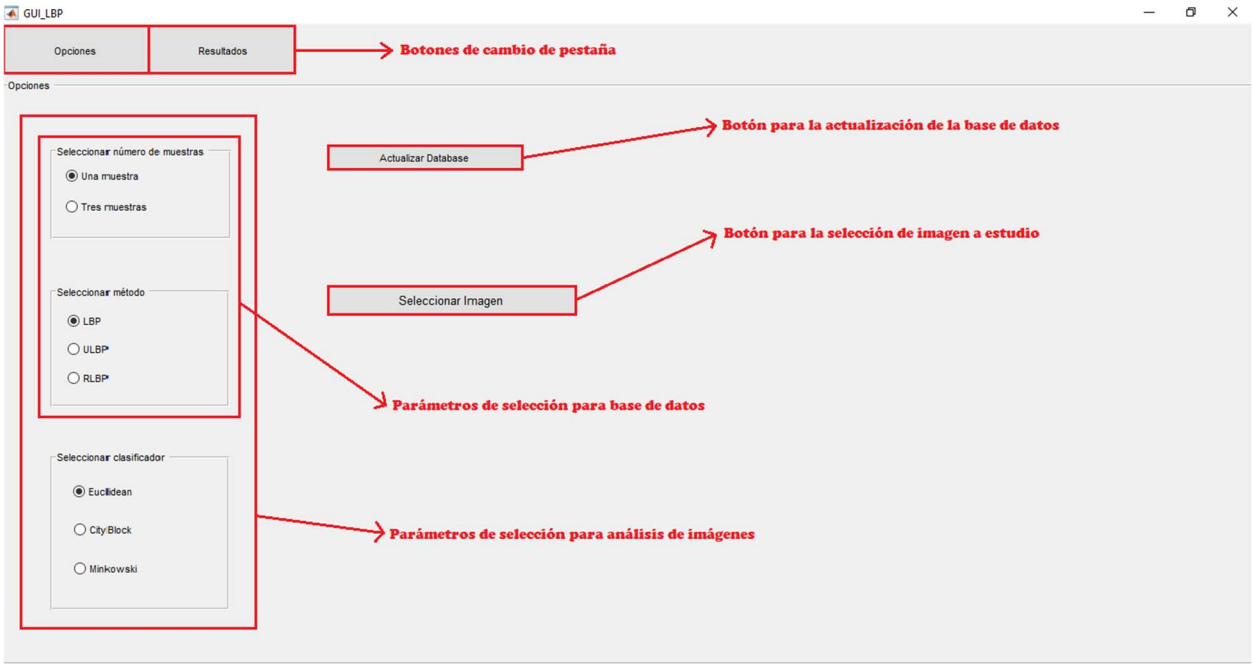



Figura 5-5. Contenido de la pesta *Opciones*

Volviendo a la creación de la base de datos, se aconseja realizar tantas actualizaciones de bases de datos como combinaciones posibles entre los parámetros de los grupos comentados existen (un total de 6 combinaciones posibles). Esto se hace así puesto que, a la hora de analizar las imágenes para identificar su textura, existen diferentes metodologías a llevar a cabo, y dependiendo de cada caso en concreto, una de ellas ofrece mejores resultados que las otras, o viceversa, además de que la aplicación no obtiene resultado si la imagen a estudio se analiza con una combinación de la cual no se ha obtenido base de datos. Teniendo entonces tantas bases de datos como metodologías contempladas, se pueden comparar los resultados obtenidas de todas ellas y obtener un resultado más fiable, por ejemplo, identificando la textura que más se repite entre estas distintas metodologías.

Dentro del grupo *Seleccionar número de muestras* se puede elegir entre el estudio de la imagen original (opción por defecto) o la de generar 3 subimágenes aleatorias obtenidas a partir de los píxeles de la original, de f tal forma que se obtienen 3 prototipos de textura con los que comparar. Esto último es de preferible uso en situaciones en las que la textura a identificar se encuentra definida por una gran cantidad de patrones distintos a lo largo de toda la imagen, ya que, de lo contrario, al crear las 3 subimágenes aleatorias, se puede deteriorar el patrón que define a la textura y perder su identidad. Dentro del grupo *Seleccionar método* se puede elegir entre 3 tipos diferentes de algoritmos utilizados para analizar la imagen, donde el *LBP* es el algoritmo base por defecto y es el más genérico, el *ULBP* es una simplificación más eficiente en el ámbito computacional y con casi la misma tasa de existo que el original, y el *RLBP* es una modificación del primero la cual es útil para casos donde la textura sea variante frente a rotaciones, es decir, que la textura identificada depende de la rotación de la imagen.

Una vez seleccionados los parámetros que se deseen, se pulsa el botón *Actualizar Database* y acto seguido aparece un mensaje de texto justo debajo de éste indicando el número de imágenes que componen la futura base de datos que quedan por analizar. Cuando este proceso acaba, el texto cambia indicando su finalización. Comentar que este proceso genera, en el directorio donde se encuentre el ejecutable, 3 archivos (en el caso de 1 muestra) ó 6 archivos (en el caso de 3 muestras), que no se deben modificar o borrar, puesto que, si se desea crear una base de datos con otro contenido, basta con volver a realizar el proceso descrito anteriormente y los datos se sobrescribirán en dichos archivos.

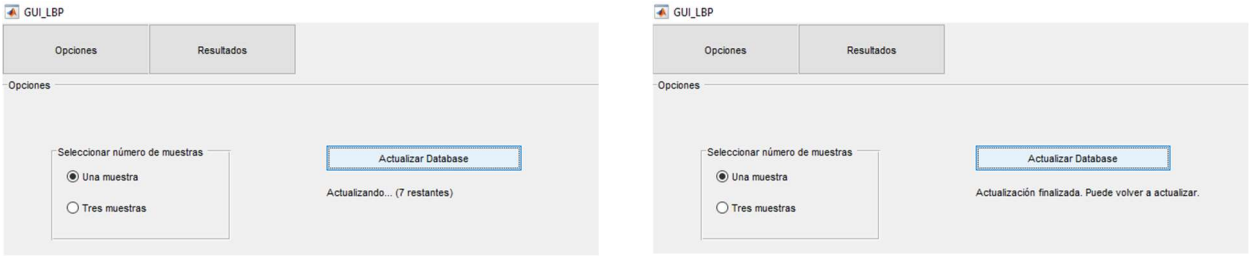



Figura 5-6. Mensajes informativos sobre el estado de actulización

El siguiente paso a realizar, una vez se han creado todas las bases de datos o únicamente las que se deseen, es el de selección de imagen a estudio. Esto se consigue haciendo clic en el botón *Seleccionar Imagen* que se encuentra justo debajo del botón de actualización de base de datos. Antes de elegir dicha imagen cuya textura se quiere identificar, hay que tener en cuentalos parámetros indicados para la obtención de la misma, donde entran en juego los mismos que para la creación de las bases de datos en conjunto con los parámetros contenidos en el grupo *Seleccionar clasificador*. Estos últimos se utilizan en el proceso de comparación entre la imagen a estudio y la base de datos, midiendo la diferencia entre los resultados obtenidos de cada imagen a través del algoritmo indicado. Existen 3 tipos distintos de clasificadores y, aunque todos se basan en el mismo concepto, se diferencian en la presicisión con la que calculán dicha semejanza, obteniendo una mejor prestación del clasificador *Minkoski*, seguido por el *Euclidean* y terminando por el *CityBlock*. Se ha dispuesto de esta diversidad de clasificadores debido a que una mayor prestación conlleva un mayor coste computacional, por lo que, según la situación, puede no ser necesaria tanta precisión y sí más velocidad de análisis.

Una vez elegidos los parámetros a usar, se pulsa el botón *Seleccionar Imagen*, lo que hace que se abra una ventana emergente con el contenido de la carpeta *Imágenes_de_prueba* ya comentada, por lo que basta con seleccionar una de ellas haciendo doble clic o un clic y pulsando en el botón *Abrir*. Es importante no dirigirse a otra carpeta/directorio a la hora de seleccionar imagen, puesto que en dicho caso la aplicación genera un fallo y

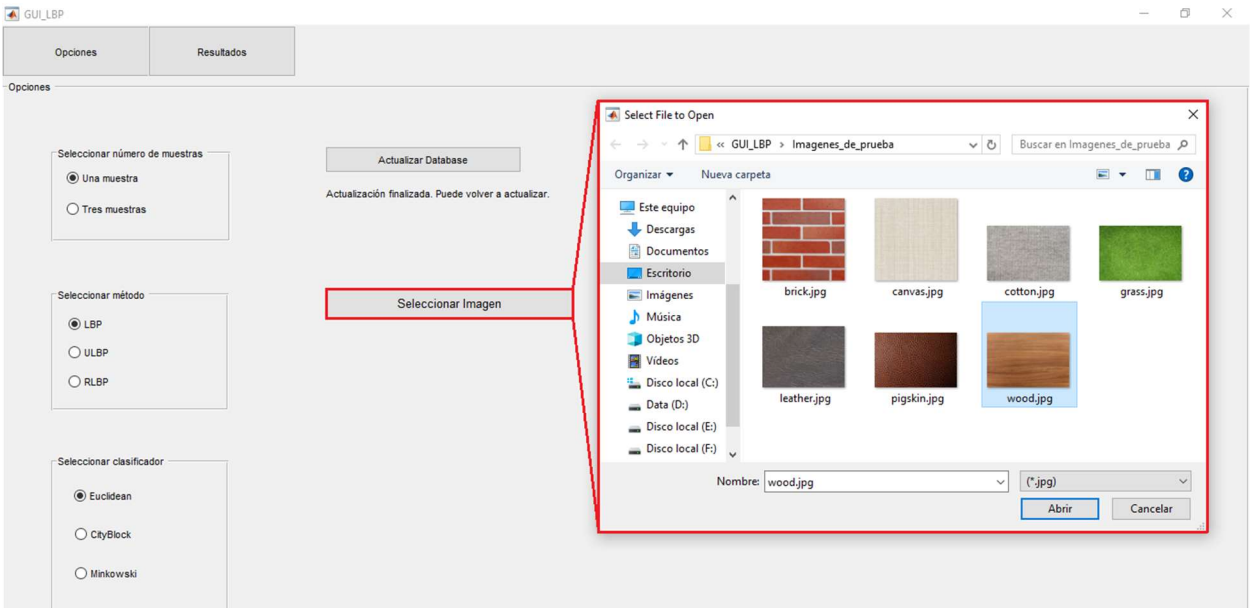



Figura 5-7. Ventana de selección de imagen a estudio

no obtiene resultado alguno.

Una vez se elige la imagen, el proceso de detección de textura comienza automáticamente, pudiendo pulsar seguidamente el botón de cambio de pestaña *Resultados* para visualizar los resultados obtenidos. La aplicación está realizada de tal forma que, si el proceso de detección no ha finalizado y se pulsa cualquier botón, éste último no es capaz de realizar la función a la que está sujeto, por lo que puede haber casos en los que se quiera cambiar de pestaña y se deba esperar algunos segundos hasta que la aplicación lleve a cabo dicho cambio.

El formato de la pestaña *Resultados* es el que se puede observar en la Figura 5-8, donde se pueden encontrar 4 elementos gráficos con sus propias descripciones y un mensaje de texto en la esquina superior izquierda indicando la textura final asociada. Dentro de los elementos gráficos se puede encontrar la imagen a estudio elegida, así como la imagen de valores *LBP/ULBP/RLBP* (según el método escogido) de dicha imagen, la cual puede presentar, si el patrón es lo suficientemente visible, una manera visual de detección de la textura, como información extra para el usuario. Los otros dos elementos gráficos son los histogramas obtenidos tanto de la imagen a estudio como de la imagen de la base de datos con la que ha sido asociada, siendo esta la ayuda visual más importante, puesto que los histrogramas representan la cantidad de veces (en tanto por ciento) que se repite un mismo valor *LBP/ULBP/RLBP* dentro de una misma imagen, es decir, la forma del histograma identifica los patrones contenidos en la imagen y con qué frecuencia aparecen.

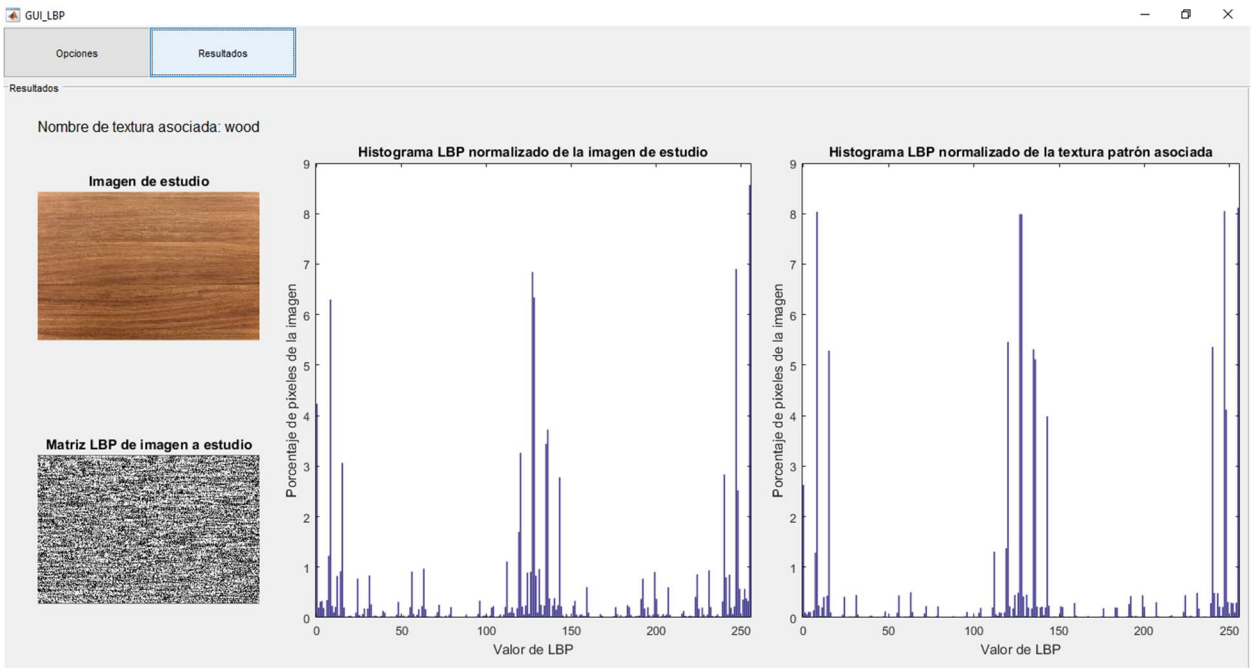



Figura 5-8. Visualización de la pestaña *Resultados* con identificación correcta de textura

En la figura anterior, además del resultado de texto obtenido que indica el nombre la textura, o en su defecto el nombre con el que se identificó a la textura en la carpeta de imágenes de base de datos, se puede observar como los histogramas poseen estructuras muy parecidas, ya que en ambas imágenes se repiten los mismos patrones, como se puede observar en la Figura 5-9. Este resultado, que además es correcto, se ha conseguido con parámetros de muestra única, algoritmo *LBP* y clasificador *euclidean*, mientras que, si se utilizan los parámetros de 3 muestras, algoritmo *ULBP* y clasificador *CityBlock*, se obtiene como textura asociada el *reptile*, cuyo histograma es bastante distinto, como se puede observar en la Figura 5-10. Este resultado se debe a lo ya comentado acerca del método de las 3 muestras, ya que, observando el histograma de la Figura 5-8 que ha sido obtenido a través del algoritmo original (*LBP*), se puede observar que la textura *wood* se define por patrones concretos que se repiten mucho a lo largo de la imagen, característica que se pierde debido a la aleatoriedad con la que se obtienen esas 3 subimágenes.

Figura 5-9. Imagen de prueba (izquierda) frente a imagen de base de datos asociada a ella (derecha)

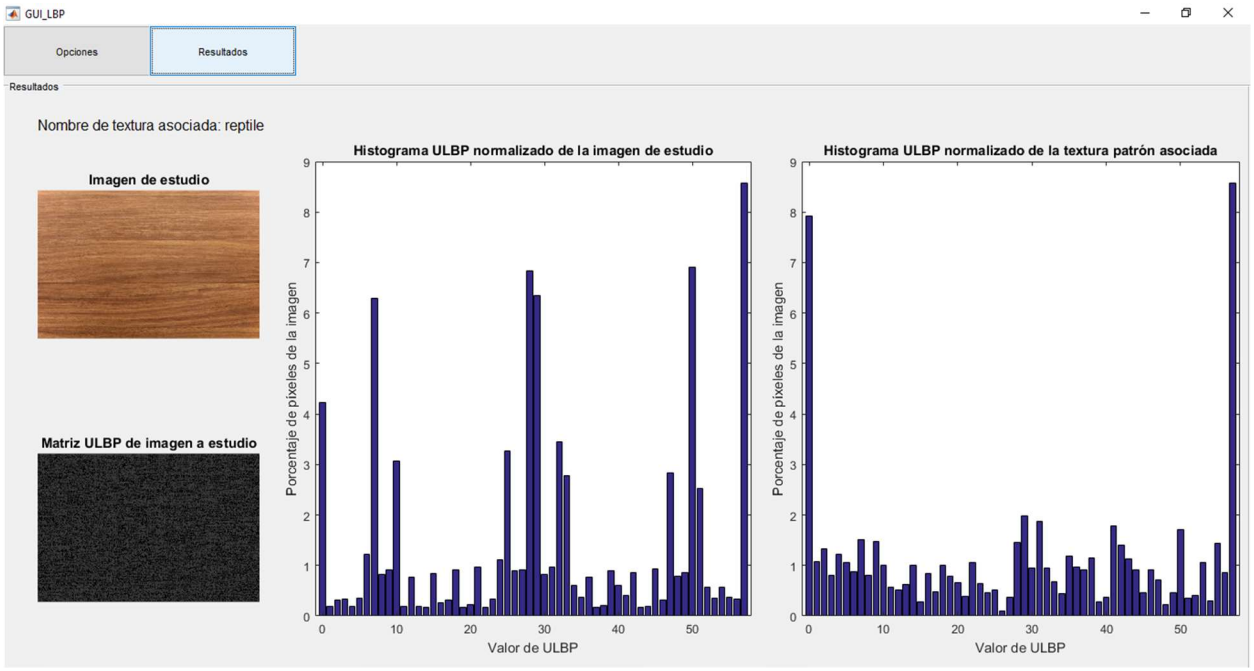



Figura 5-10. Resultado erróneo de identificación de texturas

## 5.2 Guía de Diseño

En este apartado de la guía de la aplicación se va a explicar el proceso de diseño de la GUI, paso por paso, para una posible réplica de la misma en un futuro.

Lo primero que se ha de realizar para diseñar una interfaz gráfica en MATLAB es utilizar el comando *guide* y seleccionar una plantilla en blanco, como ya se explicó en la guía de programación. Para la introducción de cualquier elemento gráfico en la interfaz se ha de hacer clic en el icono que lo represente en la tabla situada a la izquierda de la ventana. El primer paso que se ha de realizar es la creación de un panel, cuyo icono se encuentra marcado en azul y con una etiqueta con su nombre en la parte izquierda de la venta de edición. Éste comprende una de las pestañas, siendo elegida como la que contiene las *Opciones*, aunque el orden de creación de paneles es indiferente. Una vez situado el panel dentro de la interfaz, se ajusta el mismo al tamaño de la pantalla, excepto por la parte superior, donde se ha de dejar un hueco para la inclusión de los botones de cambio de pantalla, como se observa en el cambio de la Figura 5-11 a la Figura 5-12.

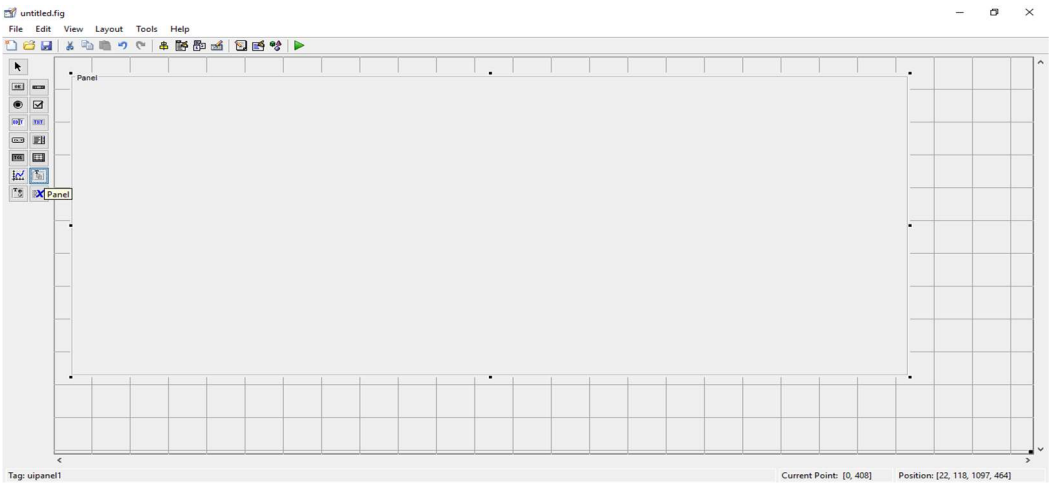



Figura 5-11. Introducción del panel en la interfaz


Figura 5-12. Expansión del panel al tamaño de la ventana

Una vez se ha situado el panel, se ha de hacer doble clic izquierdo sobre el mismo para entrar en la ventana de propiedades del mismo, donde se ha de cambiar el nombre que identifica al panel en la interfaz por *Opciones* en el parámetro *Title*, para identificar la pestaña en la que se encuentra el usuario en el momento, y el valor del parámetro *Tag*, utilizando el identificador *panel_options*, que será el nombre por el que se identificará al panel a nivel de programación. Todo esto se puede observar en la Figura 5-13 que se presenta a continuación.

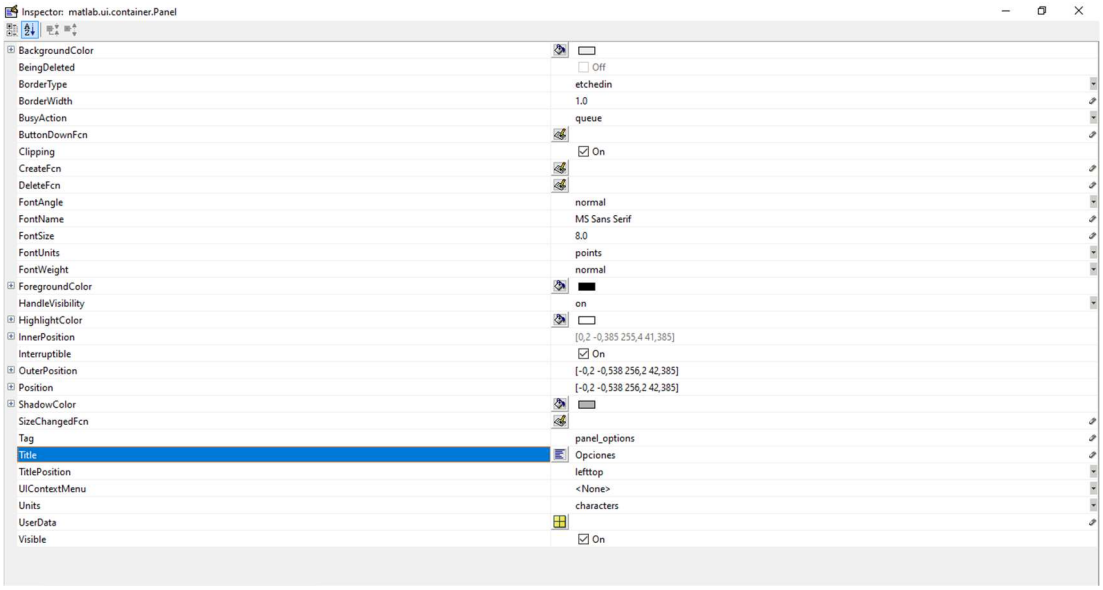



Figura 5-13. Cambio de parámetros en el panel *Opciones*

Lo siguiente a introducir son los botones de cambio de pestaña comentados anteriormente, situándolos en la parte superior izquierda de la ventana, como se observa en la Figura 5-14. Para introducir estos botones se ha de pulsar en el icono de *Push Button* que aparece en esta figura marcado en rojo y se sitúa en la posición estimada y con el tamaño deseado. Acto seguido se han de cambiar los parámetros *Title* y *Tag* de estos botones para que se muestren sus correspondientes identificaciones en la interfaz y se puedan utilizar sus pulsaciones en el programa. Para esto último se debe inicializar la función llamada cuando se pulsan, para lo cual se ha de clicar en el icono que aparece en el parámetro *Callback*, el cual crea la función y la introduce de manera automática en el código principal de la GUI. El resultado de estas operaciones se puede observar en la Figura 5-15.

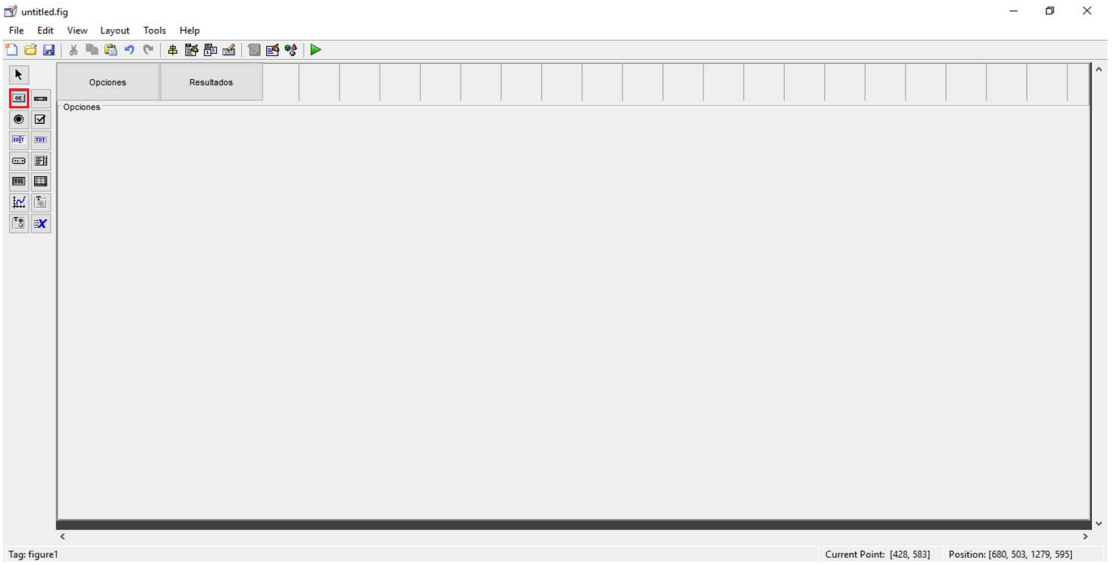



Figura 5-14. Introducción de botones de pestañas

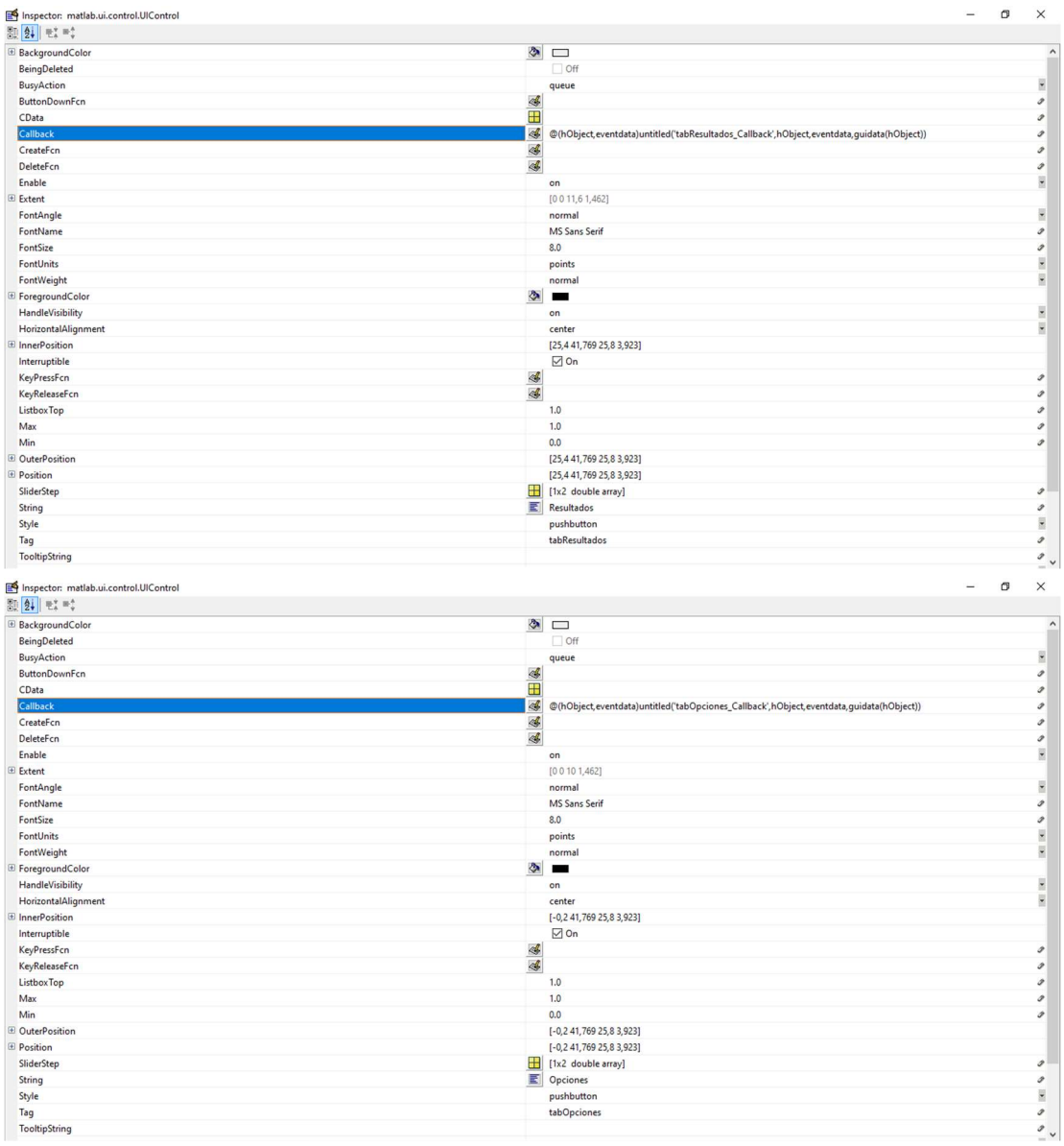



Figura 5-15. Inicialización de parámetros y función de llamada de botones de pestaña

El siguiente paso es el de crear un *Group Button*, el cual sirve para introducir las diferentes opciones en cuanto al análisis de una imagen. El icono de este objeto gráfico se encuentra situado donde se indica en la Figura 5-16, y se ha de cambiar su *Title* y *Tag*, además de inicializar su parámetro *SelectionChangedFcn* como se ha hecho en la situación anterior con el parámetro *Callback*, puesto que es gracias a esta que se puede detectar un cambio de selección en su interior, dada por los *Radio Buttons* que se explican en el siguiente párrafo.

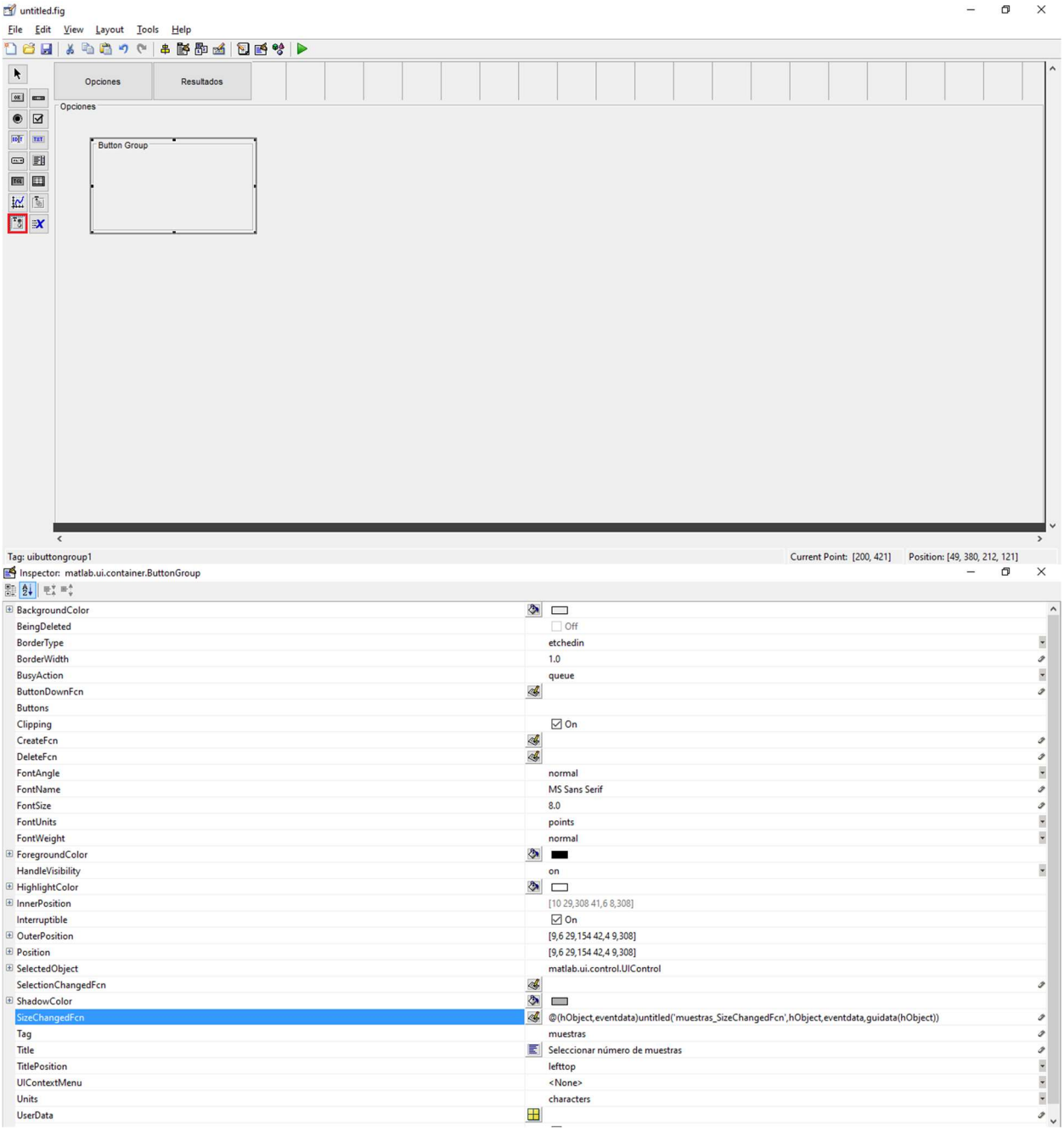



Figura 5-16. Introducción e inicialización de un *Button Group*

Una vez se tiene el *Button Group* se puede pasar a introducir los *Radio Buttons* que permiten seleccionar un cierto parámetro del análisis de la imagen. Los botones de esta índole, cuando se encuentran dentro de un grupo como el anterior, pasan a poder elegirse únicamente de uno en uno, evitando que se puedan tener dos parámetros distintos del mismo grupo en un mismo instante, lo cual ocasionaría problemas. Una vez introducidos estos botones, cuyo icono se encuentra marcado en la Figura 5-17, se hace uso de la utilidad de alineamiento también marcada en la zona superior de la interfaz, donde se encuentran las diferentes utilidades que presta MATLAB para la edición de interfaces gráficas.

Esta utilidad se encarga de alinear dos o más objetos en el eje horizontal o vertical, según se desee, siendo este caso en el que se elige que ambos *Radio Buttons* se encuentren alineados verticalmente por su lado izquierdo, lo cual se consigue con la pulsación del icono indicado en la ventana *Align Objects* de la Figura 5-17.

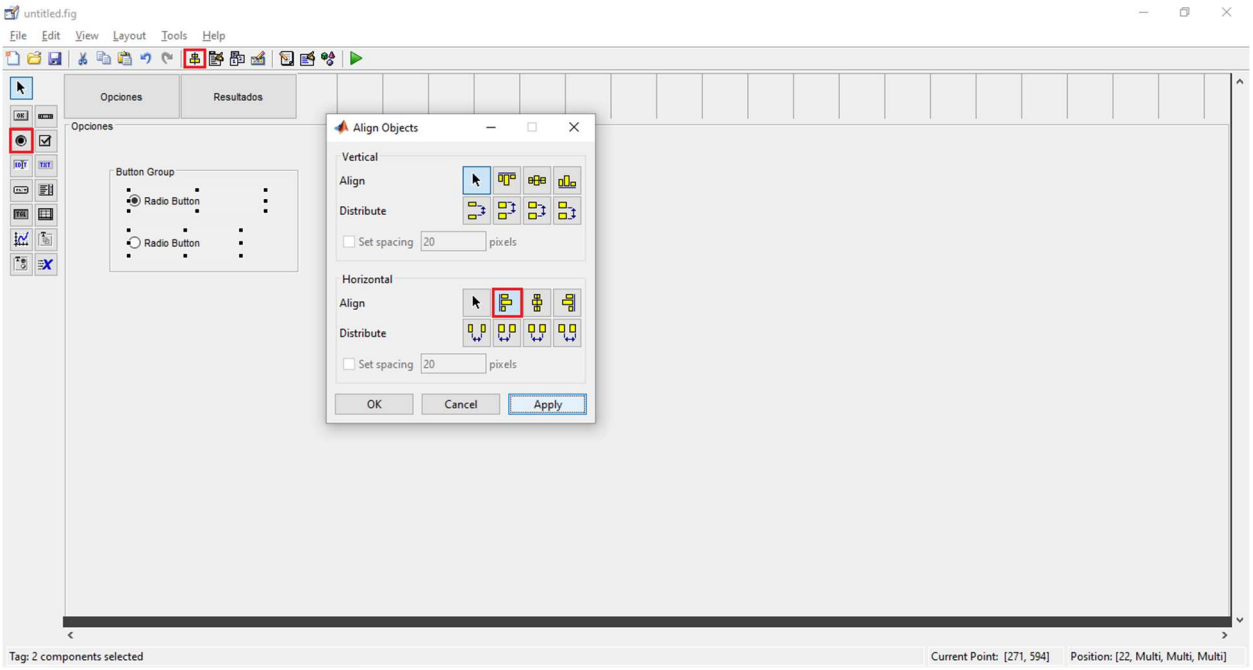



Figura 5-17. Introducción y alineación de *Radio Buttons*

Acto seguido se modifican los nombres de estos botones en el parámetro *String*, de suma importancia, pues es el nombre usado en el código de la GUI para la identificación del botón pulsado. Este proceso se repite dos veces, creando un *Button Group* para la selección del algoritmo y otro para la selección del clasificador que se desean usar. El resultado final una vez introducidos todos se puede observar en la Figura 5-18.

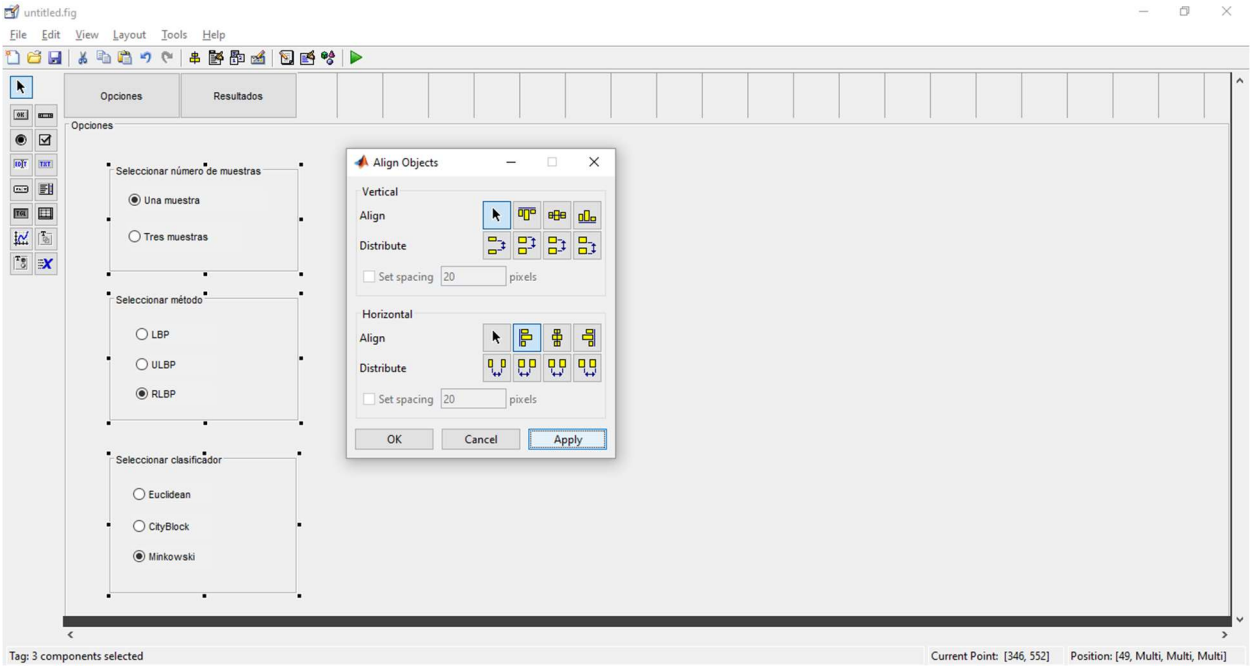



Figura 5-18. Resultado final de la introducción de *Button Groups*

Una vez finalizado este proceso, se van a incluir dos *Push Buttons* dentro del mismo panel, cuyas funcionalidades serán las de actualizar la base de datos y la de cargar una imagen para su análisis. Para dichos botones se deben inicializar los mismos campos *Title* y *Tag* que se inicializaron con los botones de cambio de pestaña, eligiendo *Actualizar Database* y *Actualiza_Base*, respectivamente, para el primero de ellos, y *Seleccionar Imagen* y *LoadImage* para el segundo. En ambos casos se debe inicializar el parámetro *Callback* como ya se hizo anteriormente.

Como último objeto en este panel o pestaña, se incluirá un *Static Text*, ubicado donde indica la marca roja en la Figura 5-19, y cuya funcionalidad es la de avisar de la cantidad de imágenes restantes para terminar la actualización, así como indicar el aviso de terminación de la misma. Se elige en formato estático para que su contenido no se pueda cambiar directamente en la GUI. Como parámetros a modificar en este tipo de objeto se encuentran el *FontSize*, que indica el tamaño de la fuente (en este caso 8), así como se puede modificar la fuente en sí utilizada con el parámetro *FontName*, que en este caso se ha dejado en su valor por defecto; el *HorizontalAlignment*, estipulado en *Left* para que el texto comience en la zona izquierda del cuadro que lo delimita; y el *Tag*, que es el identificador utilizado para modificar su contenido en el programa. Todos estos parámetros y sus correspondientes valores se pueden observar en la figura presente a continuación.

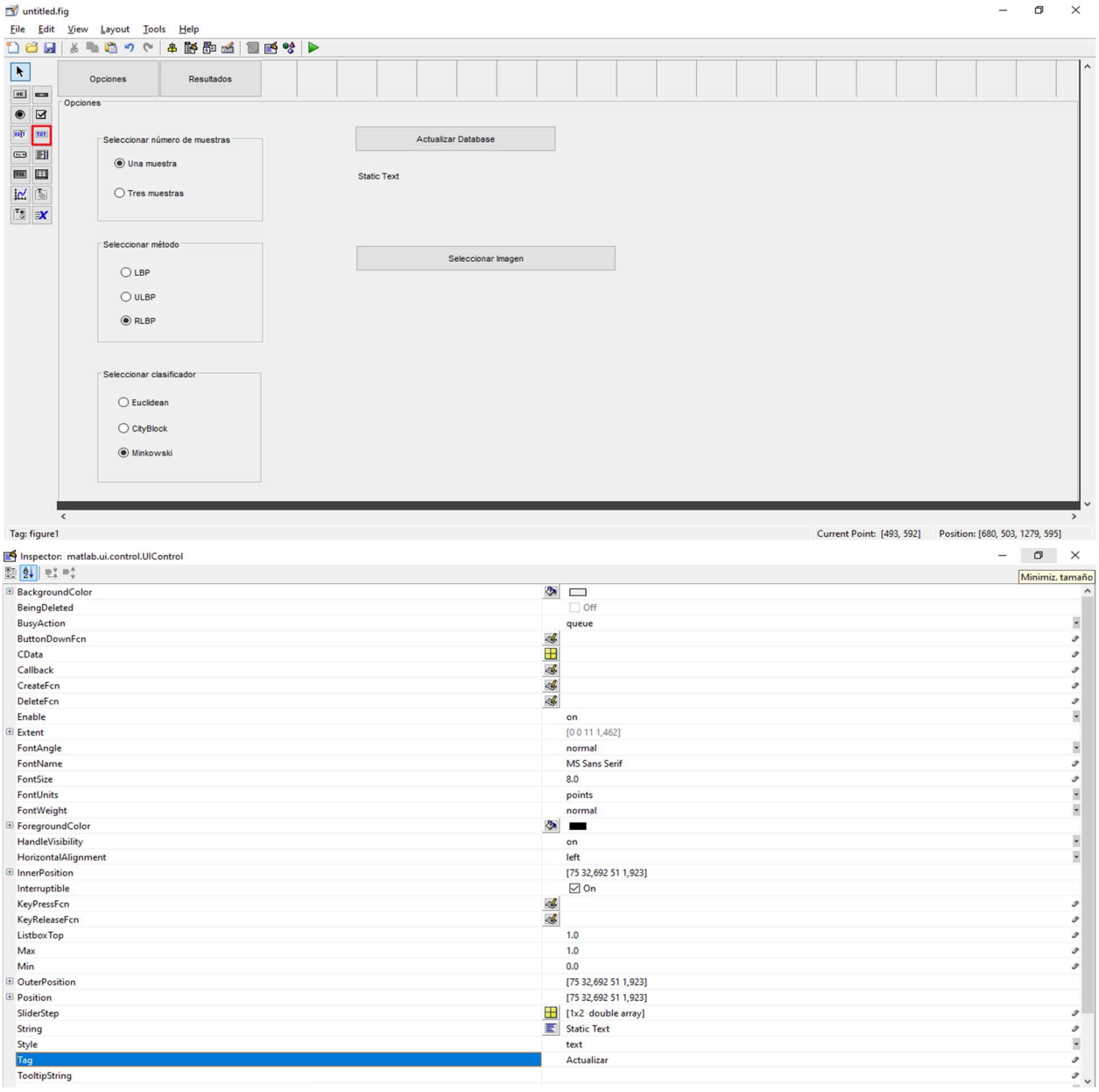



Figura 5-19. Composición final del panel *Opciones* y parámetro del *Static Text*

Una realizado este último paso, se da por concluído el diseño de la pestaña *Opciones* y se pasa a diseñar la segunda pestaña. Esta se compone nuevamente de otro panel, el cual tendrá asociado un *Title*, dado por la palabra *Resultados*, y un *Tag*, dado por el nombre *panel_resul*. Se ha de situar encima del otro panel, con las mismas dimensiones y el mismo posicionamiento, para que se encuentren solapados entre sí, y que de esta forma no se produzca ninguna diferencia en la interfaz a la hora de cambiar de pestaña, es decir, que el único cambio que se visualice sea el del contenido de la ventana de la GUI. Este proceso se ha de realizar con cuidado, puesto que si, a la hora de situar este segundo panel, la ventana de diseño entiende que se está incluyendo al mismo dentro del otro panel, ambas ventanas desaparecen al intentar cambiar de pestaña, pues se encuentran unidas.

Una vez se ha conseguido incluir este nuevo panel dentro de la interfaz, se incluyen un cuadro de texto estático (*Static Text*), como el del panel anterior, y cuatro *Axes*, cuya finalidad es la de representación de imágenes o histogramas. El objeto gráfico *Axes* se puede conseguir pulsando en el icono marcado en la Figura 5-20 y creando un cuadro con el tamaño deseado dentro del panel. Este objeto no necesita ninguna modificación en cuanto a sus parámetros, ya que el único de ellos que se utiliza es el *Tag*, el cual se identifica por defecto como *axes1 axes2*, *axes3* o *axes4*, según el utilizado, y se ha decidido dejarlo así para su posterior uso en el programa. En el caso del *Static Text*, sí se cambian sus parámetros como en el anterior caso, designando un tamaño de fuente de 12 en este caso, una alineación horizontal en el lado izquierdo y un identificador (*Tag*) dado por el nombre *Resultado*. Además, se ha vuelto a hacer uso de la utilidad de alineamiento para posicionar estos *axes* de manera correcta.

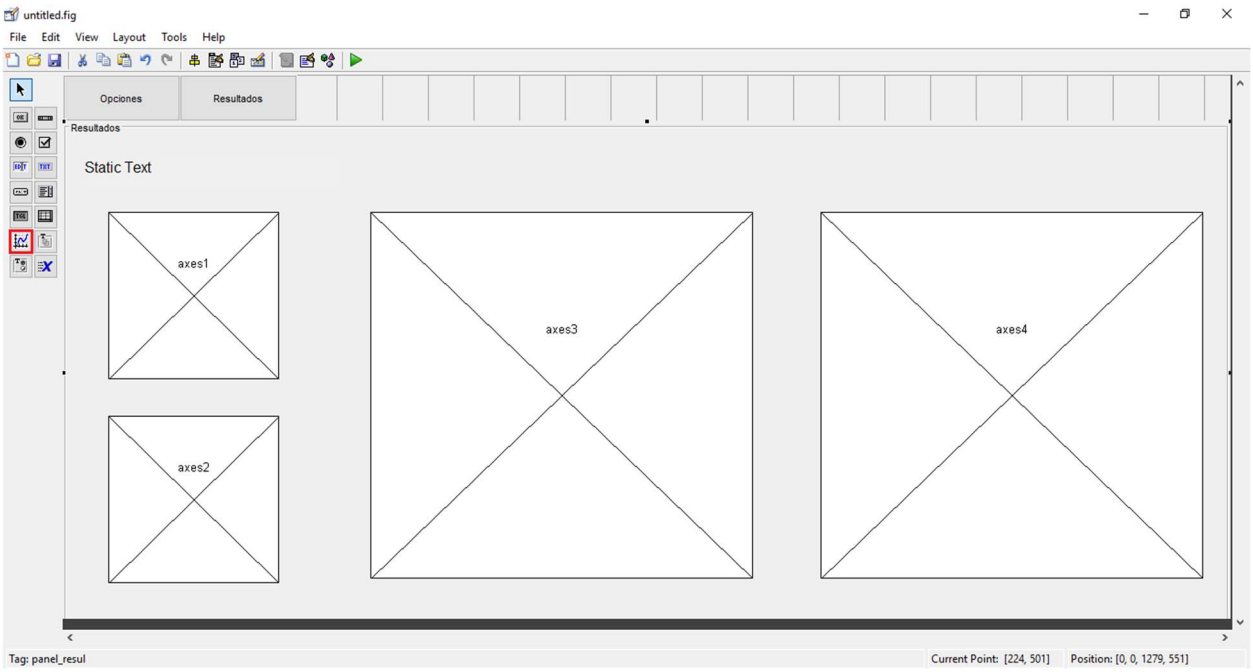



Figura 5-20. Resultado final del panel *Resultados*

Completados todos estos pasos, se puede dar por finalizado el diseño de la GUI, por lo que solo faltaría incluir las funcionalidades pertinenetes para cada elemento gráfico dispuesto. Dichas funcionalidades ya se han descrito en el apartado *Guía de Programación* de esta memoria, por lo que se puede pasar al mismo si se desea obtener el funcionamiento descrito en este trabajo o programar una funcionalidad diferente a gusto del usuario.

Un último paso sería el de la obtención del ejecutable del programa, una vez se ha terminado la aplicación en cuanto a diseño y programación, lo cual se consigue de manera muy fácil con el comando *deploytool* en la ventana de comandos de MATLAB. Una vez introducido se elige como compilador el *Application Compiler*, y, en la ventana que aparece, se rellenan los campos deseados, introduciendo el archivo principal de la GUI (el que la define), observando como automáticamente se incluyen las funciones usadas en el mismo para que se encuentren también dentro del ejecutable. Se añade una imagen de presentación para el tiempo de carga de la aplicación y se hace clic en el botón *Package*, obteniendo al final del proceso de creación del mismo, una carpeta que se emerge de forma automática y que contiene tanto al archivo ejecutable como a los demás ficheros comentados en el apartado anterior dentro de la subcarpeta *for_redistribution_files_only*.

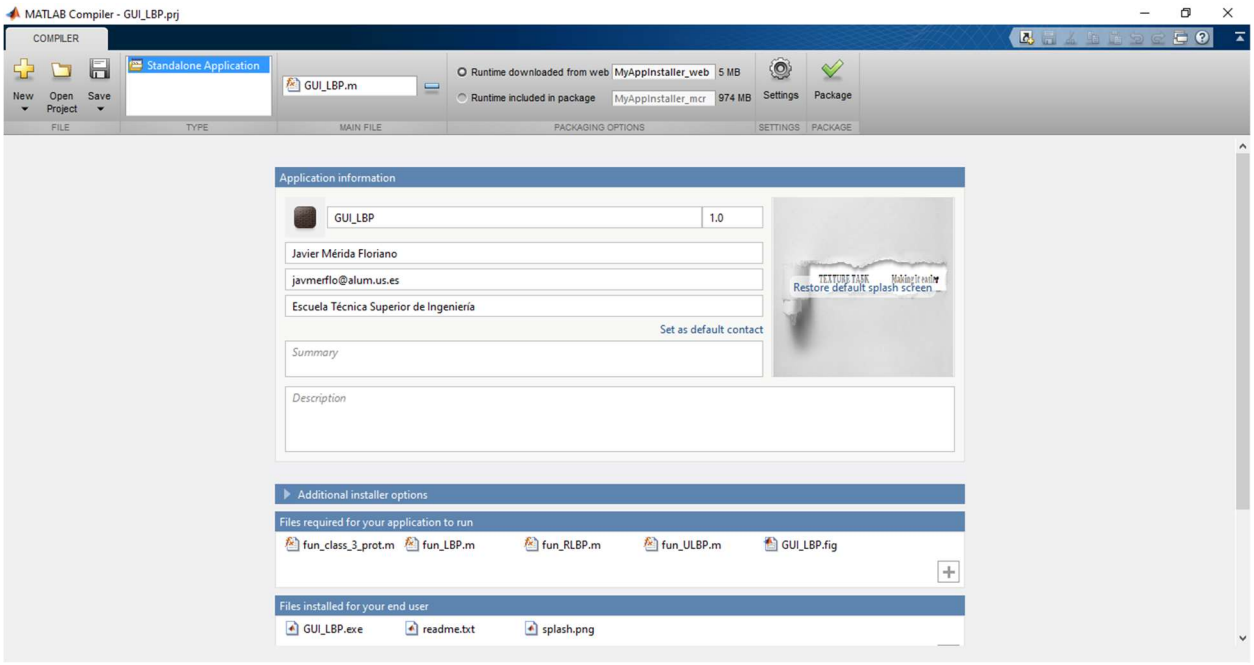



Figura 5-21. Creación del fichero ejecutable de la aplicación

# 6 CONCLUSIONES

Al término de este proyecto y de la exposición de su contenido, se pueden obtener unas últimas conclusiones acerca del mismo, englobando su estado actual y posible trabajo que se le pueda realizar en un futuro.

La primera conclusión respecto al trabajo realizado es que éste no se encuentra en su etapa final, puesto que únicamente es el primer paso para obtener la automatización casi completa del problema de clasificación de texturas. Como primera aproximación, la GUI consigue el objetivo de crear un entorno en el que encontrar una primera fase de la automatización, ya que gracias a ella el único procedimiento manual es el de la selección de la imagen a estudio, así como los parámetros seleccionados y la base de datos utilizada.

Más allá de lo que se ha conseguido en este proyecto, se busca una automatización más amplia y un funcionamiento casi perfecto de los algoritmos utilizados. En estos casos se pueden proponer algunos pasos extras a llevar a cabo:

1. Como para este problema aparecen muchos parámetros de índole distinta, existen muchos aspectos que se han de mejorar, y posiblemente mejorarán con el avance de la tecnología, para llegar al objetivo, como el avance en los métodos y algoritmos utilizados para la detección de texturas o la información contenida en el píxel, por ejemplo.
2. Otra mejoría para el funcionamiento de la clasificación se basa en la ampliación de la base de datos con texturas conocidas, es decir, por textura conocida, aumentar el número de imágenes que la definen, y así aumentar las posibilidades en cuanto a clasificación correcta. Se podría introducir una característica nueva a la aplicación en cuanto a este aspecto, como la de, una vez analizada una imagen y confirmada su textura por el usuario, introducirla como parte de la base de datos en cuestión de su textura asociada.
3. En cuanto a la aplicación como tal, se puede mejorar su portabilidad, haciendo posible su uso en cualquier sistema operativo y mejorar su portabilidad entre versiones, puesto que ahora mismo solo se puede utilizar con el *Runtime 9.1* de MATLAB.

# 7 ANEXO

## 7.1 Códigos de Programa

### A. fun_LBP

```
function [v,LBP]=fun_LBP(im,clase)

if clase==1
    im=rgb2gray(im);
end

[M,N]=size(im);
lbp=zeros(M,N);
v=zeros(1,256);

for i=2:M-1
    for j=2:N-1

        umbral=zeros(3,3);

        if (im(i-1,j-1)>=im(i,j))
            umbral(1,1)=1;
        end
        if (im(i-1,j)>=im(i,j))
            umbral(1,2)=2;
        end
        if (im(i-1,j+1)>=im(i,j))
            umbral(1,3)=4;
        end
        if (im(i,j+1)>=im(i,j))
            umbral(2,1)=8;
        end
        if (im(i+1,j+1)>=im(i,j))
            umbral(2,3)=16;
        end
        if (im(i+1,j)>=im(i,j))
            umbral(3,1)=32;
        end
        if (im(i+1,j-1)>=im(i,j))
            umbral(3,2)=64;
        end
        if(im(i,j-1)>=im(i,j))
            umbral(3,3)=128;
        end

        lbp(i,j)=sum(sum(umbral));
        v(lbp(i,j)+1)=v(lbp(i,j)+1)+1;
    end
end

v=v*100/((M-2)*(N-2));
LBP=uint8(lbp);
end
```

## B. fun_ULBP

```
function [v,ULBP]=fun_ULBP(im,clase)

if clase==1
    im=rgb2gray(im);
end

[M,N]=size(im);
ulbp=zeros(M,N);
v=zeros(1,58);

pesos_ULBP=[1;2;3;4;5;59;6;7;8;59;59;59;9;59;10;11;12;59;59;59;59;59;59;59;13
;59;59;59;14;59;15;16;17;59;59;59;59;59;59;59;59;59;59;59;59;59;59;59;18;59;5
9;59;59;59;59;59;19;59;59;59;20;59;21;22;23;59;59;59;59;59;59;59;59;59;59;59;
59;59;59;59;59;59;59;59;59;59;59;59;59;59;59;59;59;59;59;59;59;24;59;59;59;59
;59;59;59;59;59;59;59;59;59;59;25;59;59;59;59;59;59;59;26;59;59;59;27;59;28;2
9;30;31;59;32;59;59;59;33;59;59;59;59;59;59;59;34;59;59;59;59;59;59;59;59;59;
59;59;59;59;59;59;59;35;59;59;59;59;59;59;59;59;59;59;59;59;59;59;59;59;59;59
;59;59;59;59;59;59;59;59;59;59;59;59;36;37;38;59;39;59;59;59;40;59;59;59;59;5
9;59;59;41;59;59;59;59;59;59;59;59;59;59;59;59;59;59;59;42;43;44;59;45;59;59;
59;46;59;59;59;59;59;59;59;47;48;49;59;50;59;59;59;51;52;53;59;54;55;56;57;58
];


for i=2:M-1
    for j=2:N-1

        umbral=zeros(3,3);

        if (im(i-1,j-1)>=im(i,j))
            umbral(1,1)=1;
        end
        if (im(i-1,j)>=im(i,j))
            umbral(1,2)=2;
        end
        if (im(i-1,j+1)>=im(i,j))
            umbral(1,3)=4;
        end
        if (im(i,j+1)>=im(i,j))
            umbral(2,1)=8;
        end
        if (im(i+1,j+1)>=im(i,j))
            umbral(2,3)=16;
        end
        if (im(i+1,j)>=im(i,j))
            umbral(3,1)=32;
        end
        if (im(i+1,j-1)>=im(i,j))
            umbral(3,2)=64;
        end
        if(im(i,j-1)>=im(i,j))
            umbral(3,3)=128;
        end

        val=sum(sum(umbral));

        valor_ULBP=pesos_ULBP(val+1);
        ulbp(i,j)=valor_ULBP;
        if valor_ULBP<59
            v(valor_ULBP)=v(valor_ULBP)+1;
```

```
        end
    end
end

v=v*100/((M-2)*(N-2));
ULBP=uint8(ulbp);
end
```

### C. fun_RLBP

```
function [v,RLBP]=fun_RLBP(im,clase)

if clase==1
    im=rgb2gray(im);
end

[M,N]=size(im);
rlbp=zeros(M,N);
v=zeros(1,256);

plant_1=[  1   2   4; 128 0   8;  64  32  16];
plant_2=[128   1   2;  64 0   4;  32  16   8];
plant_3=[ 64 128   1;  32 0   2;  16   8   4];
plant_4=[ 32  64 128;  16 0   1;   8   4   2];
plant_5=[ 16  32  64;   8 0 128;   4   2   1];
plant_6=[  8  16  32;   4 0  64;   2   1 128];
plant_7=[  4   8  16;   2 0  32;   1 128  64];
plant_8=[  2   4   8;   1 0  16; 128  64  32];


for i=2:M-1
    for j=2:N-1

        bin=zeros(1,8);

        bin(1)=im(i-1,j-1);
        bin(2)=im(i-1,j);
        bin(3)=im(i-1,j+1);
        bin(4)=im(i,j+1);
        bin(5)=im(i+1,j+1);
        bin(6)=im(i+1,j);
        bin(7)=im(i+1,j-1);
        bin(8)=im(i,j-1);

        [maximo,ind]=max(bin);

        if ind==1
            plant=plant_1;
        elseif ind==2
            plant=plant_2;
        elseif ind==3
            plant=plant_3;
        elseif ind==4
            plant=plant_4;
        elseif ind==5
            plant=plant_5;
        elseif ind==6
            plant=plant_6;
        elseif ind==7
            plant=plant_7;
        elseif ind==8
```

```
            plant=plant_8;
        end

        umbral=zeros(3,3);

        if (im(i-1,j-1)>=im(i,j))
            umbral(1,1)=1;
        end
        if (im(i-1,j)>=im(i,j))
            umbral(1,2)=1;
        end
        if (im(i-1,j+1)>=im(i,j))
            umbral(1,3)=1;
        end
        if (im(i,j+1)>=im(i,j))
            umbral(2,3)=1;
        end
        if (im(i+1,j+1)>=im(i,j))
            umbral(3,3)=1;
        end
        if (im(i+1,j)>=im(i,j))
            umbral(3,2)=1;
        end
        if (im(i+1,j-1)>=im(i,j))
            umbral(3,1)=1;
        end
        if(im(i,j-1)>=im(i,j))
            umbral(2,1)=1;
        end

        mat_rlbp=plant.*umbral;
        rlbp(i,j)=sum(sum(mat_rlbp));
        v(rlbp(i,j)+1)=v(rlbp(i,j)+1)+1;
    end
end

v=v*100/((M-2)*(N-2));
RLBP=uint8(rlbp);
end
```

### D. fun_class_3_prot

```
function [plant_1,plant_2,plant_3]=fun_class_3_prot(im)

im=rgb2gray(im);

[M,N]=size(im);
M3=round(M/3);
N3=round(N/3);

plant_1=zeros(M3,N3);
plant_2=zeros(M3,N3);
plant_3=zeros(M3,N3);

i1=1;j1=1;
i2=1;j2=1;
i3=1;j3=1;

for i=1:M
    for j=1:N
        muest=randi([1 3]);
```

```
        if muest==1
            plant_1(i1,j1)=im(i,j);
            if i1<(M3)
                i1=i1+1;
            else
                j1=j1+1;
                i1=1;
            end
        elseif muest==2
            plant_2(i2,j2)=im(i,j);
            if i2<(M3)
                i2=i2+1;
            else
                j2=j2+1;
                i2=1;
            end
        else
            plant_3(i3,j3)=im(i,j);
            if i3<(M3)
                i3=i3+1;
            else
                j3=j3+1;
                i3=1;
            end
        end
    end
end

plant_1=uint8(plant_1);
plant_2=uint8(plant_2);
plant_3=uint8(plant_3);
end
```

## E. GUI_LBP

```
function varargout = GUI_LBP(varargin)

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
    'gui_Singleton',  gui_Singleton, ...
    'gui_OpeningFcn', @GUI_LBP_OpeningFcn, ...
    'gui_OutputFcn',  @GUI_LBP_OutputFcn, ...
    'gui_LayoutFcn',  [] , ...
    'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT


% --- Executes just before GUI_LBP is made visible.
function GUI_LBP_OpeningFcn(hObject, eventdata, handles, varargin)

% Choose default command line output for GUI_LBP
```

```
handles.output = hObject;

set(handles.Actualizar,'String',' ');
set(handles.Resultado,'String',' ');
handles.Samples=1;
handles.Method=1;
handles.Classifier='euclidean';
set(handles.panel_resul,'visible','off');
set(handles.panel_options,'visible','on');

guidata(hObject, handles);


% --- Outputs from this function are returned to the command line.
function varargout = GUI_LBP_OutputFcn(hObject, eventdata, handles)

% Get default command line output from handles structure

varargout{1} = handles.output;


% --- Executes on button press in Actualizar_Base.
function Actualizar_Base_Callback(hObject, eventdata, handles)

d=dir('Texturas_para_base_de_datos/*.jpg');

n=[];

if handles.Method==2
    vector=zeros(length(d),58);
else
    vector=zeros(length(d),256);
end

for i=1:length(d)

    n=[n,{d(i).name}];

    set(handles.Actualizar,'String',strcat('Actualizando...',{'
('},string(length(d)+1-i),{' '},'restantes',{')'}));
    drawnow;

    im=imread(['Texturas_para_base_de_datos/',d(i).name]);

    if handles.Method==1
        [v_i,LBP_i]=fun_LBP(im,1);
        if handles.Samples==3
            [plant_1,plant_2,plant_3]=fun_class_3_prot(im);
            [v1,LBP1]=fun_LBP(plant_1,3);
            [v2,LBP2]=fun_LBP(plant_2,3);
            [v3,LBP3]=fun_LBP(plant_3,3);
        end
    elseif handles.Method==2
        [v_i,LBP_i]=fun_ULBP(im,1);
        if handles.Samples==3
            [plant_1,plant_2,plant_3]=fun_class_3_prot(im);
            [v1,LBP1]=fun_ULBP(plant_1,3);
            [v2,LBP2]=fun_ULBP(plant_2,3);
            [v3,LBP3]=fun_ULBP(plant_3,3);
        end
    else
        [v_i,LBP_i]=fun_RLBP(im,1);
```

```
        if handles.Samples==3
            [plant_1,plant_2,plant_3]=fun_class_3_prot(im);
            [v1,LBP1]=fun_RLBP(plant_1,3);
            [v2,LBP2]=fun_RLBP(plant_2,3);
            [v3,LBP3]=fun_RLBP(plant_3,3);
        end
    end

    vector(i,:)=v_i;
    if handles.Samples==3
        vector1(i,:)=v1;
        vector2(i,:)=v2;
        vector3(i,:)=v3;
    end
end

Tam=length(d);
Names=erase(n,'.jpg');
if handles.Samples==1
    if handles.Method==1
        save('LBP_1','vector');
        save('Length_LBP_1','Tam');
        save('Names_LBP_1','Names');
    elseif handles.Method==2
        save('ULBP_1','vector');
        save('Length_ULBP_1','Tam');
        save('Names_ULBP_1','Names');
    else
        save('RLBP_1','vector');
        save('Length_RLBP_1','Tam');
        save('Names_RLBP_1','Names');
    end
else
    if handles.Method==1
        save('LBP_3','vector');
        save('LBP_3a','vector1');
        save('LBP_3b','vector2');
        save('LBP_3c','vector3');
        save('Length_LBP_3','Tam');
        save('Names_LBP_3','Names');
    elseif handles.Method==2
        save('ULBP_3','vector');
        save('ULBP_3a','vector1');
        save('ULBP_3b','vector2');
        save('ULBP_3c','vector3');
        save('Length_ULBP_3','Tam');
        save('Names_ULBP_3','Names');
    else
        save('RLBP_3','vector');
        save('RLBP_3a','vector1');
        save('RLBP_3b','vector2');
        save('RLBP_3c','vector3');
        save('Length_RLBP_3','Tam');
        save('Names_RLBP_3','Names');
    end
end

set(handles.Actualizar,'String','Actualización finalizada. Puede volver a
actualizar.');

guidata(hObject,handles);
```

```
% --- Executes on button press in LoadImage.
function LoadImage_Callback(hObject, eventdata, handles)

im=imread(uigetfile('*.jpg'));

if handles.Method==1
    [v,LBP]=fun_LBP(im,1);
    lim_x=256;
    x=0:255;
elseif handles.Method==2
    [v,LBP]=fun_ULBP(im,1);
    lim_x=58;
    x=0:57;
else
    [v,LBP]=fun_RLBP(im,1);
    lim_x=256;
    x=0:255;
end

if handles.Samples==1
    if handles.Method==1
        datos=load('LBP_1');
        Length=load('Length_LBP_1');
        Tags=load('Names_LBP_1');
    elseif handles.Method==2
        datos=load('ULBP_1');
        Length=load('Length_ULBP_1');
        Tags=load('Names_ULBP_1');
    else
        datos=load('RLBP_1');
        Length=load('Length_RLBP_1');
        Tags=load('Names_RLBP_1');
    end
else
    if handles.Method==1
        datos=load('LBP_3');
        datos1=load('LBP_3a');
        datos2=load('LBP_3b');
        datos3=load('LBP_3c');
        Length=load('Length_LBP_3');
        Tags=load('Names_LBP_3');
    elseif handles.Method==2
        datos=load('ULBP_3');
        datos1=load('ULBP_3a');
        datos2=load('ULBP_3b');
        datos3=load('ULBP_3c');
        Length=load('Length_ULBP_3');
        Tags=load('Names_ULBP_3');
    else
        datos=load('RLBP_3');
        datos1=load('RLBP_3a');
        datos2=load('RLBP_3b');
        datos3=load('RLBP_3c');
        Length=load('Length_RLBP_3');
        Tags=load('Names_RLBP_3');
    end
end

suma=1000;

Database_Length=Length.Tam;
Database_Tags=Tags.Names;
```

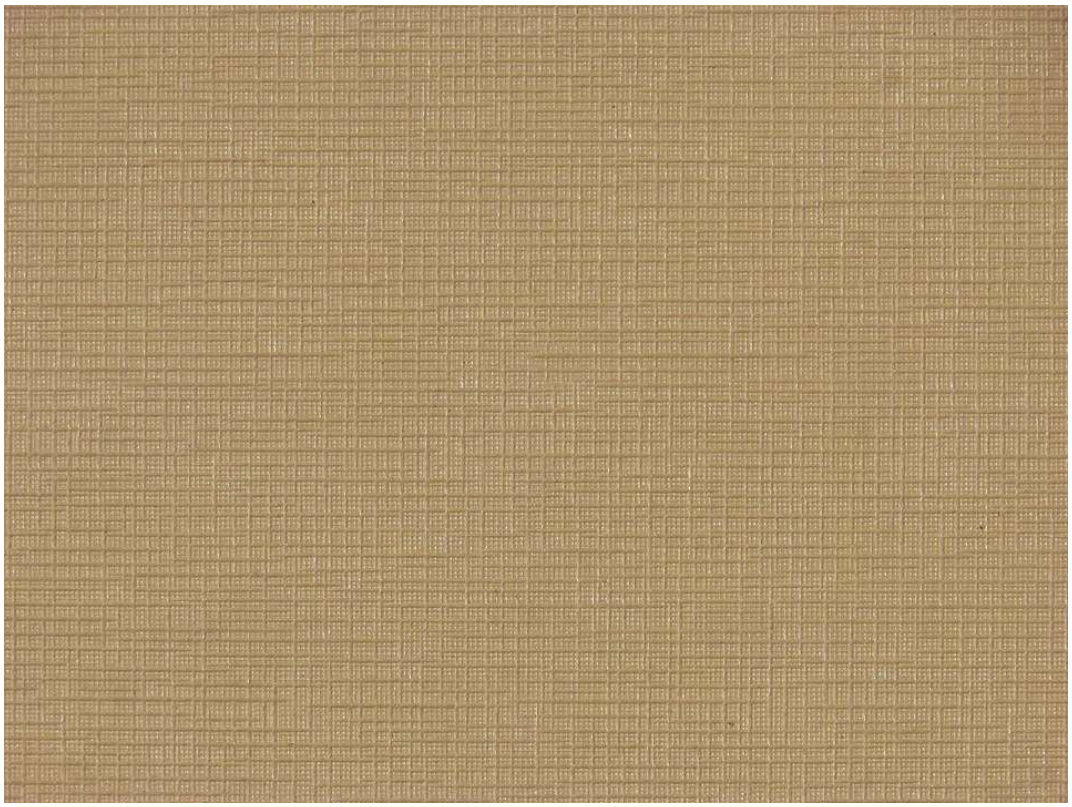
```
for k=1:Database_Length

    if handles.Samples==1
        if handles.Classifier=='minkowski'
            suma_aux=pdist2(v,datos.vector(k,:),handles.Classifier,3);
        else
            suma_aux=pdist2(v,datos.vector(k,:),handles.Classifier);
        end
    else
        if handles.Classifier=='minkowski'
            suma_aux1=pdist2(v,datos1.vector1(k,:),handles.Classifier,3);
            suma_aux2=pdist2(v,datos2.vector2(k,:),handles.Classifier,3);
            suma_aux3=pdist2(v,datos3.vector3(k,:),handles.Classifier,3);
        else
            suma_aux1=pdist2(v,datos1.vector1(k,:),handles.Classifier);
            suma_aux2=pdist2(v,datos2.vector2(k,:),handles.Classifier);
            suma_aux3=pdist2(v,datos3.vector3(k,:),handles.Classifier);
        end
        suma_aux=min([suma_aux1,suma_aux2,suma_aux3]);
    end

    if suma_aux<suma
        suma=suma_aux;
        res=k;
    end
end

axes(handles.axes1);
imshow(im);title('Imagen de estudio');

axes(handles.axes2);
imshow(LBP);
if handles.Method==1
    title('Matriz LBP de imagen a estudio');
elseif handles.Method==2
    title('Matriz ULBP de imagen a estudio');
else
    title('Matriz RLBP de imagen a estudio');
end

axes(handles.axes3);
bar(x,v);xlim([-1 lim_x]);
if handles.Method==1
    title('Histograma LBP normalizado de la imagen de estudio');xlabel('Valor
de LBP');
elseif handles.Method==2
    title('Histograma ULBP normalizado de la imagen de
estudio');xlabel('Valor de ULBP');
else
    title('Histograma RLBP normalizado de la imagen de
estudio');xlabel('Valor de RLBP');
end
ylabel('Porcentaje de píxeles de la imagen');

axes(handles.axes4);
bar(x,datos.vector(res,:));xlim([-1 lim_x]);
if handles.Method==1
    title('Histograma LBP normalizado de la textura patrón
asociada');xlabel('Valor de LBP');
elseif handles.Method==2
```

```
    title('Histograma ULBP normalizado de la textura patrón
asociada');xlabel('Valor de ULBP');
else
    title('Histograma RLBP normalizado de la textura patrón
asociada');xlabel('Valor de RLBP');
end
ylabel('Porcentaje de píxeles de la imagen');


set(handles.Resultado,'String',strcat({'Nombre de textura asociada:
'},Database_Tags(res)));


% --- Executes on button press in tabOpciones.
function tabOpciones_Callback(hObject, eventdata, handles)

set(handles.panel_resul,'visible','off');
set(handles.panel_options,'visible','on');

guidata(hObject,handles);

% --- Executes on button press in tabResultados.
function tabResultados_Callback(hObject, eventdata, handles)

set(handles.panel_resul,'visible','on');
set(handles.panel_options,'visible','off');

guidata(hObject,handles);


% --- Executes when selected object is changed in muestras.
function muestras_SelectionChangedFcn(hObject, eventdata, handles)

opt=get(hObject,'String');
switch opt
    case 'Una muestra'
        handles.Samples=1;
    case 'Tres muestras'
        handles.Samples=3;
end

guidata(hObject,handles);


% --- Executes when selected object is changed in metodo.
function metodo_SelectionChangedFcn(hObject, eventdata, handles)

opt=get(hObject,'String');
switch opt
    case 'LBP'
        handles.Method=1;
    case 'ULBP'
        handles.Method=2;
    case 'RLBP'
        handles.Method=3;
end

guidata(hObject,handles);


% --- Executes when selected object is changed in clasificador.
function clasificador_SelectionChangedFcn(hObject, eventdata, handles)
```

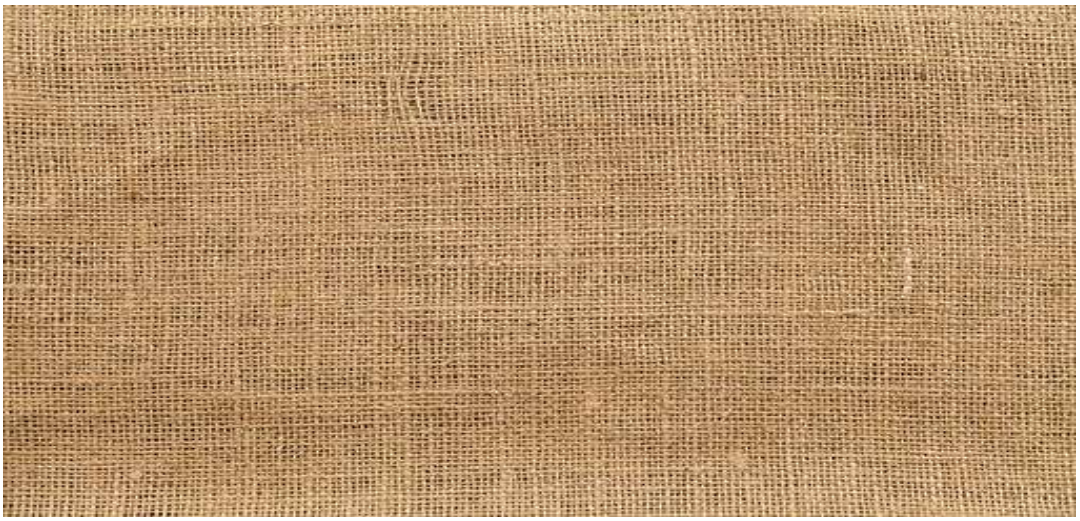
```
handles.Classifier=lower(get(hObject,'String'));

guidata(hObject,handles)
```

## 7.2 Imágenes para Base de Datos

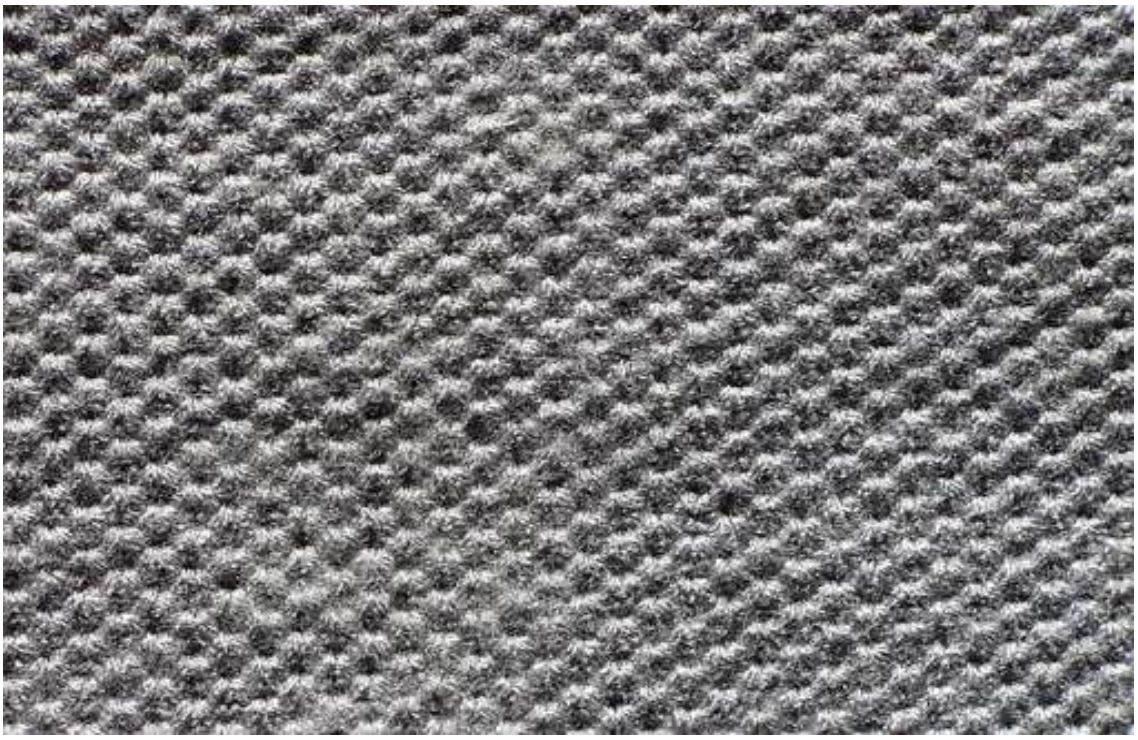
### A. Canvas



### B. Cloth (1 y 2)



**C. Cotton**



### D. Grass (1 y 2)

### E. Leather (1, 2 y 3)

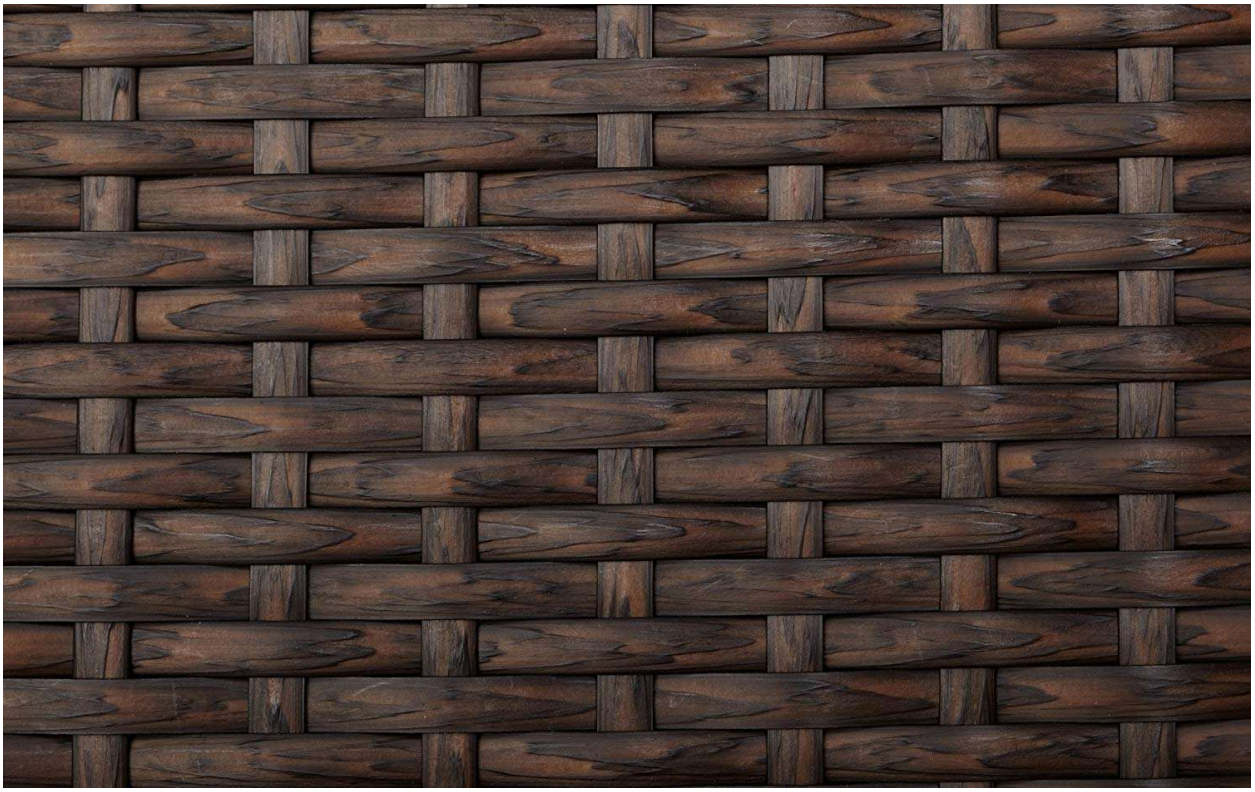
## F. Matting

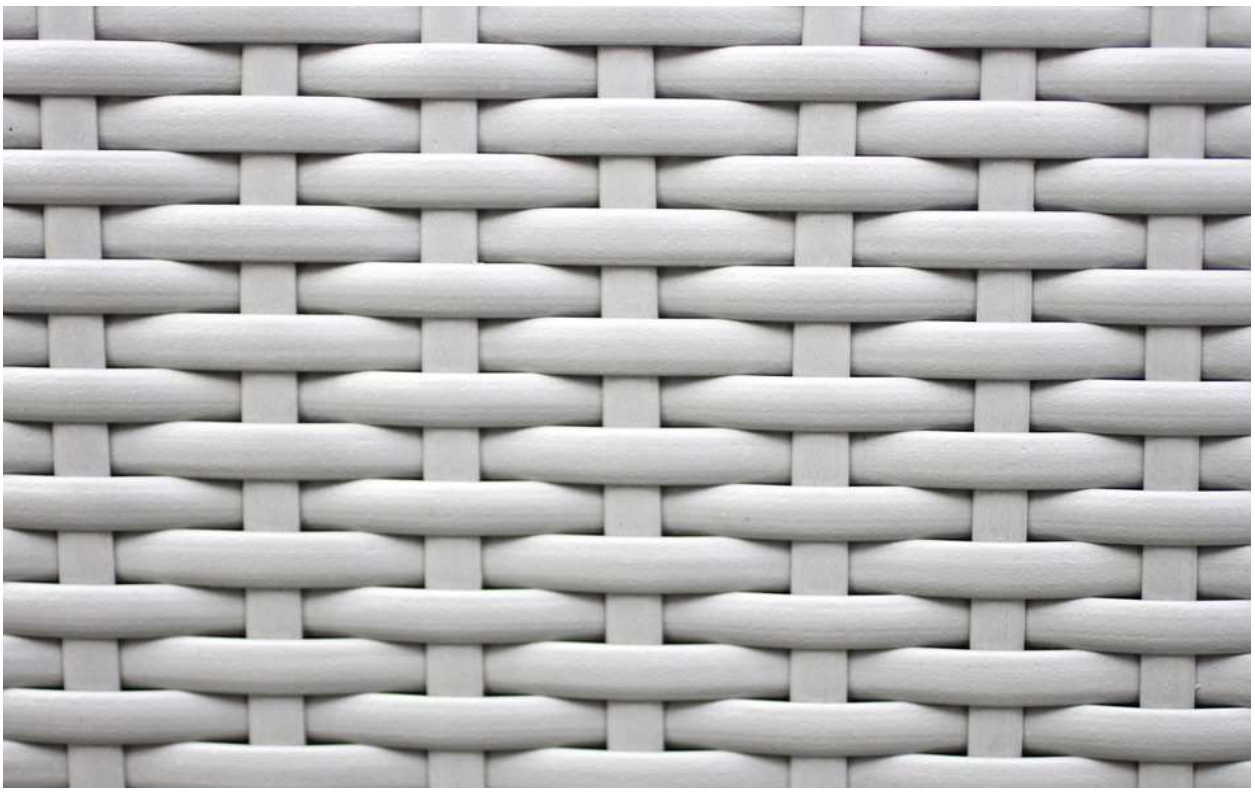
## G. Paper

**H. Pigskin**

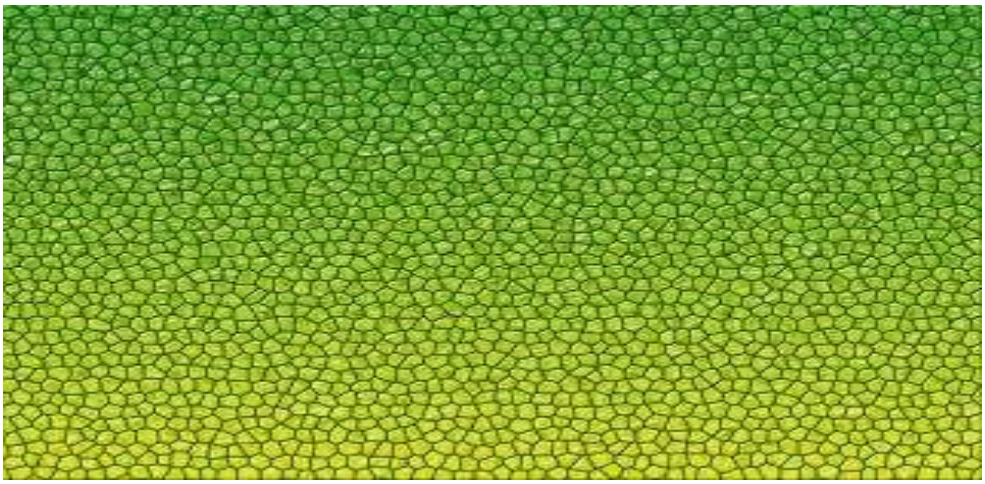
**I. Raffia**

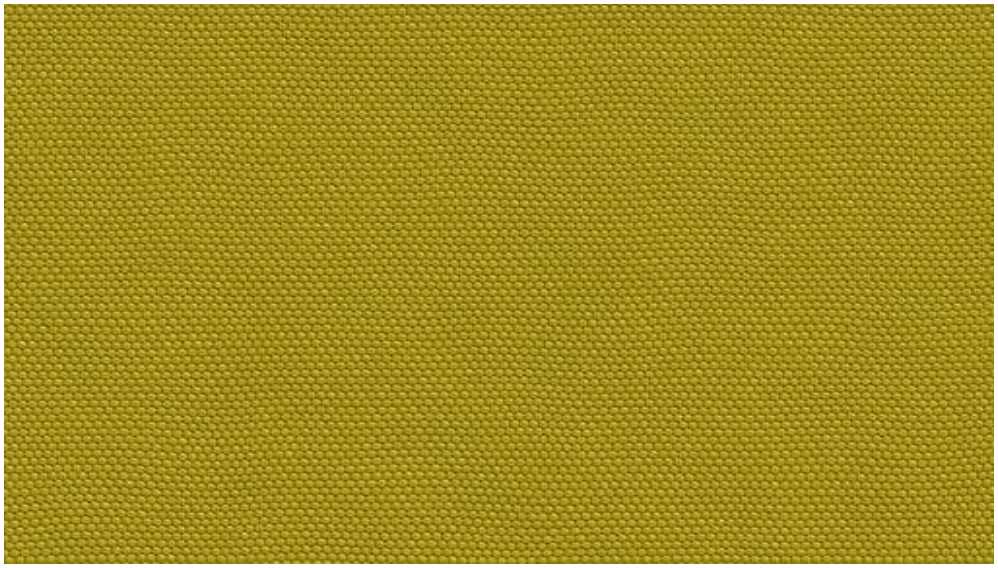
## J. Rattan (1 y 2)





**K. Reptile (1, 2 y 3)**



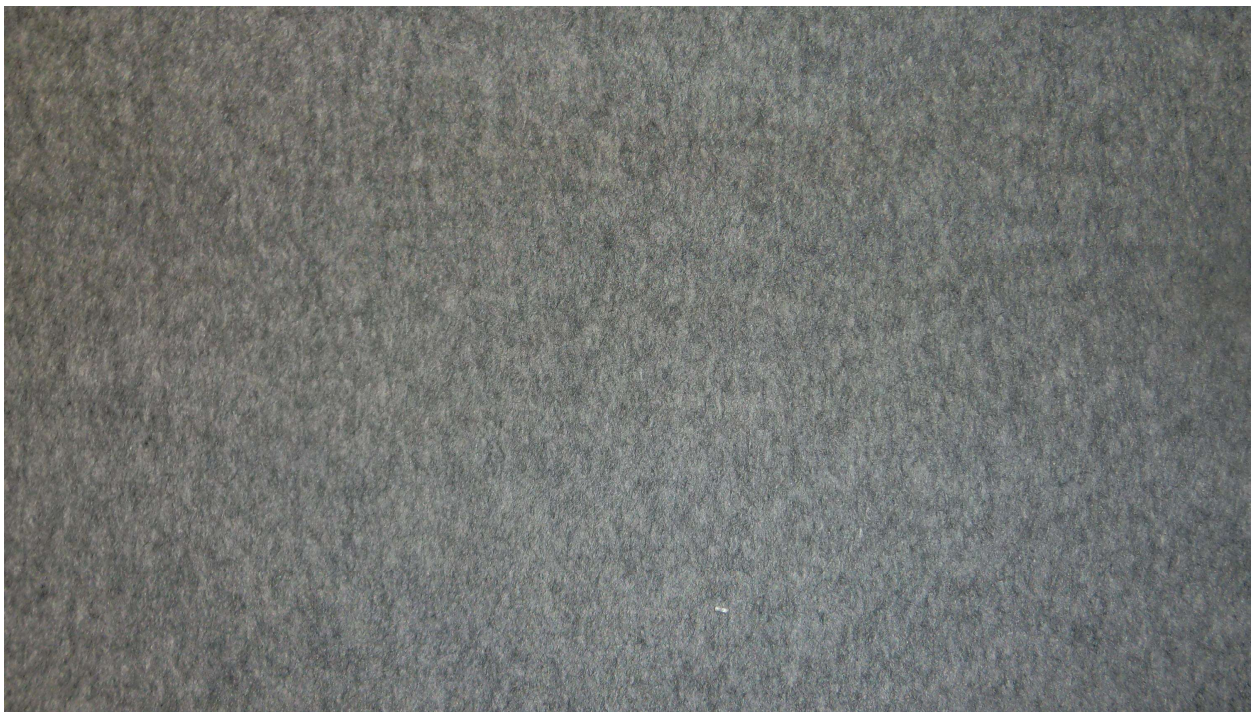


## L. Sand

## M. Straw

**N. Wood**

**O. Wool**



# REFERENCIAS

[1] T. Ojala, M. Pietikäinen and D. Harwood, «Performance Evaluation of Texture Measures with Classification Based on Kullnack Discrimination of Distributions,» de *Proceedings of 12th International Conference on Pattern Recognition*, Jerusalem, Israel, 1994.

[2] L. Wang and D. C. He, Texture classificationusing texture spectrum. Pattern recognition, vol. 23, 1990.

[3] G. Zhao, T. Ahonen, J. Matas and M. Pietikäinen, «Rotation-invariant image and video description with local binary pattern features,» de *IEEE Transactions on Image Processing*, 2012.

[4] X. Tang and B. Triggs, «Enhanced local textures feature sets for face recognition under difficult lighting conditions,» de *Analysis and Modeling of Faces and Gestures (AMFG)*, Rio de Janeiro, Brazil, 2007.

[5] Autor, «Este es el ejemplo de una cita,» *Tesis Doctoral,* vol. 2, nº 13, 2012.

[6] O. Autor, «Otra cita distinta,» *revista,* p. 12, 2001.

[7] «MATLAB - El lenguaje del cálculo técnico,» [En línea]. Available: https://es.mathworks.com/products/matlab.html.

[8] C. Troya Sherdek. [En línea]. Available: https://cesartroyasherdek.wordpress.com/2016/02/26/deteccion-de-objetos-vi/.