

Práctica 1

1) Líneas de nivel

Se implementaron las siguientes funciones para solucionar el ejercicio 1:

- $[p1, p2, n] = lut(v)$
- $cn = marchSquares(imagen, nivel, divGrilla)$
- $p = interpPreLin(v, p1, p2, v1, v2)$
- $imRes = dibujarRecta(imagen, p1, p2, intensidad)$

LUT

La función “lut” (“Look up table”), dado un vector de 4 componentes con valores 0 o 1 (correspondientes a los valores de los vértices de un cuadrado tomado en sentido horario), devuelve “las aristas” en las que se encuentran los vértices de los segmentos a dibujar. Si se tomaran los puntos medios de dichas aristas se podrían directamente devolver los vértices, pero como se va a realizar una interpolación para determinar la ubicación “más exacta” de esos puntos, sólo se devuelven “aristas”.

Cada “arista” está determinada por 4 componentes, 2 coordenadas por cada uno de sus vértices. Por lo tanto, para definir un segmento a dibujar se devuelven 8 componentes (o 2 aristas), en los casos en que se dibuja 1 segmento, o 16 componentes, en los casos en que se dibujan 2 segmentos. La cantidad de segmentos a dibujar viene dada por “n”.

Abajo se copia la tabla mostrando la codificación adoptada (“n” se encuentra en la posición 9, en la mitad de las columnas, debido a que inicialmente se consideró un sólo segmento; los casos ambiguos tienen n=2):

```
% tabla = [i_p1_ant j_p1_ant i_p1_pos j_p1_pos i_p2_ant j_p2_ant
i_p2_pos j_p2_pos n i_p3_ant j_p3_ant i_p3_pos j_p3_pos i_p4_ant
j_p4_ant i_p4_pos j_p4_pos]
```

```
tabla = [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
          0 0 0 1 0 1 1 1 1 0 0 0 0 0 0 0;
          1 0 1 1 1 1 0 1 1 0 0 0 0 0 0 0;
          0 0 0 1 1 0 1 1 1 0 0 0 0 0 0 0;
          0 0 1 0 1 0 1 1 1 0 0 0 0 0 0 0;
          0 0 1 0 0 0 0 1 2 1 0 1 1 1 1 0 1; %ambiguo
          0 0 1 0 0 1 1 1 1 0 0 0 0 0 0 0;
          0 0 1 0 0 0 0 1 1 0 0 0 0 0 0 0;
          0 0 1 0 0 0 0 1 1 0 0 0 0 0 0 0;
          0 0 1 0 0 1 1 1 1 0 0 0 0 0 0 0;
          0 0 1 0 1 0 1 1 2 0 0 0 1 0 1 1 1; %ambiguo
          0 0 1 0 1 0 1 1 1 0 0 0 0 0 0 0;
          0 0 0 1 1 0 1 1 1 0 0 0 0 0 0 0;
          1 0 1 1 1 1 0 1 1 0 0 0 0 0 0 0;
          0 0 0 1 0 1 1 1 1 0 0 0 0 0 0 0;
          0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

];

Esta LUT sólo trabaja con valores 1 o 0 y por lo tanto no realiza la desambiguación de los puntos silla. Esto se hace de forma externa en la función “marchSquares”.

marchSquares

“marchSquares” es la función principal de este ejercicio. Dada una imagen y un nivel, devuelve la curva de nivel para ese valor.

Se debe especificar un valor “divGrilla” para que el resultado del algoritmo tenga algo de sentido. “divGrilla” determina cuántos puntos se agregan en la grilla resultante por cada punto de la grilla original. Si este valor fuera 1, toda la elaboración de los segmentos de recta dejaría de tener sentido porque se reducirían a 1 píxel. Aumentar mucho el valor de “divGrilla” provoca que el algoritmo implementado demore mucho (aunque no era el objetivo conseguir un algoritmo rápido), pero mejora la calidad del resultado. Para imágenes naturales (con muchos píxels) se probó con valores hasta 5, pero un valor de 3 es muy aceptable (yo no llego a notar diferencias visuales para valores mayores que 3). Para la prueba con una imagen artificial se tomaron valores más grandes (20 o hasta 100), ya que al haber menos píxels el tiempo de procesamiento es menor (20 es un valor perfectamente aceptable).

La versión final de esta función realiza desambiguación entre los casos “silla”, e interpola para encontrar, dentro de cada arista, una posición mejor aproximada para los extremos de los segmentos a dibujar (para esto último llama a la función interpPreLin).

InterpPreLin

Esta función se utiliza para, dados los 2 vértices de una arista, y sus valores (en escala de grises u otra) correspondientes, encontrar dónde, en la arista, estaría un punto con un valor de intensidad dado. Las coordenadas se ingresan en posiciones de píxel (i,j) y se devuelven como (i*,j*) donde estas últimas pueden no coincidir con ningún punto de la grilla a utilizar (quien llama a la función se debe encargar de redondear luego para ajustar a la grilla que vaya a utilizar).

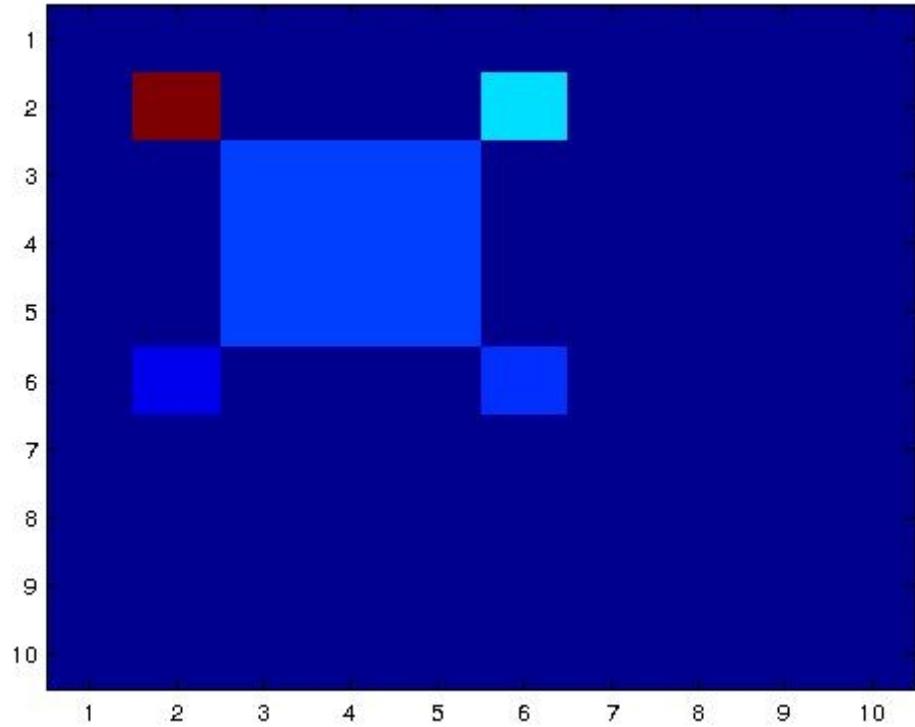
dibujarRecta

A esta función se le pasan dos puntos (con 2 coordenadas cada uno), una imagen y un nivel de intensidad y dibuja un segmento que los une en la imagen resultante (igual a la imagen original con el segmento agregado).

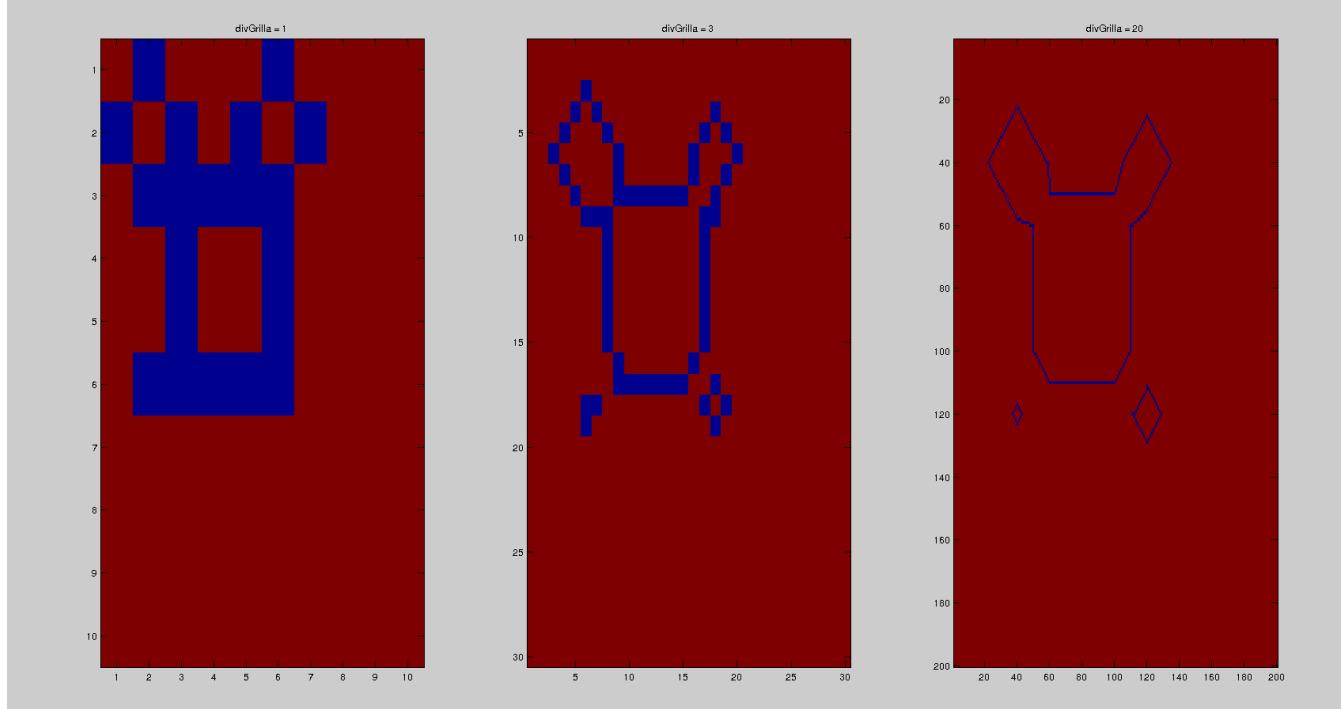
Todas las coordenadas se trabajan en “posición de píxel”. De inmediato se reconoció que había que distinguir las rectas verticales para evitar divisiones por cero. Luego se descubrió que las rectas “casi verticales” quedaban muy mal dibujadas también, al usar las “x” (“j”) como parámetro siempre, así que se dividió el plano en “rectas con ángulo menor a 45°” y “rectas con ángulo mayor a 45°”, parametrizando según la variable más conveniente en cada caso. La mejora lograda con esa técnica fue muy notoria en la imagen.

Resultados

La imagen artificial (10x10 pixels) utilizada es la de abajo:



El resultado con umbral=0.5, y distintos valores de “divGrilla” (1, 3, 20) puede verse abajo. Es importante notar que la imagen fue elegida para mostrar todos los casos “silla” posibles, así como los efectos de la aplicación de la interpolación (ver las distintas pendientes de las rectas).

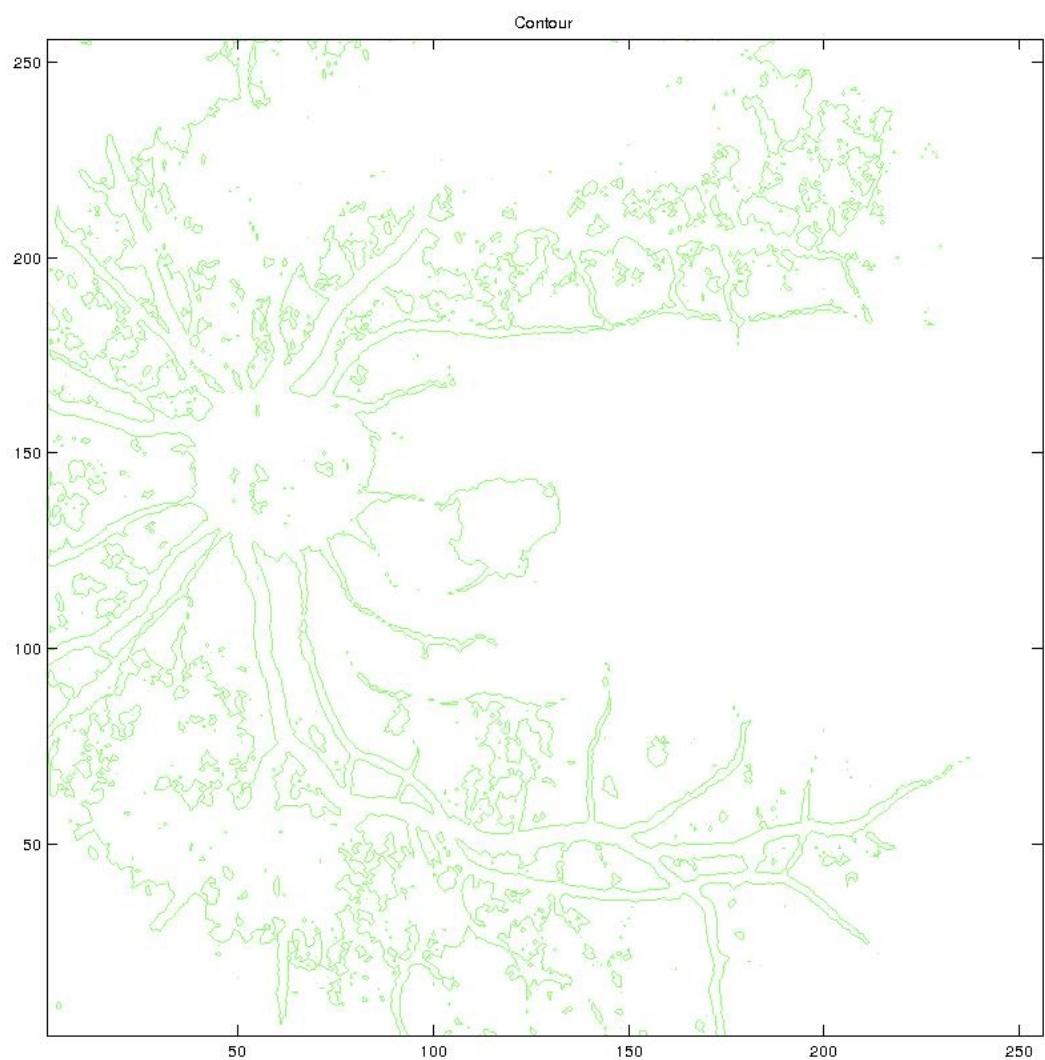


Luego se utilizó el algoritmo, con valor “divGrilla=3” para las imágenes “retina” y “lena”. Se podría utilizar valores de “divGrilla” mayores que 3, pero, además del mayor tiempo de demora, la imagen queda tan grande que Matlab no la puede mostrar completamente, por lo que, visualmente, queda peor (aunque “en memoria” quede mejor).
Se compara con la función “contour” de Matlab.

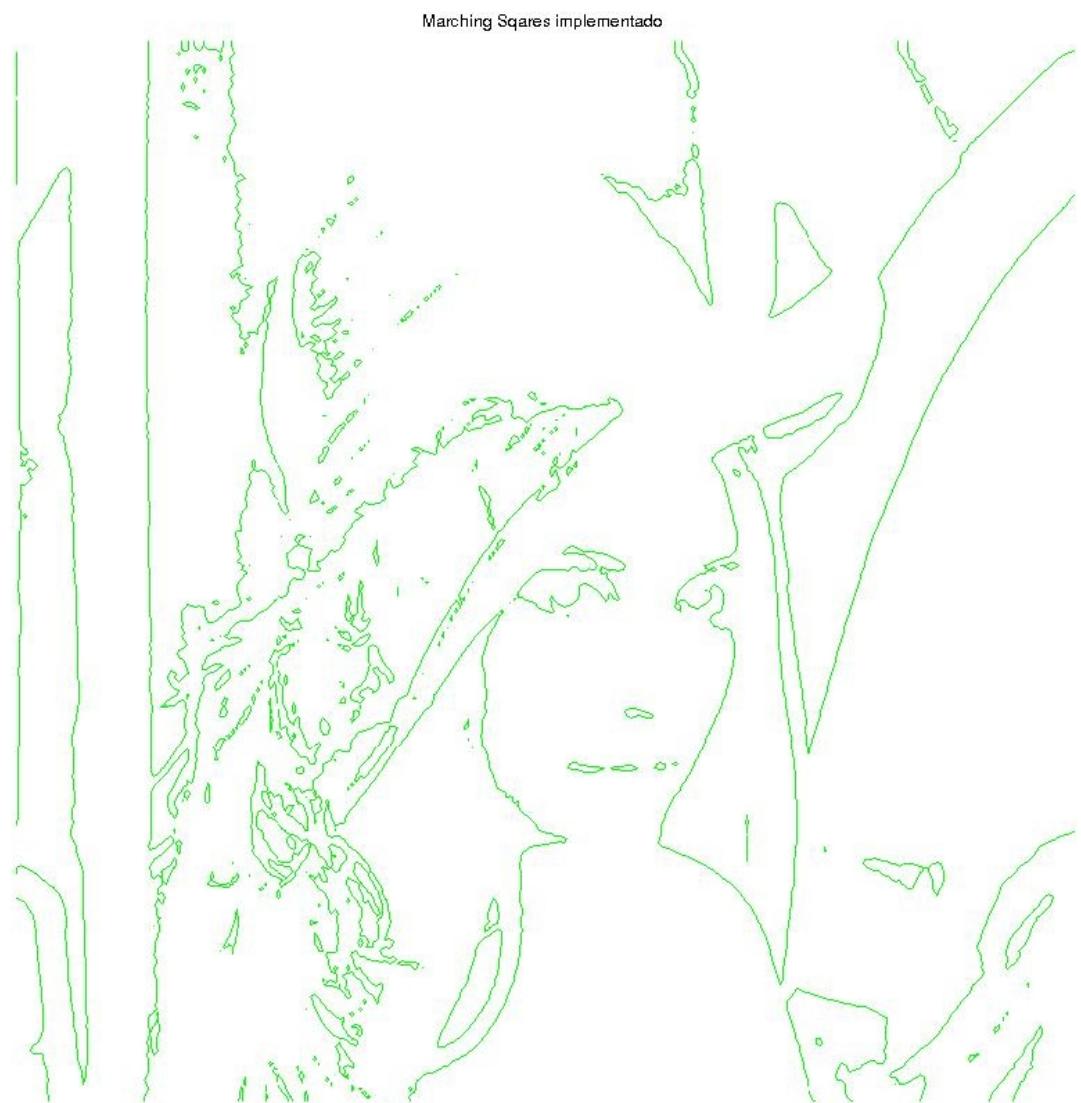
Versión implementada:



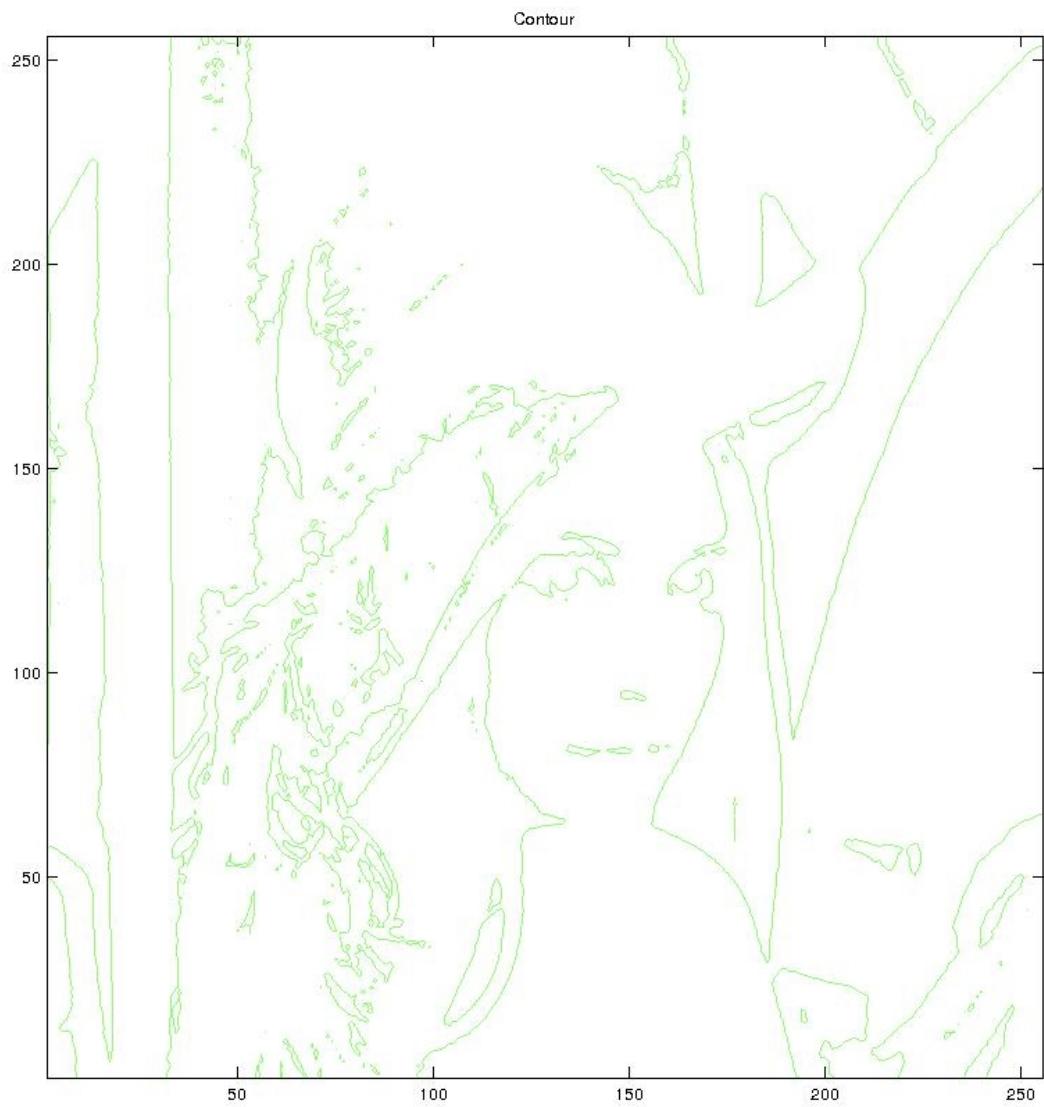
Versión de Matlab:



Versión implementada:



Versión de Matlab:



Conclusiones de esta parte:

Se implementó un algoritmo de Marching Squares con desambiguación de puntos “silla” e interpolación.

Los resultados visuales entre la implementación y la función “contour” son indistinguibles, excepto por el uso de distintos modos de visualización y/o nivel de zoom.

El tiempo de procesamiento de la versión implementada es MUCHO mayor que el de la función “contour” de Matlab.

Transformaciones Geométricas

Parte 2)

Se implementó la función “transformar” con los tres tipos de interpolación. Un problema que se enfrentó fue el de “la ventana de visualización”. El problema es que se tiene una ventana rectangular, acotada, y al transformar la imagen la nueva puede salirse de la misma. En esos casos se decidió elegir entre dos opciones:

Opción 1) Mantener la ventana y llenar con “blanco” las zonas que la imagen deje libre
Opción 2) Agrandar la ventana para que la nueva imagen entre completamente en ella, dejando en “blanco” las zonas que puedan haber quedado libres.

Las dos opciones se implementaron, y “transformar” realiza una o la otra dependiendo de un parámetro llamado “entera” (agregado a la definición de la función).

Este problema también afecta al momento de querer componer transformaciones, debido a la pérdida de información de la imagen en el caso de la opción 1, o a la pérdida de información sobre el origen y tamaño de la ventana original en la opción 2. Creo que para realizar composiciones de transformaciones llamando varias veces a la función “transformar” se debería guardar toda la información (como en la opción 2) pero además devolver el dato de la ventana original (por ejemplo, manteniendo una copia en memoria más grande que la que se visualice).

Por el momento, la composición pedida en la parte “e” se realizó multiplicando matrices, aunque se sospecha que la idea era lograr que la función “transformar” fuera “compatible con composiciones”. Si ese fuera el caso, creo que el camino sería agregando información de “ventana de visualización” independiente de la “imagen en memoria”.

La opción 2, se utilizó más que nada para mostrar los tipos distintos de interpolación.

Abajo pueden verse los resultados:

Rotación a 45° con Interpolación del Vecino Más Cercano



Rotación con vecino mas cercano. Ventana agrandada.



Rotación a 45° con Interpolación Bilineal



NOTA: Se tuvieron en cuenta los casos “degenerados”, en que los puntos caen en aristas o vértices para la interpolación.

Rotación a 45° con Interpolación Bicúbica



NOTA: Se observa un ruido artificial que parece “periódico” en el caso de interpolación bicúbica. No se implementaron casos “degenerados” para la interpolación “bicúbica” (esa podría ser la causa; encontré información sobre overshooting en este tipo de interpolación pero no parece coincidir con el tipo de ruido).

Rotación a 45º con Interpolación Bilineal, sin agrandar la ventana (rotación “natural”)

Rotación con int. bilineal. Ventana no cambiada.

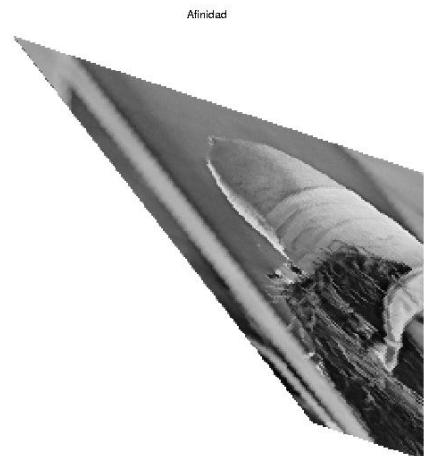


Traslación con Interpolación del Bilineal

Traslación con int. bilineal



Transformación Afín (vecino más cercano)



Afinidad

NOTA: Aparecían algunos puntos aislados en blanco para la interpolación bilineal y la bicúbica

Transformación Proyectiva general (vecino más cercano)



Proyectividad

NOTA: Aparecían algunos puntos aislados en blanco para la interpolación bilineal y la bicúbica

Composición de transformaciones (multiplicando matrices), que da como resultado la rotación centrada en el centro de la imagen.



Rotación centrada en cero, multiplicando matrices



Parte 3)

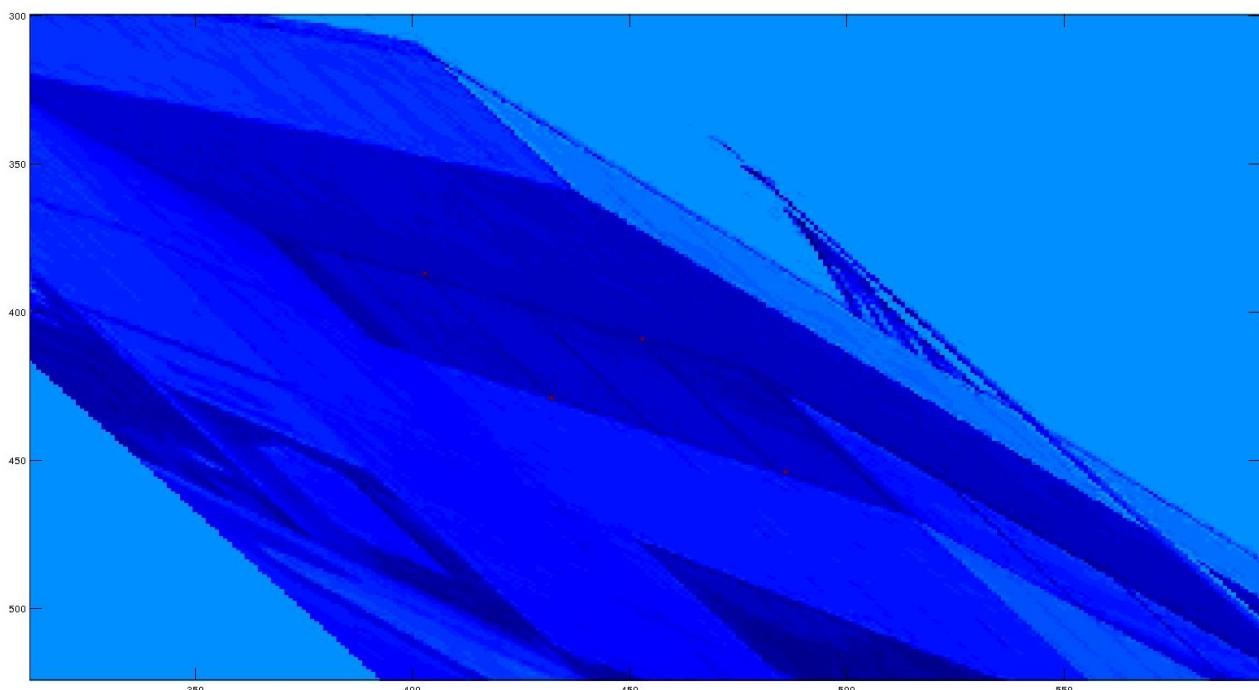
a)

Se implementó la función calcularHomografía y se la probó en el script ej3.m.

Para la prueba se utilizó la imagen “edificio.jpg”. Se la modificó mediante la función “transformar” de la parte 2, con una homografía general pero conocida. Luego se seleccionaron cuatro puntos en la imagen original. Los puntos fueron elegidos por ser fácilmente identificables (en este caso son vértices de unas ventanas). Se comprobó que al transformar por H , los puntos correspondieran a los vértices de las ventanas transformados (asignándoles un valor 1000 en una imagen que tiene una escala de 0-255, razón por la que se ven en rojo en la imagen de abajo).

Abajo se ve la comprobación de que los puntos son correctamente transformados.

Luego se comparó la “ H calculada” según “calcularHomografía”, y la “ H conocida” inicial. El resultado fue correcto.



$H =$

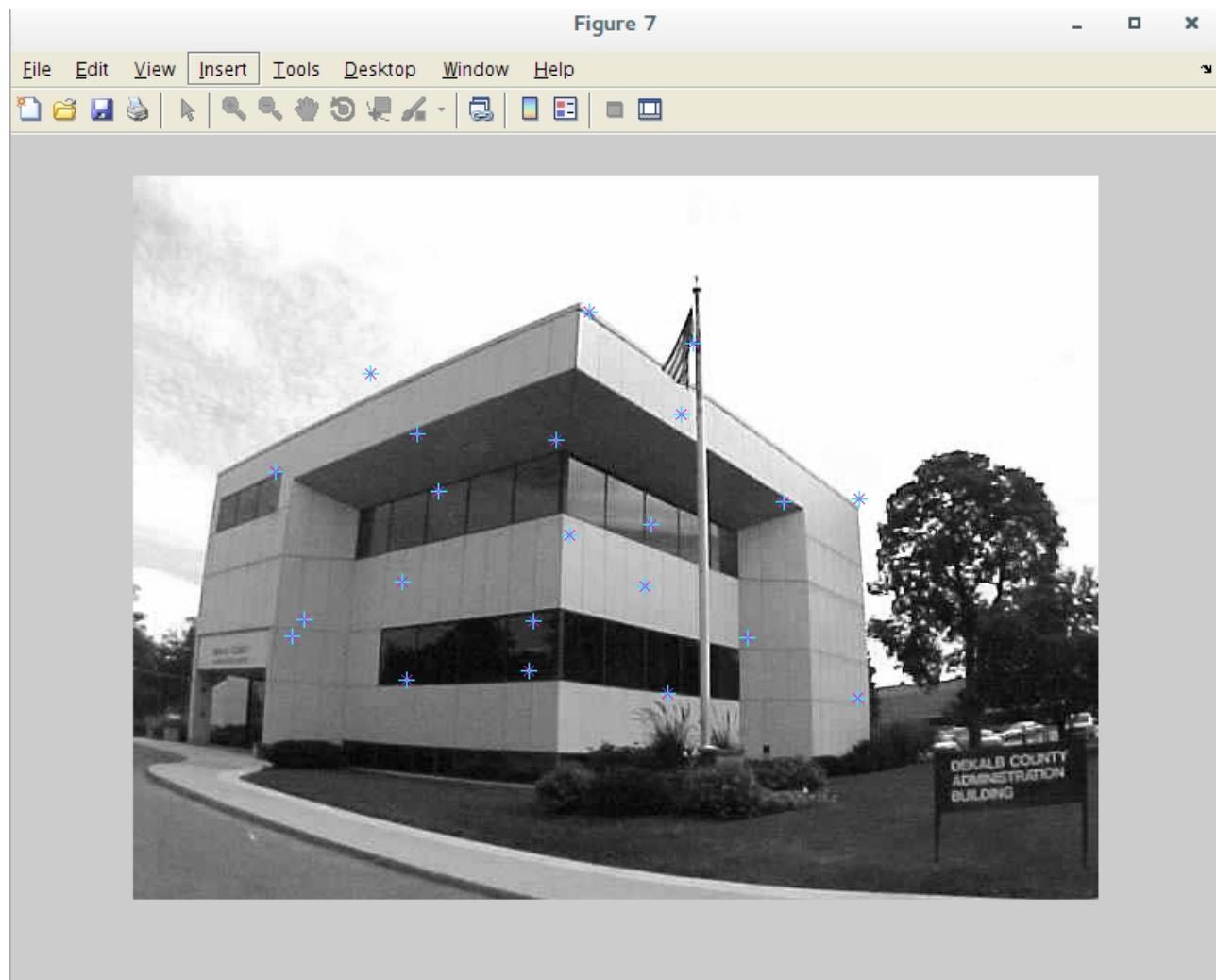
$$\begin{matrix} 1.1000 & 0.9000 & 0 \\ 0.7000 & 1.2000 & 0 \\ 0 & 0 & 1.0000 \end{matrix}$$

$H_{calc} =$

$$\begin{matrix} 1.1000 & 0.9000 & -0.0000 \\ 0.7000 & 1.2000 & -0.0000 \\ 0 & 0 & 1.0000 \end{matrix}$$

b) Si se tienen más de 4 puntos se puede hallar la matriz H utilizando mínimos cuadrados. Para eso se modificó la función “calcularHomografia.m”, resultando la función “calcularHomografíaMC.m”. En el script “ej3.m”, se prueba ahora con muchos puntos (cantidad arbitraria, mayor que 4), y se calcula de nuevo la matriz H . El resultado fue el correcto (abajo se puede ver la imagen de los puntos y el resultado). En este caso no se tenían “datos con ruido”, ya que las correspondencias

fueron establecidas a partir de una matriz conocida (sin agregar ruido), por lo que no habrán “outliers”. Para hacer que el algoritmo sea robusto a “outliers”, se puede implementar (sobre la implementación de mínimos cuadrados), el algoritmo RANSAC, que justamente permite descartar valores que parecen “outliers”, con un procedimiento iterativo.



Resultado con mínimos cuadrados:

HcalcMC =

1.1000	0.9000	0.0000
0.7000	1.2000	0.0000
-0.0000	-0.0000	1.0000

Parte 4)

Se utilizaron 3 imágenes de la página:

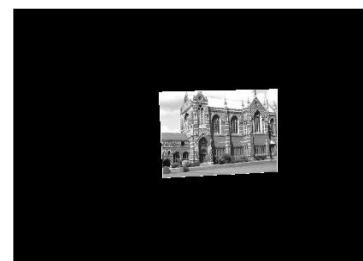
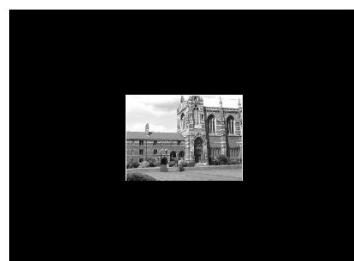
http://www.robots.ox.ac.uk/~vgg/hzbook/hzbook2/WebPage/chapter_8.html.

Se tomó la imagen central como fija.

El script “ej4_conseguirPuntos.m” permite elegir los 4 puntos en cada una de las imágenes y los guarda en el archivo “puntos.mat” (tener en cuenta que modificar los puntos puede cambiar los resultados; como esta era una tarea tediosa, se decidió elegirlos una sola vez para poder probar el algoritmo con mayor rapidez). Es importante notar que, antes de elegir los puntos, se colocan las 3 imágenes en un lienzo común (usando la función “centrarImagen.m”), lo que además les provee de un sistema de coordenadas común (en ej4 se reproduce este procedimiento para el pegado). De esta forma no será necesario trasladarlas para el pegado (como se verá más adelante).

Luego el script “ej4.m” calcula las homografías correspondientes, transforma las imágenes 1 y 3, según la homografía que lleva sus puntos en los correspondientes de la imagen 2. Por último se “pegan” las 3 imágenes tomando el máximo “punto a punto”. De esa forma se evita que los “ceros” del agrandado de los lienzos “tapen” puntos de imagen. En los casos en que más de una imagen tiene “puntos de imagen real” tomar el máximo no debería causar mayores problemas (los valores serán similares), por lo menos si se asume una iluminación parecida (estandarizar las imágenes según la iluminación iría más allá del objetivo de este ejercicio).

Abajo pueden verse las 3 imágenes transformadas (separadas), y luego la panorámica resultante.

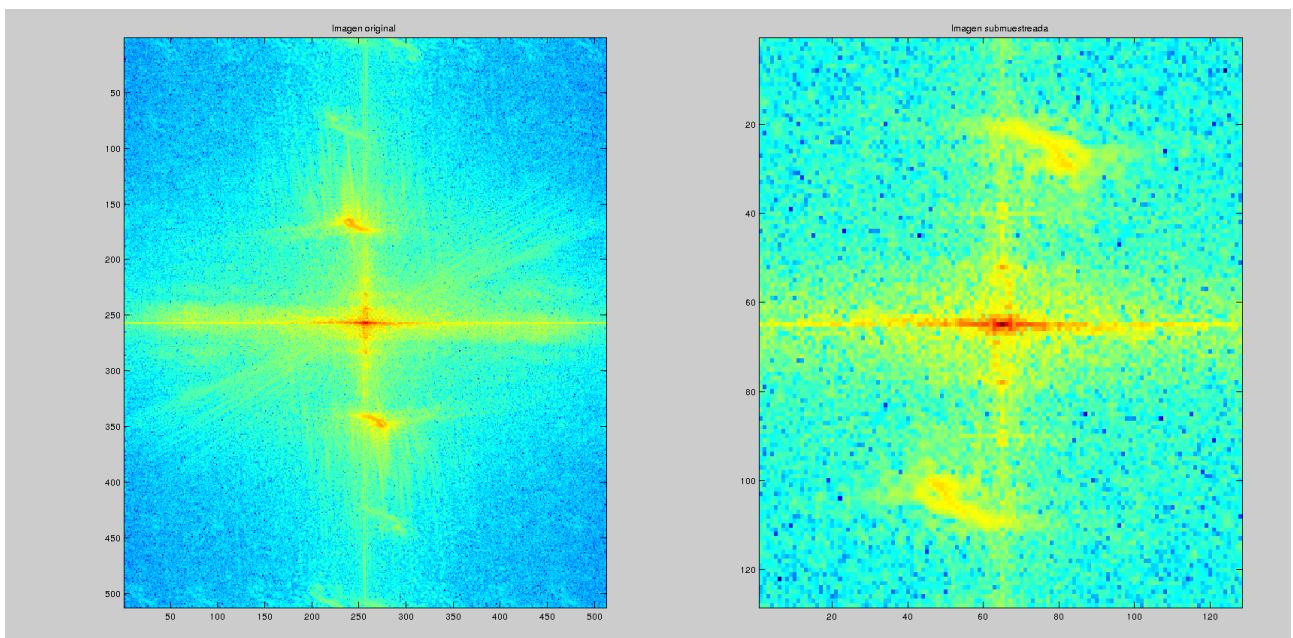




Transformada DFT

Parte 5)

- Se submuestreó usando la función “subMuestreo” implementada.
- Abajo pueden verse la imagen original, la submuestreada y sus transformadas.

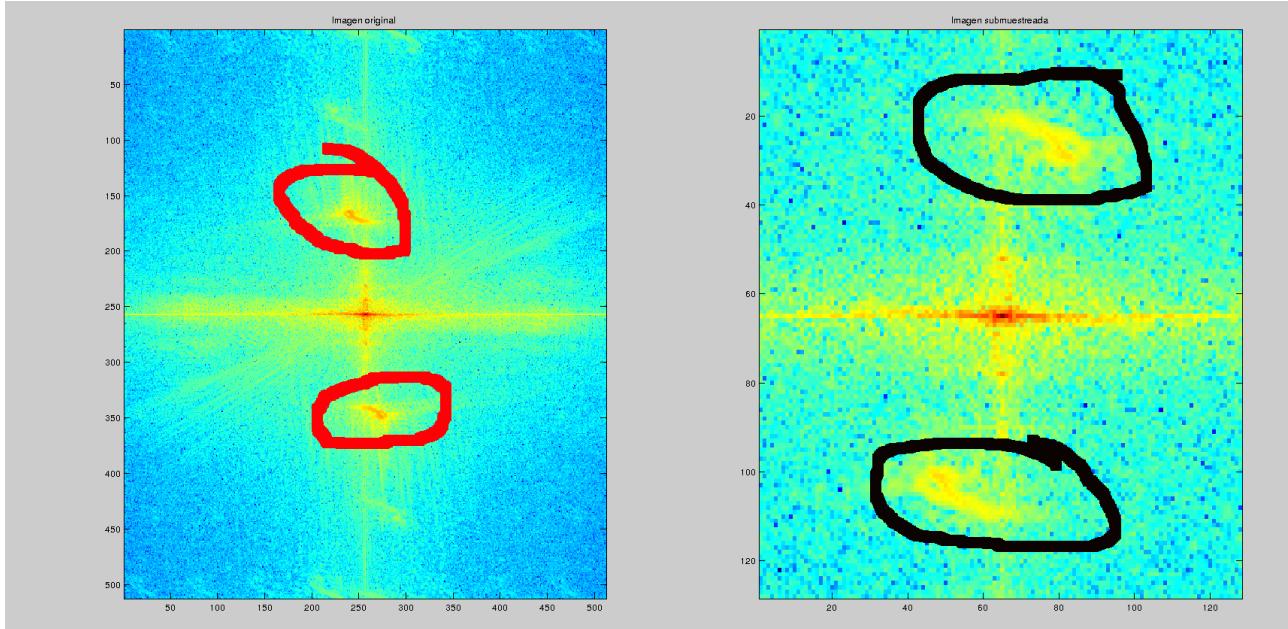


En la imagen original, el trapo tiene un periodo de aproximadamente 6 píxeles, en vertical. Dado que el periodo de la frecuencia máxima es de 2 píxeles, y está representado por la frecuencia “256” (en píxeles de la imagen de la DFT), la frecuencia del trapo es 3 veces menor que la máxima, y por lo tanto estará a $256/3 = 85,3$ píxeles del origen, aproximadamente. Es decir que para encontrar las frecuencias del trapo hay que mirar en el eje vertical, manchas centradas en 171 y 341 aproximadamente (dado que el centro de la imagen está en el píxel 256). En la imagen de la DFT original se ven unas “comas” aproximadamente en ese lugar.

Las comas invertidas que se ven en la DFT de la imagen submuestreada provienen del fenómeno de aliasing como se explicará en la parte c.

Abajo se repite la imagen con las DFT, pero marcando las “comas” del trapo original en rojo, y

otras “comas” que después se explicará por qué aparecen, en negro.

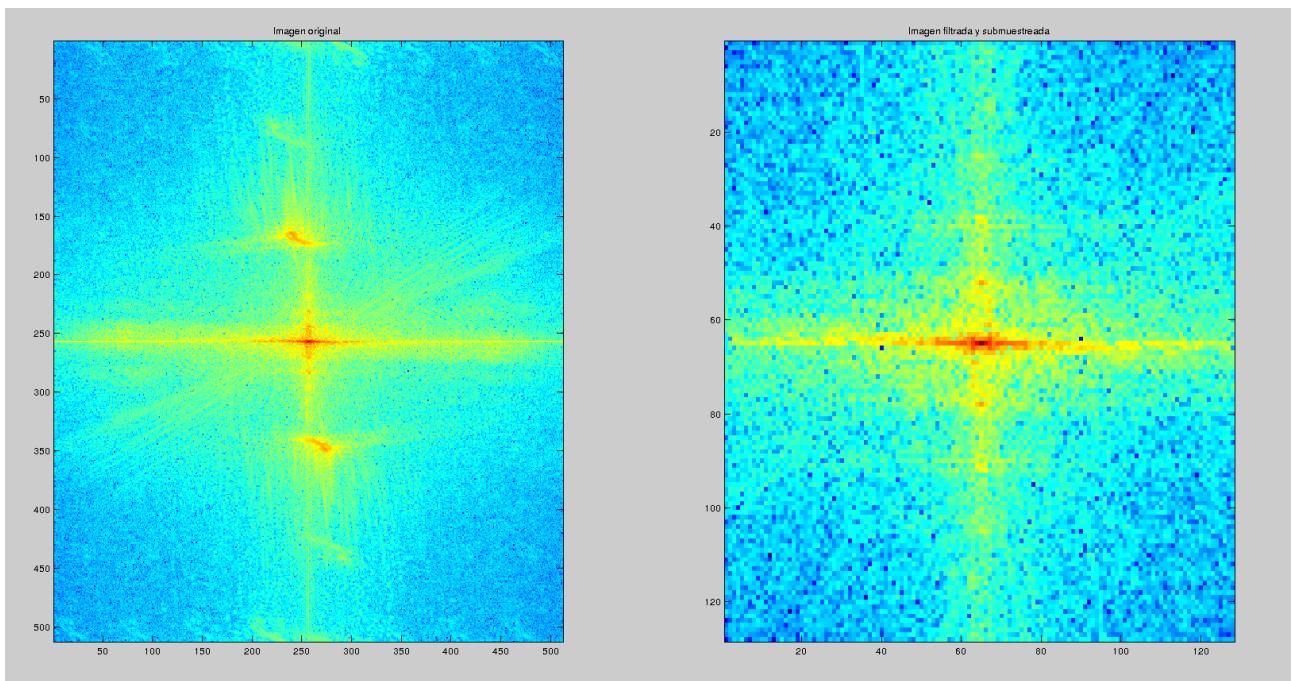


c) Al submuestrear a un cuarto de la frecuencia se está multiplicando, en el espacio, con un peine de deltas. Eso es lo mismo que convolucionar con otro peine de deltas en frecuencias, por lo tanto la imagen de la DFT original queda repetida infinitas veces en ambas direcciones espaciales. En este caso interesa la dirección vertical. La nueva frecuencia de muestreo coincide en la gráfica con $512/4 = 128$ píxeles que es menor que el doble de la frecuencia del trapo (eso sería por lo menos $85,3 \times 2 = 170,6$, y posiblemente algo más). Por lo tanto, las frecuencias altas de la “repetición 2” de la forma de la DFT en la vertical, aparecen en la nueva DFT submuestreada. Eso explica además por qué las “comas” de la DFT submuestreada están “invertidas”. Es porque en realidad la que aparece abajo es la que estaba arriba en la siguiente “copia del espectro”, y viceversa.

Por otro lado, si en la imagen original se ven frecuencias hasta los 256 píxeles (mitad de la freq. de muestreo), en la imagen submuestreada se verán hasta los 64 píxeles, y por lo tanto las “comas” originales del trapo no aparecen.

d) Dado que 64 es menor que 85,3 la frecuencia original del trapo no va a aparecer con esa frecuencia de muestreo (eso no se puede evitar). Sin embargo, para evitar que las frecuencias altas de las “copias del espectro” aparezcan, se puede filtrar, antes de submuestrear, con un pasabajos que corte en la frecuencia correspondiente al píxel 64 si se quiere evitar todo posible “aliasing” (cortar en la mitad de la frecuencia de muestreo), o sólo que deje afuera las frecuencias del trapo (aproximadamente filtrando a partir de la frecuencia correspondiente al píxel 78 en la DFT original).

Se filtró a partir de la “frecuencia del píxel 64”, es decir la mitad de la frecuencia de submuestreo (como el píxel 256, o máximo en la imagen original, corresponde a la mitad de la frecuencia de muestreo original, el píxel $256/4 = 64$ corresponde a la mitad de la frecuencia de submuestreo). El resultado se puede ver abajo.



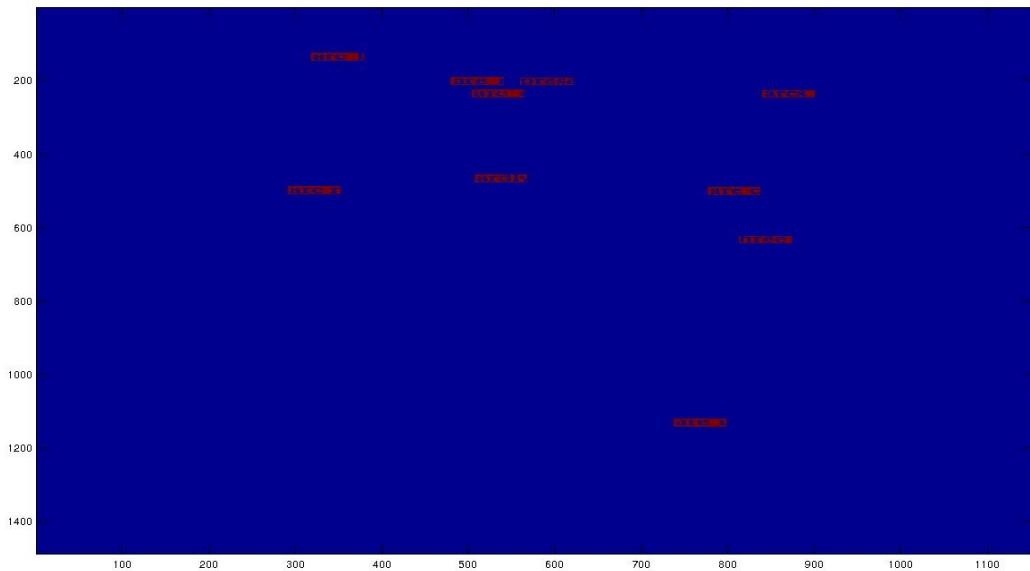
Conclusión

Se logró solucionar el problema del “aliasing”.

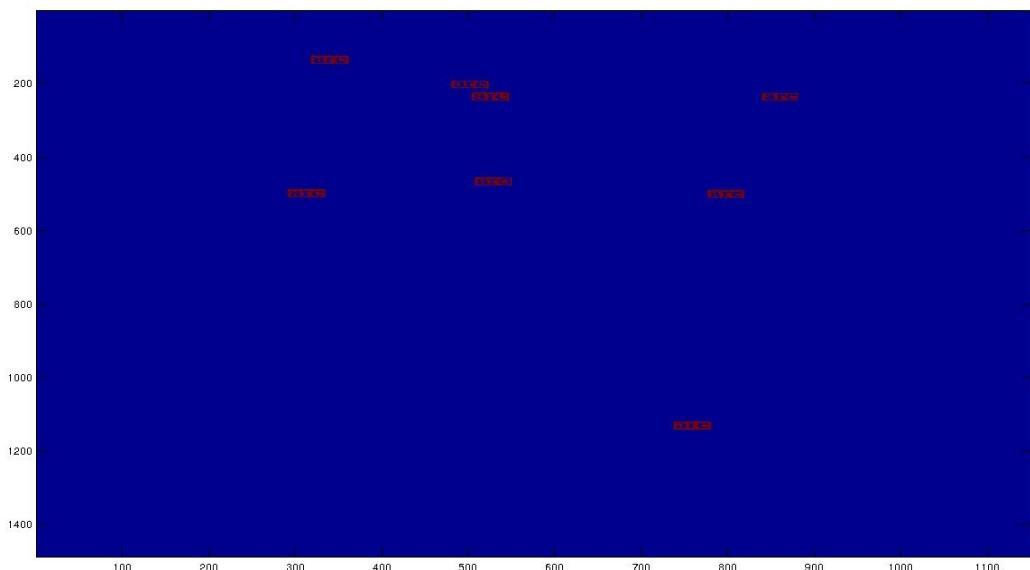
Parte 6)

a) Se recortó una muestra de cada una de las palabras a buscar. Luego se aplicó el filtro de correlación a cada una de ellas, calculando la DFT y convolucionando directamente. Luego, para visualizar el resultado, en las posiciones en que la correlación superara cierto umbral (ajustado “a mano”), se copió el valor del texto original, en una ventana de un tamaño similar a la de la frase/palabra buscada. Los resultados pueden verse abajo:

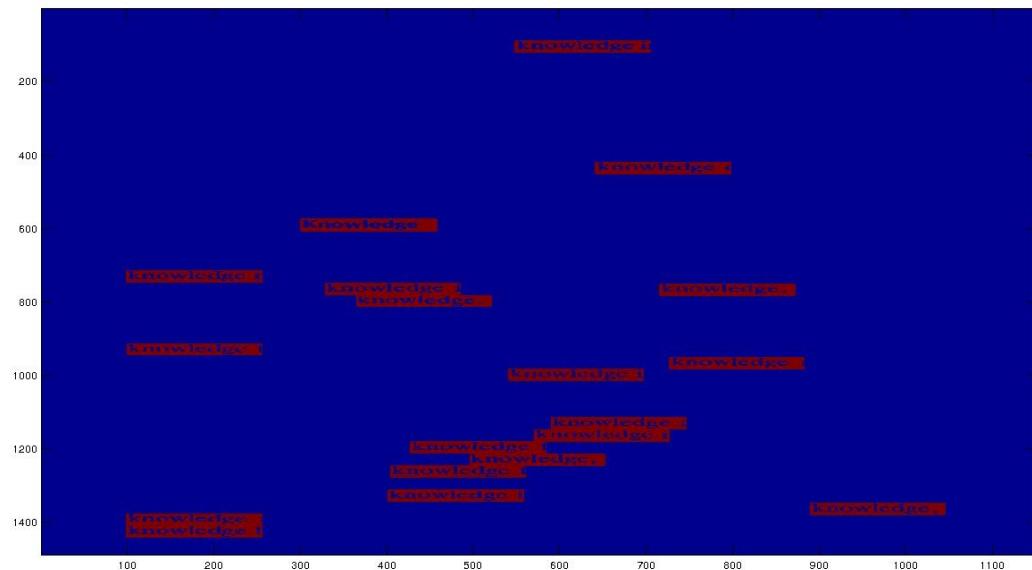
Frase: “are”, Método: DFT



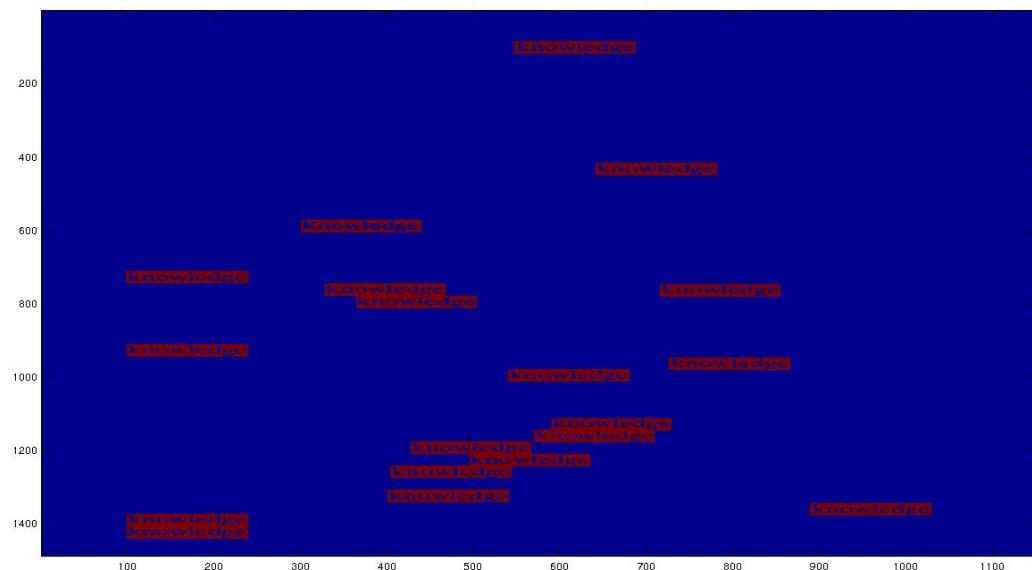
Frase: “are”, Método: Convolución



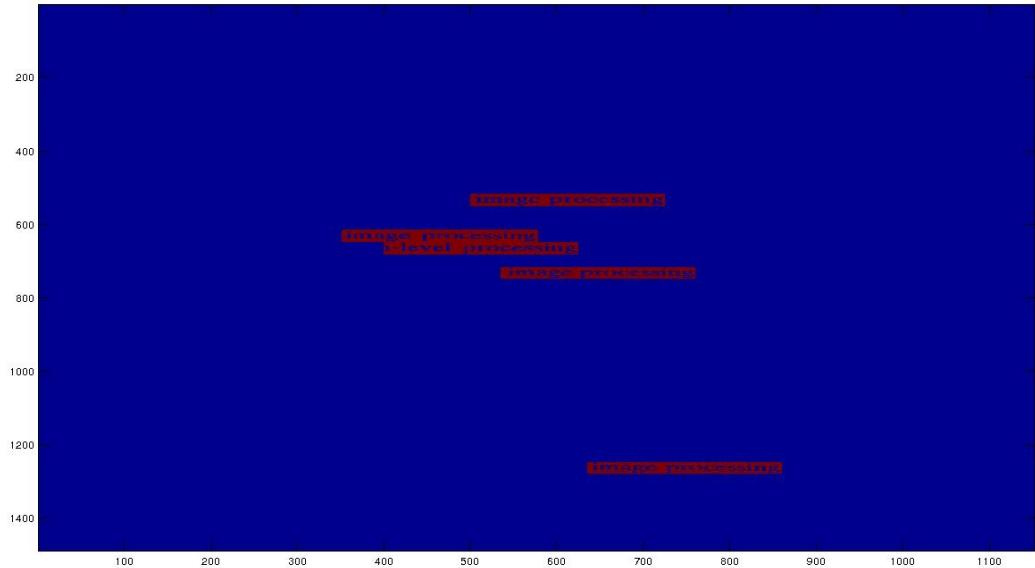
Frase: “knowledge”, Método: DFT



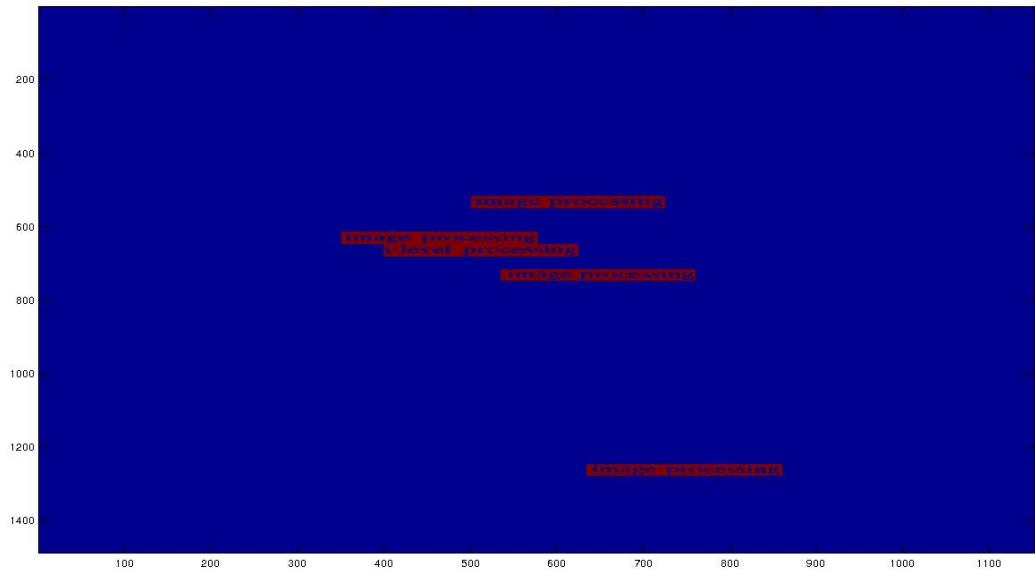
Frase: “knowledge”, Método: Convolución



Frase: “image processing”, Método: DFT



Frase: “image processing”, Método: Convolución



b) Tiempos

Frase: “are”) DFT: 0,3s Convolución: 20,9 Relación: 70 veces más

Frase “knowledge”) DFT: 0,2s Convolución: 60,2 Relación: 300 veces más

Frase “image processing”) DFT: 0,2s Convolución: 98,2 Relación: 491 veces más

Conclusiones:

Se implementó un filtro de correlación utilizando transformada DFT y convolución directa. Ambos métodos dan resultados bastante buenos. En el caso de “are” hay algunos falsos positivos, pero se ve que vienen de frases “similares” visualmente. Hay un falso positivo menos al usar convolución directa.

El caso “image processing” presenta un falso positivo tanto con DFT como con convolución. El caso “knowledge” parece detectar correctamente todas las apariciones sin falsos positivos. Los dos métodos dan resultados muy similares. El tiempo para hacer la convolución directa es mucho mayor que utilizando DFT.

