Tratamiento de Imágenes 2016
Miguel Tasende
C.I.: 3.835.258-4

Práctica 3

1) a) Se implementó un detector de borde con la transformada de Sobel (“sobel.m”), que además umbraliza la imagen, dejando las p por unidad intensidades mayores.
Para calcular y mostrar el resultado de la transformada de Hough, se implementaron las funciones:
- “houghM.m” que devuelve rho, theta, y la matriz de la Transformada de Hough
- “picosHough.m” que devuelve los índices de rho y theta donde se dan los máximos de la T. de Hough, dados un umbral y un máximo de picos.
- “dibujarLineas” que dibuja las líneas en el plano “xy” dados los valores de rho y theta.

Primero se probó con la imagen “oclusion.bmp”, y los resultados se muestran abajo, para un umbral alto y bajo, respectivamente.

Imagen “oclusion.bmp” con umbral alto)

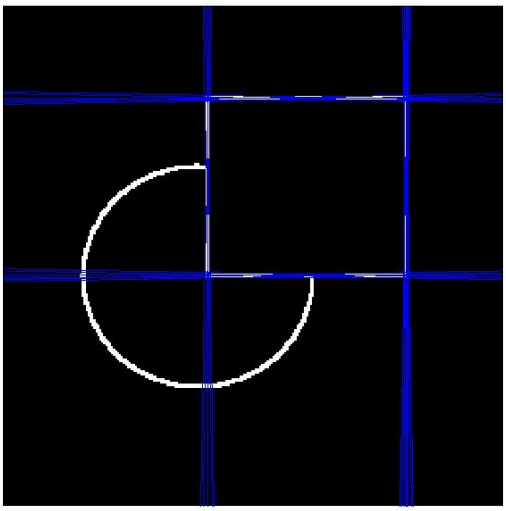
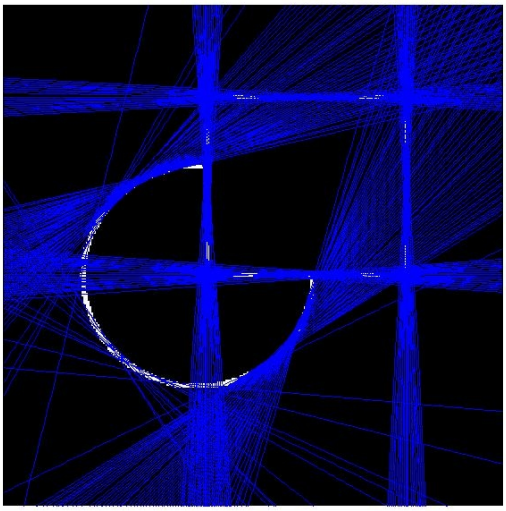


Imagen “oclusion.bmp” con umbral bajo)

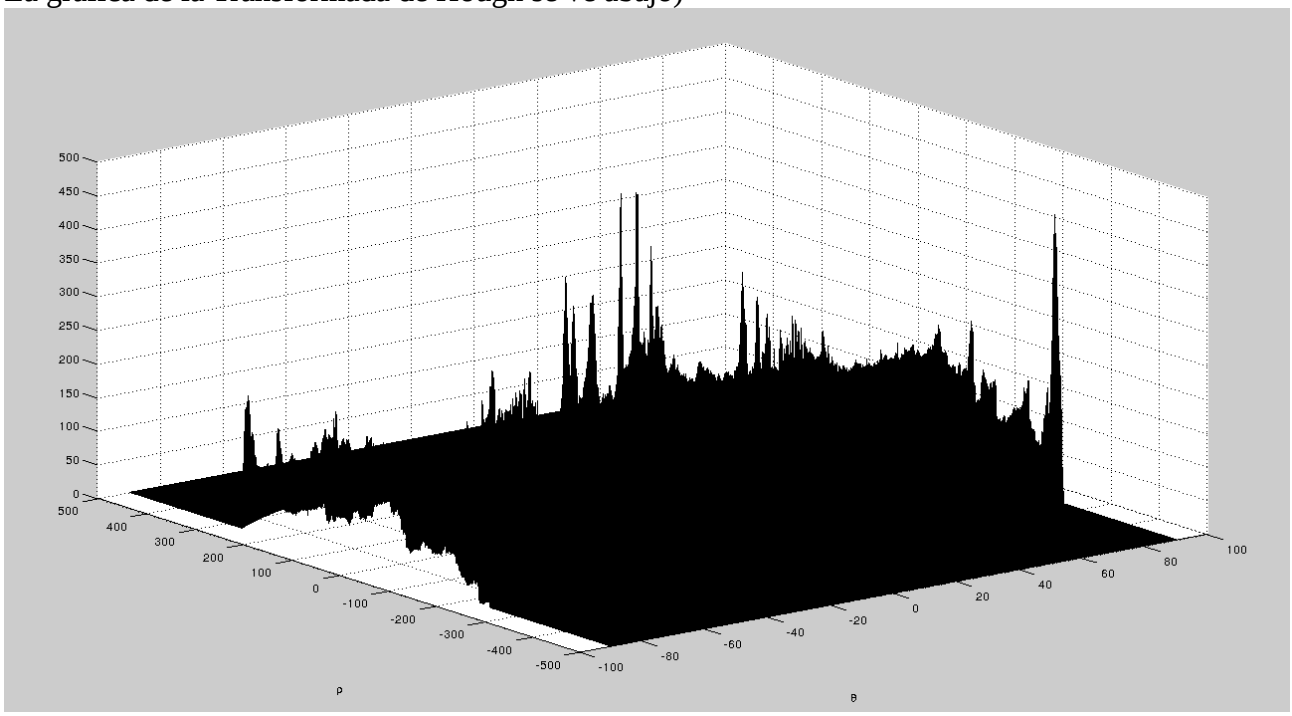


En el caso del umbral bajo, se ve que también se incluyen la lineas “tangentes” a la circunferencia.

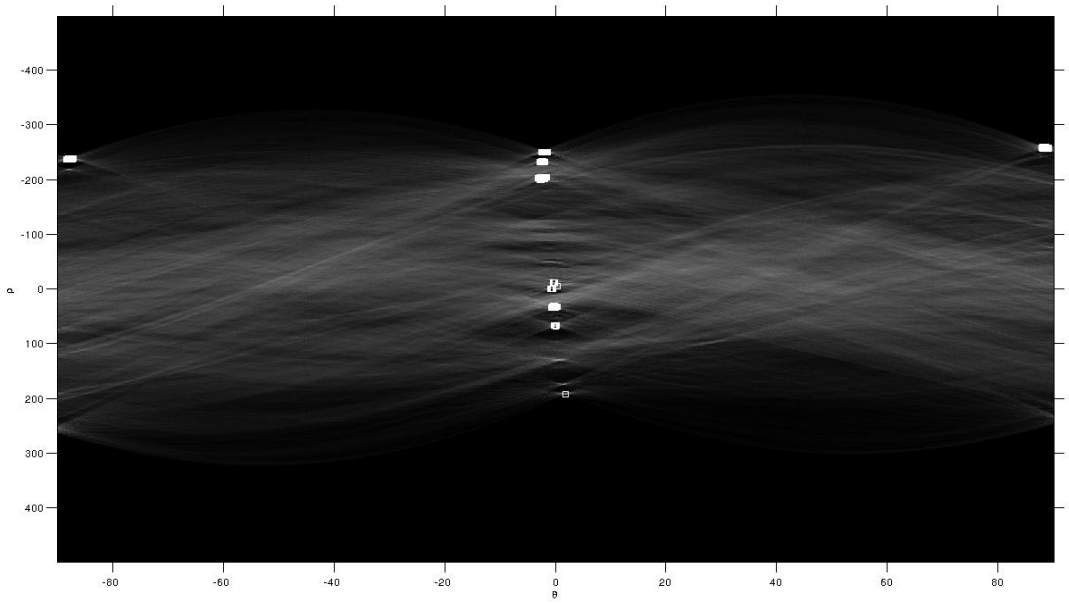
Después se probó con una imagen de cédula. El resultado es el de abajo:

Al aplicar el filtro de Sobel con umbral 0,1)

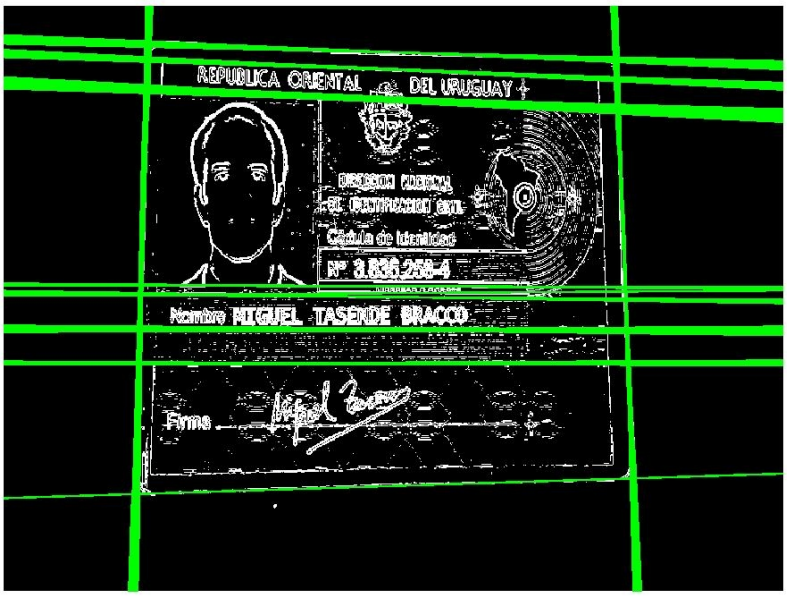
La gráfica de la Transformada de Hough se ve abajo)



La gráfica de las sinusioides resultantes en el plano theta-rho, con sus puntos de corte principales)

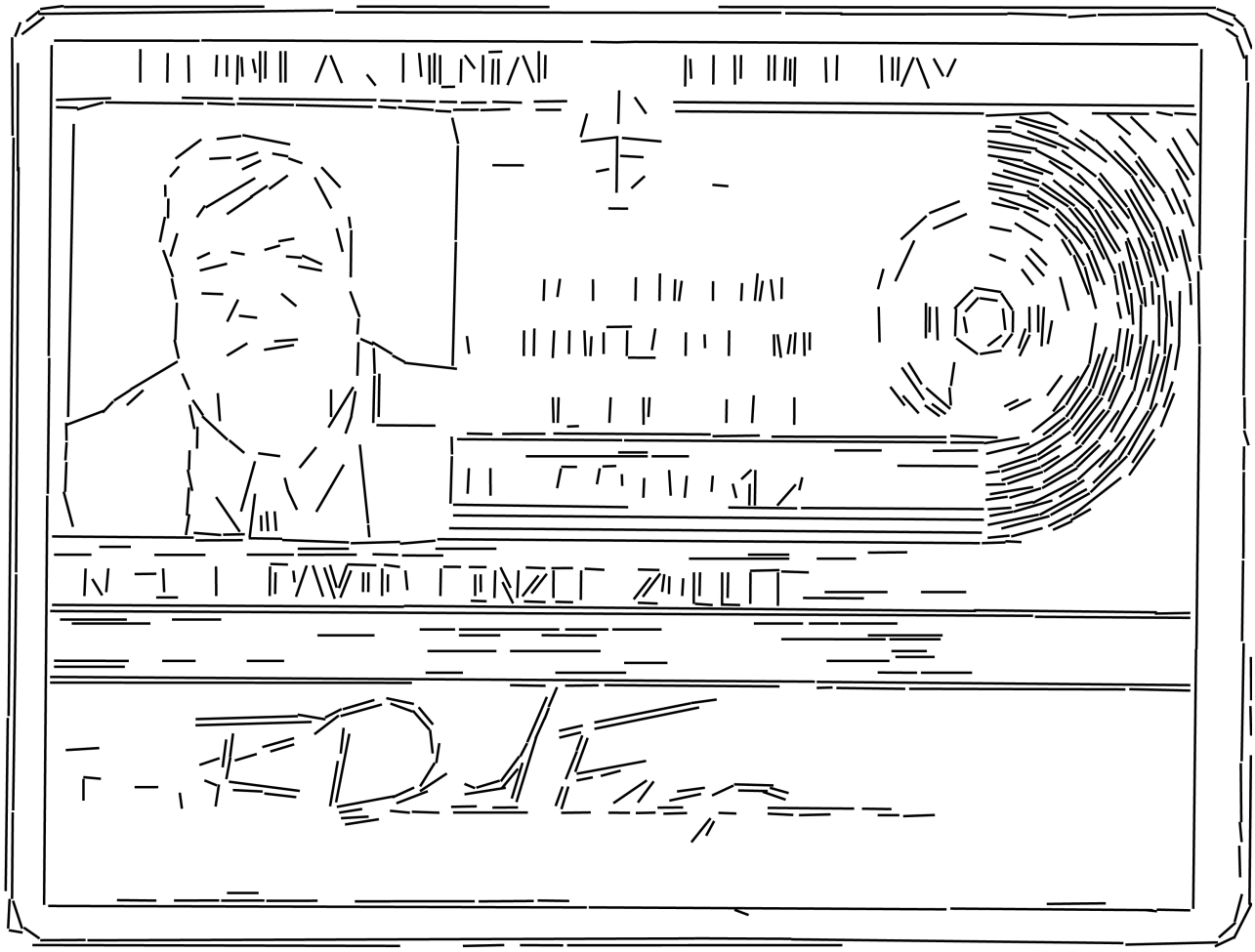


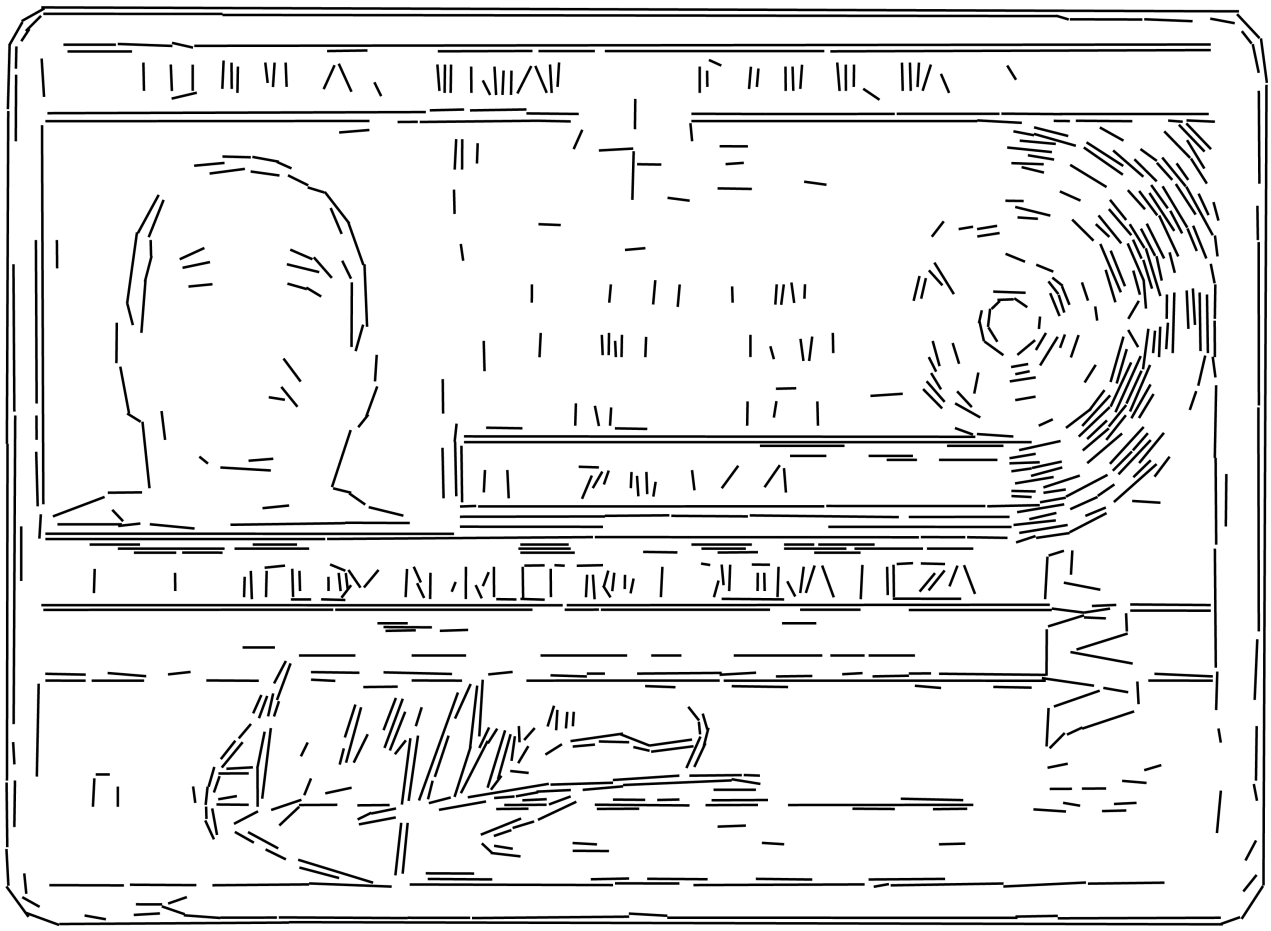
Y por último las líneas detectadas, dibujadas sobre la imagen “original” (ya filtrada por Sobel):



b) Los efectos de cambios en el valor del umbral se entienden fácilmente. Si el umbral es más alto, considero más potenciales rectas. Si el umbral es más bajo, considero menos. Hay un detalle a tener en cuenta y es que conviene saber si la limitante en la cantidad de rectas en una imagen particular, se consiguió por el umbral o por el máximo permitido. En el caso del “máximo permitido”, la implementación de la función “picosHough” se encarga de que se devuelvan los N picos máximos. Los casos de dTheta y dRho son totalmente distintos.
dRho representa una “tolerancia a errores”: si dRho es muy chico, es muy difícil encontrar rectas largas (ya que en un recorrido largo, siempre hay más curvatura y los errores de representación y operaciones también se magnifican), y por lo tanto se tienden a encontrar sólo rectas chicas (las grandes “aparecen”, pero sólo como un montón de rectas chicas indistinguibles de pares o tríos de píxels que justo quedaron alineados, por lo tanto las probabilidades de encontrar una recta “real” son bajas). Si dRho es muy grande, entonces “cualquier cosa parece una recta”, y se va a “detectar” una recta que no pase por “nada especial” (sólo un promedio de un “clúster” de puntos sin forma muy específica). Lo ideal sería intentar que dRho sea del orden del ancho de las rectas que se quiere detectar (en este caso se usó dRho = 1).
dTheta determina la cantidad de puntos de ángulo a probar. No representa “un segmento de tolerancia”, sino que influye en una cantidad de valores discretos. Cuantos más ángulos se pruebe hay mejores chances de encontrar la dirección correcta de una recta. Lo ideal sería utilizar dTheta lo más chico posible. La limitante es el costo computacional. Por otro lado un dTheta tan chico que coincida en los mismos puntos para valores de rho constantes, no aporta nada (tampoco empeora, “la calidad” de las rectas encontradas). El problema práctico al hacer dTheta muy chico (además del costo computacional), es que es muy probable que “se encuentre la misma recta muchas veces”, y por lo tanto se dan menos oportunidades a otras rectas, si se limitó la cantidad máxima a detectar. Por lo tanto hay un “dTheta” óptimo dado un dRho, que es el que logra que “las rectas se detecten una sóla vez” (o lo más cercano a eso posible).

2) Se utilizó el algoritmo LSD como estaba en la página de “IPOL”. Los resultados pueden verse abajo, para dos cédulas encontradas por internet.

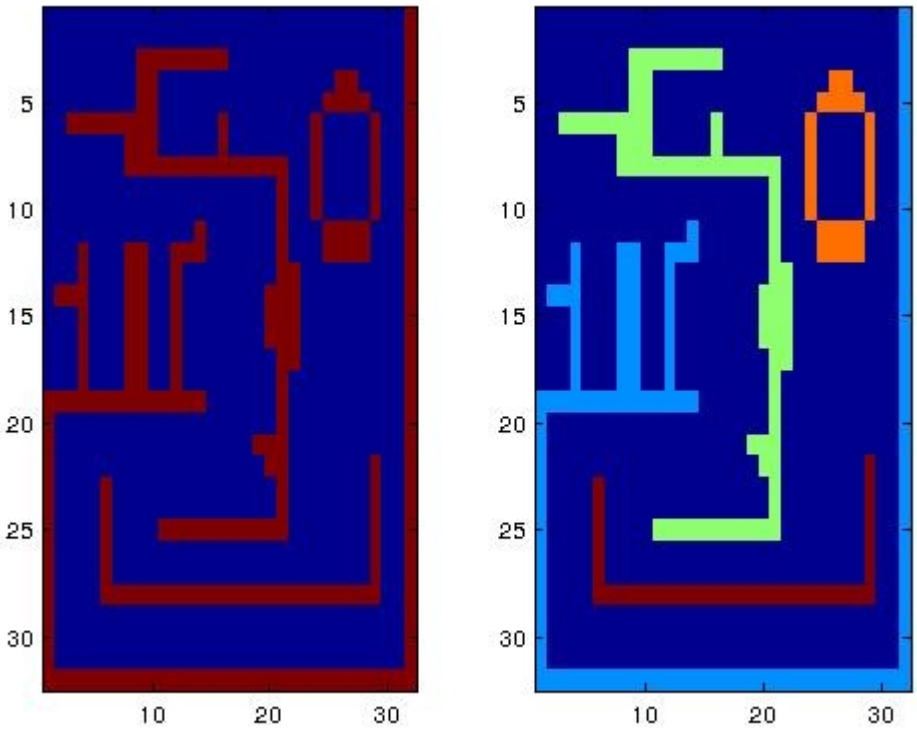




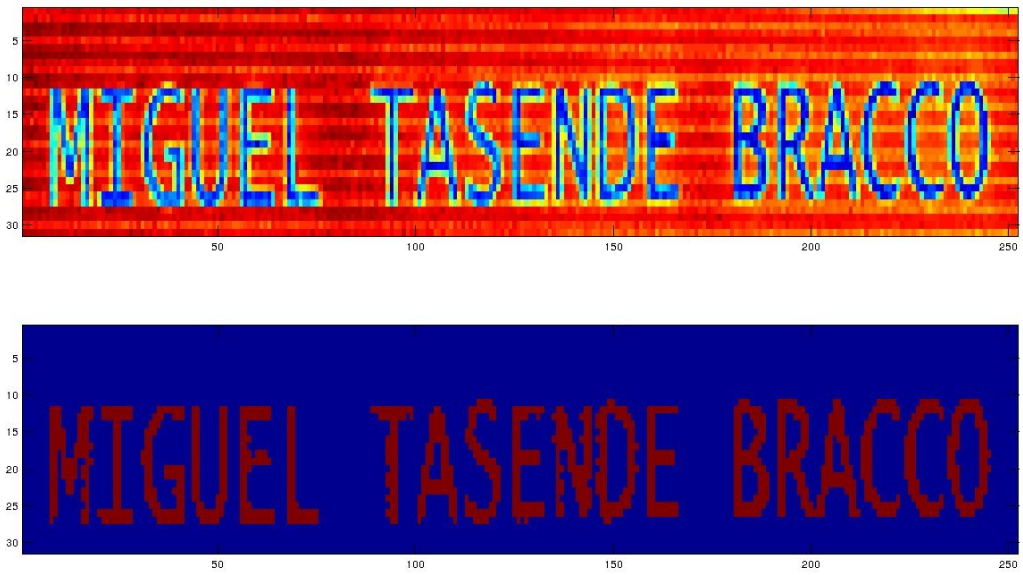
Se puede ver que este algoritmo detecta una gran proporción de los segmentos existentes, que no detecta segmentos “dos veces”, y que además detecta bien tanto segmentos largos como cortos. Todo lo anterior se puede lograr con la transformada de Hough, aunque para hacerlo hay que ajustar muy bien los parámetros de dRho, dTheta, y el umbral, a la imagen particular, y así y todo, es muy posible que sea imposible lograr todo al mismo tiempo (tener muchos segmentos cortos y largos, por ejemplo, podría implicar tener muchos “repetidos”, y no es claro como ajustar los parámetros para conseguir todos los objetivos al mismo tiempo).

3) Se implementó la función “etiquetar.m” que realiza el etiquetado de regiones, dada una imagen en 0s y 1s (devuelve la imagen etiquetada). Se pueden usar “4 vecinos” u “8 vecinos” con una opción de entrada.
Como funciones auxiliares, se implementó:
- “vecinos.m” a la que se le pasa la ventana de vecinos del píxel en cuestión y devuelve la etiqueta a asignar, las relaciones nuevas, y si se creó una etiqueta nueva.
- “aplicarRelacion” sencillamente lleva los elementos que tenían una serie de valores de etiqueta a un valor pedido.
- “equivR” es la función más interesante que se implementó. Dada una relación simétrica cualquiera, impone la condición de transitividad y devuelve las clases de equivalencia resultantes (hay que pasarle cuántas etiquetas posibles hay). Utiliza una función recursiva “buscarClase” que explora el arbol de relaciones (de forma bastante eficiente, yo creo). Para esta parte se me planteaba la opción de resolver las relaciones recorriendo la imagen, o resolverlas directamente entre las equivalencias. Considero que la segunda forma (que es la implementada) consigue un tiempo de ejecución mucho menor, si la cantidad de etiquetas no es muy grande (en particular si es mucho menor que la cantidad de píxels). En el caso de tener muchas etiquetas posibles, eso podría no ser cierto.

Abajo se puede ver el resultado:



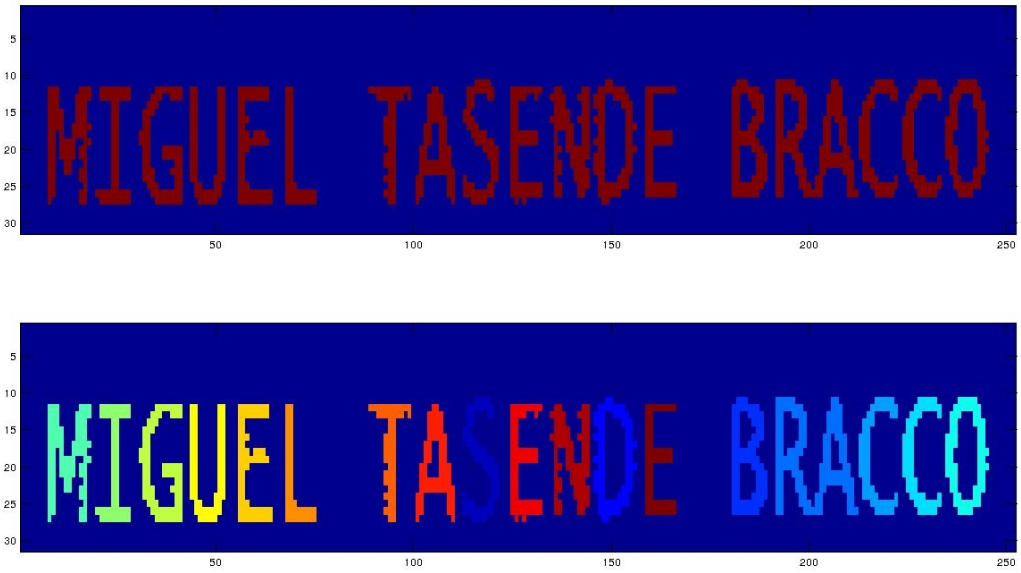
4) a) Se implementó el algoritmo de Otsu, en la función “otsu.m”. Abajo puede verse la comparación de la imagen original con la imagen filtrada (usando “imagesc”):



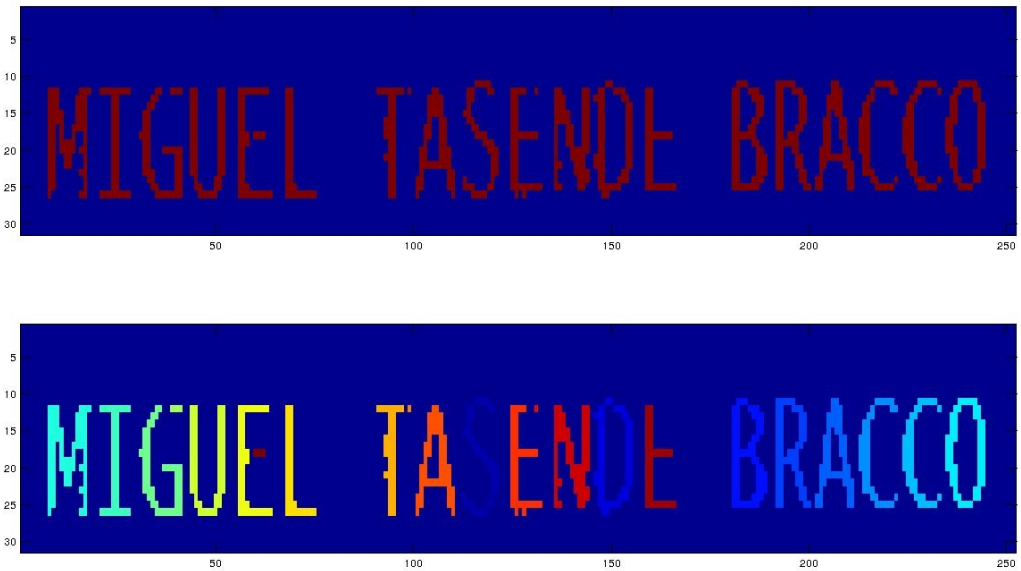
b y c) Luego se utilizó la operación “imerode” para conseguir la erosión de la imagen. Se utilizaron dos elementos estructurantes distintos (después de probar varios, de variar los tamaños y de probar también con imopen, sin mejores resultados). Uno es un cuadrado de lado = 1, y el otro un

rectángulo de 1 de ancho y 2 de altura (se eligió así para separar las letras). En general, se pudo ver que los elementos de mayor tamaño logran separar las letras, pero “separan demasiado”, destruyendo a las letras mismas. En este caso, están todas separadas originalmente, excepto las que forman “RA”. Después de la erosión, se etiquetó el resultado. Los resultados están abajo:

Caso con un elemento estructurante cuadrado de lado = 1 (la “RA” está unida; el resto de las letras está bien):

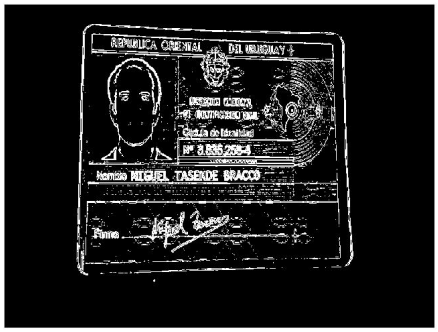


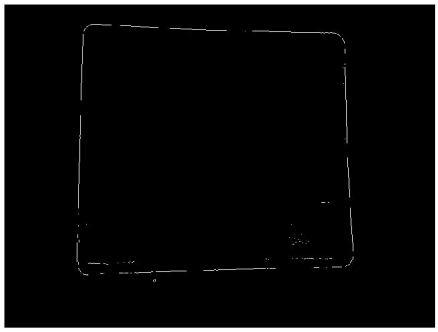
Caso con un elemento estructurante rectangular de ancho=1, y altura=2 (la “RA” se separa, pero algunas otras letras quedan cortadas).



En este último caso se ve que las letras “cortadas”, pierden un poco de su forma, pero además aparecen nuevas pequeñas regiones. Una forma de eliminarlas sería establecer un umbral de píxels para las regiones aceptadas, o quedarse con las N más grandes, si uno conoce de antemano cuántas son. Eso se implementó como parte del próximo ejercicio, para la detección de números.

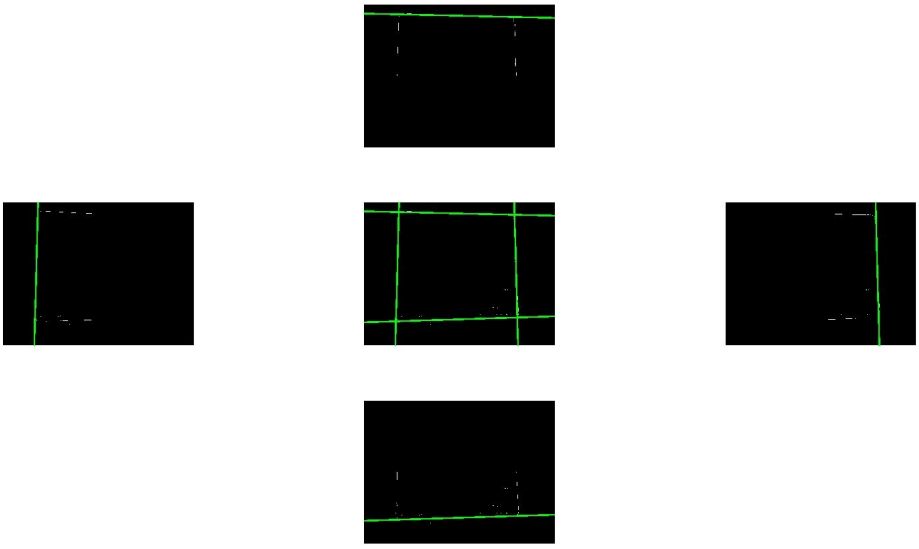
5) a) Para la parte “a” de este ejercicio se utilizaron las funciones “sobel”, “houghM”, “picosHough”, “dibujarLineas” de los ejercicios anteriores. Además se implementaron las funciones “interseccion” y “contorno”, que se encarga de encontrar el contorno de la imagen de la siguiente manera: se consideran “rayos” horizontales y verticales, uno por píxel, y se busca en qué pixel cortan primero partiendo desde el borde. El resultado de aplicar “sobel” y después “contorno” es el de abajo:





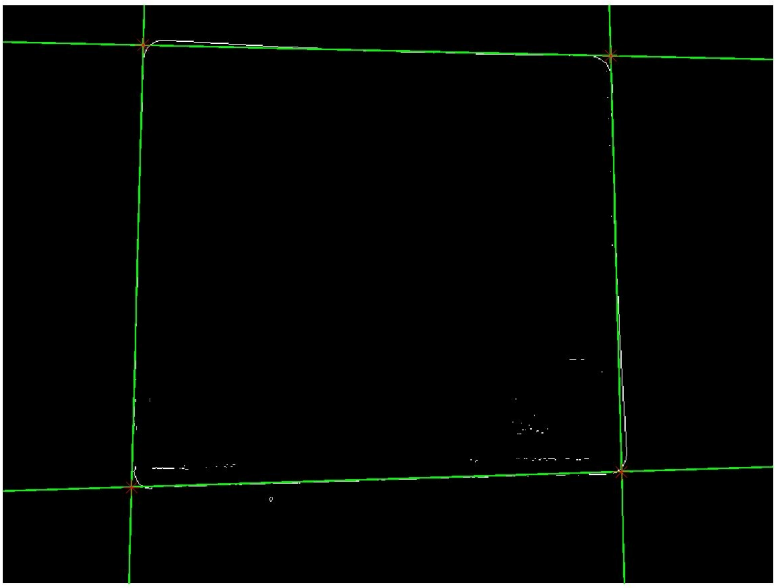
Luego se va a aplicar la transformada de Hough, pero antes se divide la imagen en 4 partes (mitad izquierda, mitad derecha, mitad de arriba y mitad de abajo). En cada una de esas partes se aplica “hough” por separado, teniendo en cuenta, además las características de la recta que se pretende encontrar.
En la mitad de arriba y en la mitad de abajo, se busca una recta sóla cada vez, que esté entre -35º y 35º.
En la mitad izquierda y en la derecha, se busca una sóla recta cada vez, que esté entre 55º y 125º.
De esa forma se cumple con el requisito de “inclinación máxima de 30º”.
La división en partes, es para tener seguridad de que no: “se confunde la recta de arriba con la de abajo”, “se confunde la de la derecha con la de la izquierda”, “se detectan dos rectas de abajo y ninguna de arriba”, etc.
La limitación del ángulo asegura que la recta que se detecta es la que se quiere (hay que tener en cuenta que en cada parte aparecen 3 rectas “grandes”, por lo menos, y hay que “filtrar” las direcciones perpendiculares a la deseada).
El resultado puede verse en la próxima imagen.

En las cuatro regiones con una recta se ve la detección, y en el centro se unieron todas las rectas detectadas:



Luego de detectar las rectas se buscan los vértices de forma analítica (se conocen las ecuaciones de las aristas). Para eso se implementó la función “intersección”. El resultado se ve abajo:

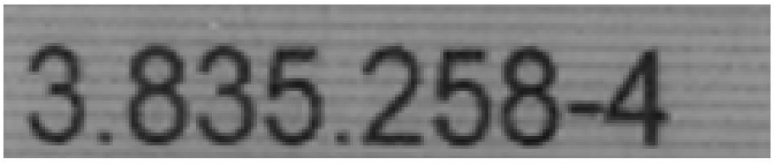
Se pueden ver los vértices como asteriscos rojos:



Después de detectados los vértices, se buscó la homografía que los lleva a cuatro puntos predeterminados, y se le aplicó esa homografía a la imagen. Inicialmente se utilizaron las funciones del entregable 1, y funcionaron bien, pero lento. Por lo tanto se utilizaron finalmente, las funciones de Matlab (“cp2tform”, “imtransform”).
El resultado se ve abajo:

Se ve que el pasaje a un tamaño y posición estándar funciona muy bien, y eso se repite con todas las cédulas y posiciones que se probó.
Un dato interesante es que se comprobó que la performance de las partes siguientes depende en parte del tipo de interpolación que se use en la homografía. Usando el “vecino más cercano” muchas veces, después no se pueden separar los números sin “destruirlos”. La interpolación bilineal y la bicúbica dan buenos resultados.

b) Esta parte consiste únicamente en quedarse con una zona preestablecida de la imagen anterior. La mayor dificultad que se tiene en esto es que los números están muy cerca de una línea inferior, y no siempre están exactamente en la misma posición “izquierda-derecha”. Eso afecta la performance entre distintas cédulas, pero no con la misma en distintas posiciones. El resultado se ve abajo:



c) Esta es la parte más larga. Inicialmente se intentaron dos estrategias.

Antes que nada, se tenían 3 cédulas. La primera se rotó en varias posiciones. De las otras dos, una de ellas tenía los bordes desgastados y, más importante, los números escritos “a máquina” y no con computadora-impresora. Por esa razón no se esperaba una gran performance en esa detección.
Los templates se sacaron de las cédulas. Mayormente de la primer cédula, aunque las cifras faltantes se sacaron de las otras.

En cualquier estrategia (y en la que se siguió) inicialmente se filtra con un umbral calculado según el algoritmo de Otsu, y se busca erosionar la imagen hasta separar los números. Se vio que el nivel de erosión necesario era muy variable entre imágenes (incluso de la misma cédula), por lo que se terminó implementando una función (que será la principal) que erosiona “según las necesidades” (se explica más abajo). La función es “erosionAdapt.m”, y realiza varias otras operaciones.

La primera estrategia, más sencilla, consiste en asumir que los números están en posiciones fijas dentro de la “zona de números”, y utilizar ventanas para separar cada uno. Esta estrategia tiene la ventaja de que es robusta frente a pequeños movimientos de la posición de los números. En el caso de distintas rotaciones de una misma cédula, la detección se logró (después del matcheo) con 0% de error. Sin embargo esta idea se abandonó, cuando se vio que las distintas cédulas varían mucho la posición inicial de las cifras, dentro del área de números. Se puede suponer que esto se debe a que se está tomando el “borde del plástico” y que el “cartón interno” puede estar corrido respecto del plástico. Se podría intentar detectar el cartón directamente y en ese caso es muy posible que sea una buena estrategia (para cédulas realizadas con computadora e impresora; no son todos los casos). Luego de algunas pruebas esta estrategia se descartó.

La segunda estrategia (que es la utilizada) consiste en detectar las posiciones de los números etiquetando regiones. Esta estrategia tiene la ventaja de que, cuando funciona correctamente, los números a comparar están “muy claros”. Tiene algunas desventajas:

1) No es muy robusta frente a algunos pequeños cambios. En particular si dos números no son separables, la estrategia falla completamente y no es posible identificar el número (se podría usar la otra estrategia o variantes [como detectar antes posiciones con rayos horizontales y estadística], posiblemente, pero no se hizo en esta implementación). Eso pasó en el caso de la cédula 4 analizada (no se detecta ningún dígito bien porque las regiones resultantes no son dígitos).

2) Los dígitos aparecen inicalmente “desordenados” (no se sabe qué región corresponde a qué número de dígito). Esto no es un problema grave, sólo requiere un poco más de complejidad pero se solucionó en esta implementación.

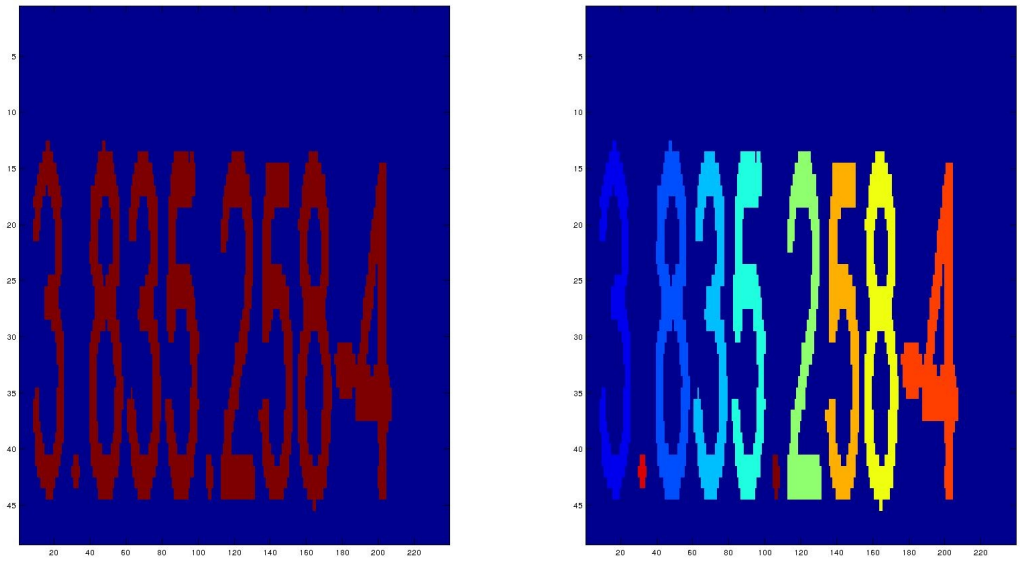
3) Puede haber regiones que no sean dígitos. Esto, en el caso en que los dígitos están presentes y separados, se soluciona tomando las 8 regiones más grandes de la imagen (no parece razonable que haya una región de ruido más grande que un dígito en la zona de los números). Es lo que se hizo, y da resultado (siempre que se puedan separar las 8 regiones de los dígitos).

La implementación depende, de forma muy fuerte de la función “erosionAdapt”. Esta función toma la imagen de la región de números, luego del filtro con umbral de Otsu, y hace lo siguiente:
Erosiona con un elemento “line”, de ancho 0.
Etiqueta las regiones con la función “etiquetar” del ejercicio 4, y se queda con las 8 regiones más grandes. Entre esas 8 hace el siguiente test, que está pensado para que no hayan “dos o tres números pegados”. Se fija si la octava región en tamaño, tiene un tamaño que se encuentra entre p1 y p2 veces del promedio de las 7 regiones más grandes. De esa forma, se asegura que la región más chica

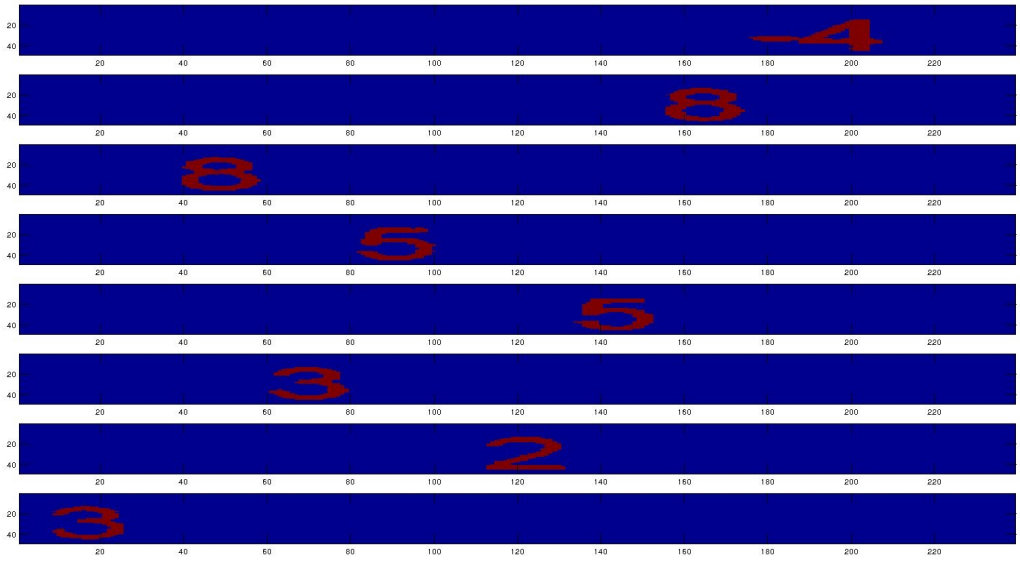
de las 8 contiene un dígito, y por lo tanto las 8 también (se asume que no hay más de 8 regiones que puedan cumplir esa característica en la “región de números”).
Si no se consiguió pasar el test, se aumenta la erosión a 1, a 2, y así sucesivamente, hasta llegar a 20.
Si se llega a “erosión=20”, se termina el algoritmo y se devuelve el “PlanB” (que fue guardado antes). Este consiste en devolver la primer dupla “imagen-etiquetas” que tenga por lo menos 8 regiones etiquetadas. No se garantiza que esas 8 regiones sean dígitos, pero se asume que es mejor probar suerte, que nada.

De las 6 imágenes de cédula probadas, en 5 este algoritmo funcionó correctamente.
Los resultados de un caso se ven abajo:

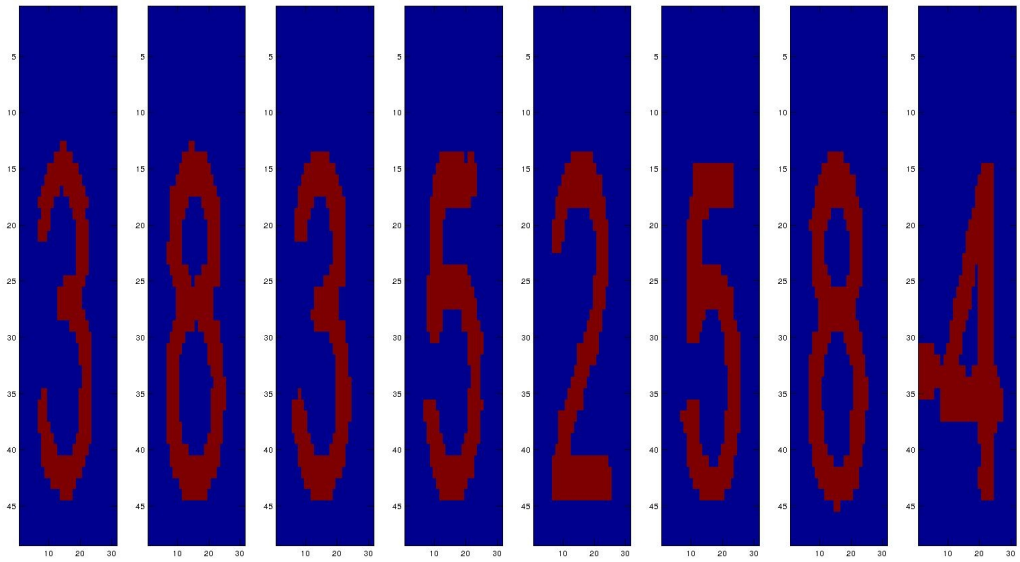
Primero, la imagen después de erosionar y etiquetar todo:



Abajo se ven las regiones seleccionadas como “dígitos” (notar que están desordenadas):



Dado que las cifras están desordenadas, se implementó la función “marcoCifra.m”, que realiza dos operaciones (que además son útiles para generar “templates”). Busca el “centro de masas horizontal” de la región en la imagen, y lo devuelve, así como una imagen centrada en ese centro y de ancho 30 píxels (quedan todas estandarizadas igual que los templates). Según la posición del centro de masas se ordenan las regiones, y el resultado es el de abajo:



Por último, sólo queda hacer el “matching” entre esas cifras y los “templates”. En cuanto a los “templates” se implementó una función “conseguirCifras.m” que ayuda a extraerlos de la cédula.

Para el “matching” se consideraron algunas funciones, todas basadas en la correlación. Finalmente se terminó usando la función “matchear.m” que había sido implementada para el Entregable 1, debido a que dio mejores resultados que “diferencia.m”, que se implementó en este.
“diferencia.m” calcula la suma de los valores absolutos de las diferencias entre píxels, entre el template y la cifra. Hace el cálculo moviendo el template sobre la cifra a todas las posiciones de píxel posible y se queda con la menor diferencia encontrada. Como la performance de “matchear” era mejor se terminó usando esta última.
Un problema inicial que se detectó en “matchear” es que todos los “3” se detectaban como “8”. Se identificó que esto pasaba porque el “3” está incluido en el “8”, y dada la codificación “0/1” de la imagen, no se estaba penalizando “la región sobrante del template”. Para evitar ese problema se centró la codificación de la cifra y el template en cero (valores -1 para fondo, 1 para “cosa”). Eso solucionó el problema.
El resultado fue el de abajo:

```
cifraMatch =

     3     8     3     5     2     5     8     4
     3     8     3     5     2     5     8     4
     3     8     3     5     2     5     8     4
     7     7     7     7     7     7     7     7
     9     9     9     0     9     9     0     4
     4     7     4     2     3     7     9     8

>> resultadoEsperado

resultadoEsperado =

     3     8     3     5     2     5     8     4
     3     8     3     5     2     5     8     4
     3     8     3     5     2     5     8     4
     3     8     3     5     2     5     8     4
     3     9     8     0     8     5     8     4
     4     7     4     2     3     7     9     8
```

La cédula 5 (están por filas, la última columna es el dígito verificador) tiene los dígitos escritos a máquina, y los errores se deben a “problemas de matcheo” (todo el proceso anterior funcionó bien). En esa cédula se detectaron 3 cifras bien de 8, pero una mala performance era esperable.
La cédula 4 es la que tuvo el problema previo, en la detección de regiones. Es igual a la “cédula base” (de la que se sacaron los templates y los datos de posición iniciales), pero rotada. El problema no fue de matcheo, sino de que no fue posible separar algunos dígitos.
Las primeras 3 cédulas son rotaciones de “la base”, y funcionan bien en todo el proceso.
La última es una cédula distinta a la base (pero escrita en computadora-impresora) y funciona bien en todo el proceso.

Claramente es un conjunto muy chico para determinar la performance de los algoritmos, pero se puede ver que, excepto por el problema de la separación de dígitos (que es detectable por el programa, sólo que no se implementó una buena solución alternativa), funcionan bien (detectar números de máquina de escribir con templates de computadora no estaba en los planes, y seguramente requeriría un sistema de reconocimiento de dígitos mejor).