Automatizar la creación de un mosaico de fotos

El objetivo de este proyecto es escribir las rutinas para automatizar la composición de un mosaico de fotos. Los datos de partida son un conjunto de 12 imágenes en color (fc_torre01.jpg, ..., fc_torre12.jpg) que forman un mosaico.

LAB 1: (Obtención de keypoints en imágenes usando el algoritmo SIFT)

Lo primero es encontrar puntos relevantes en las imágenes que puedan servir para emparejarlas si los detectamos en varias de ellas. Usaremos un algoritmo llamado SIFT (Scale-Invariant Feature Transform). No lo implementaremos, sino que usaréis el ejecutable suministrado (siftWin32.exe, solo para Windows), cuyo uso es:

siftWin32 <bw.pgm >data.txt (bw.pgm=imagen entrada data.txt=datos salida)

Podemos ejecutarlo desde MATLAB con `system('siftWin32 <bw.pgm >data.txt');`
El comando system() recibe una orden (como una cadena de texto) y se la pasa al SO para que la ejecute como si se hiciera desde la línea de comandos.

El programa siftWin32 recibe una imagen (monocroma y en formato pgm) y detecta puntos relevantes (keypoints) susceptibles de ser emparejados con puntos similares en otras imágenes. La información de estos puntos se vuelca en un fichero de texto. Al ser nuestras imágenes en color y formato jpg, lo primero que hay que hacer para procesarlas es pasarlas a niveles de gris y guardarlas en formato pgm:

```
im=imread('fc_torre01.jpg'); bw=rgb2gray(im); imwrite(bw,'bw.pgm');
```

Tras ejecutar SiftWin32, usad la función suministrada **`fc_info_puntos('data.txt')`**, que lee el fichero 'data.txt' y devuelve los datos en una estructura con campos:

.xy = matriz Npx2 con coordenadas x,y de los Np keypoints hallados en la imagen.
.id = matriz de Np x 128. Cada fila es un vector de 128 identificadores describiendo cada uno de los Np keypoints.

Esta función también elimina parte de los puntos detectados por SiftWin32 para quedarnos con un número más razonable (del orden de unos pocos centenares de puntos por imagen, en vez de miles) y facilitar así el posterior procesado.

En este primer ejercicio extraeremos la información de los puntos más significativos de todas las imágenes y la guardaremos para trabajar luego con ella. Para ello:

1. Definir name='fc_torre' y NF=12 (prefijo de fotos y número de imágenes).

2. Reservar un array de celdas (tamaño NF) con s=cell(1,NF). Un array de celdas es como un vector con la diferencia de que en cada una de sus "casillas" podemos guardar objetos de tipos o tamaños distintos.

3. Hacer un bucle barriendo todas las imágenes. En cada paso:

   - Usad **`fich=sprintf('%s%02d.jpg',name,k);`** para crear automáticamente el nombre del archivo de cada foto, Leer la imagen original, pasarla a BW y guardarla en formato PGM ('bw.pgm') como se ha indicado antes.

   - Ejecutar siftWin32 desde MATLAB con system().

   - Usar fc_info_puntos('data.txt') para leer la información de los puntos encontrados por el algoritmo. Guardar los resultados en s{k}.

Al terminar tendréis un array de celdas donde cada elemento s{1},…,s{12} es una estructura con la información (posición xy + vector descriptor id) de los keypoints detectados en cada una de las imágenes fc_torre01, … , fc_torre12.

El nº de "keypoints" detectados en las primeras 4 imágenes debe ser: 309, 138, 72, 107.

Haced un volcado con el nº de "keypoints" encontrados en el resto de las imágenes. Mostrar la 5ª imagen superponiendo sobre ella los "keypoints" encontrados en ella como círculos rojos ('ro'). Adjuntar imagen.

Finalmente hacer **save keypoints s**, para guardar el array s con los resultados en el fichero keypoints.mat. De esta forma, en el resto de la práctica no tendremos que volver a usar SiftWin32.exe (ni trabajar en Windows). Basta hacer **load keypoints** para cargar los datos de s{} y trabajar a partir de esa información.

LAB 2: (detectar posibles emparejamientos entre pares de imágenes)

Se trata ahora de encontrar posibles parejas entre los "keypoints" de las diferentes imágenes (punto de partida para el algoritmo RANSAC que escribiremos luego). Para ello vamos a escribir una función: **`function [xy1,xy2]=find_matches(s1,s2)`**

La función recibe un par de estructuras s1 y s2 (con los datos de dos imágenes) y encuentra posibles emparejamientos entre sus "keypoints". La función devuelve 2 matrices (tamaño N x 2) con las posiciones en la 1ª imagen (xy1) y en la 2ª (xy2) de las N posibles parejas encontradas.

Para estimar si dos puntos en diferentes imágenes corresponden al mismo punto en la realidad usaremos sus vectores descriptores (campo id).

Consideremos el punto 7 de la 1ª imagen, cuyo vector descriptor es d1=s1.id(7,:). y el punto 15 de la 2ª imagen, con descriptor d2=s2.id(15,:). Como criterio de parecido entre ambos puntos usaremos 1/||d1-d2||. La idea es que si los vectores d1, d2 que describen ambos puntos son similares la norma de su diferencia será pequeña y su inversa alta, indicando un alto parecido entre los puntos (posible pareja).

El algoritmo de emparejamiento será una simple búsqueda exhaustiva.

Lo primero es determinar el número de puntos en s1 y s2 (N1 y N2) y reservar un vector **parejas** (inicializado con 0's) de tamaño N2 x 1, para guardar los índices que vamos emparejando. Si parejas(8)=57, es que una posible pareja del punto 8 (en la 2ª imagen) es el punto 57 (en la 1ª imagen). Si al terminar parejas(10)=0, es que no hemos encontrado ningún punto en la primera imagen suficientemente parecido al punto 10 de la 2ª imagen.

Tras reservar el vector anterior hacemos un bucle k=1:N1, barriendo los keypoints de la primera imagen (s1). Para cada punto:

1. Inicializamos M=0 y extraemos el vector ID del k-ésimo punto de 1ª imagen.

2. Barremos (j=1:N2) los N2 puntos de la 2ª imagen. Si parejas(j)>0 es que dicho punto ya está emparejado y lo saltamos sin hacer nada más. Para el resto de los casos calculamos el parecido entre el k-ésimo punto de la lista de s1 y el j-ésimo de la lista de s2 usando el criterio anterior. Si dicho parecido supera el actual valor de M lo actualizamos y guardamos en J el índice en el que estamos.

3. Al terminar hay que decidir si la posible pareja propuesta entre el punto k de la 1ª imagen y el J en la 2ª es lo suficientemente buena. Para ello, usaremos el criterio de que la similitud M encontrada entre ellos debe ser mayor que 4. Si esto se cumple se guarda la pareja haciendo parejas(J)=k (indicando que el punto J de la 2ª imagen es una potencial pareja del punto k de la 1ª imagen).

Tras barrer todos los puntos de s1 tendremos guardados todos los emparejamientos encontrados en el vector parejas:

- Las posición de las entradas no nulas, index2=find(parejas>0), corresponden a los índices de las parejas en la 2ª imagen.
- El contenido de parejas en dichas posiciones (index1=parejas(index2)) serán los índices de sus correspondientes parejas en la 1ª imagen.

Usando index1 podemos obtener las coordenadas (xy1) de los parejas en 1ª imagen (extraídas de s1.xy) y con index2 las coordenadas (xy2) en la 2ª imagen (s2.xy).

Correr find_matches con los datos de la 1ª imagen y todas las demás, hallando el número de parejas encontradas para cada par de imágenes (a partir del tamaño de las matrices xy1,xy2 devueltas). Como referencia, para la 1ª imagen yo obtengo:

(1,2)->41 ptos (1,5)->25 ptos (1,6)->100 ptos (1,7)->31 ptos (1,8)->17 ptos

Para las otras imágenes no encuentro ningún emparejamiento con la imagen 1.

Volcad el nº de parejas encontradas entre la segunda imagen y todas las demás.
Adjuntar vuestro código de find_matches.m

LAB 3: En este ejercicio se usa la función get_proy del LAB anterior. Aseguraros primero de que funciona para el caso de ajuste (hallar la transformación proyectiva cuando tenemos más de 4 pares de coordenadas). Reescribirla en caso necesario.

Probadla con
```
xy1=[200  400  600  300  500; 375  125  375  200  200]';
xy2=[ 87  319  600  200  450; 445   69  375  185  155]';
```

Debéis obtener la matriz P =
```
 1.2921    0.0758 -202.6169
-0.1373    1.3306  -40.4375
 0.0001   -0.0001    1.0000
```

A partir de (xy1) cread una matriz de coord. homogéneas (1ª fila = x's, 2ª fila=y's, 3ª fila=1's). Aplicad P a esta matriz (multiplicando) y volcad las coordenadas u,v obtenidas. ¿Por qué no son iguales que las coordenadas deseadas (xy2)?

Para calcular el error cometido cuando el número de puntos a ajustar es mayor de 4 escribiremos la función: **`function err=error_ajuste(P,xy1,xy2)`**

La función recibe una matriz (3x3) de transformación P, las coordenadas (xy1) del espacio de partida y las coordenadas deseadas del espacio de llegada (xy2) como dos matrices Nx2. La rutina devuelve un vector err (Nx1) con el error del ajuste para cada punto. El proceso es el siguiente:

- Calcular las posiciones obtenidas aplicando P a (xy1) --> (uv).
- Calcular la distancia entre las coordenadas obtenidas (uv) y deseadas (xy2) para los N puntos y guardarlas en las correspondientes casillas del vector err.

Aplicando vuestra función a los datos del ejemplo debéis obtener 0.1263 y 1.1994 como errores en los 2 primeros puntos (las 2 primeras componentes del vector err).

Adjuntar código de vuestra función.
Dad el valor del error para los 3 otros puntos del ejemplo anterior.

LAB 4: Escribir una función con una implementación del algoritmo RANSAC:

```
function [T Nok]=ransac(xy1,xy2)
```

A partir de N parejas de puntos se trata de hallar el mejor ajuste (usando una T proyectiva) que pase de las coordenadas de partida (xy1) a las de llegada (xy2).

Si los puntos no tuvieran errores usaríamos directamente **`T=get_proy(xy1,xy2);`** para encontrar el mejor ajuste. Debido a que nuestros datos pueden incluir falsas parejas es necesario aplicar un algoritmo robusto tipo RANSAC para descartar los "outliers". La función anterior devuelve dos parámetros:

- T = la matriz de transformación (3x3) de ajuste (usando solo los "inliers").
- Nok = el número de puntos correctos ("inliers") aceptados.

El algoritmo es el siguiente:

1) Determinar Np = número de parejas a partir del tamaño de los datos de entrada.

2) Inicializar ok=[]; para guardar los índices de los puntos correctos encontrados.

3) Hacer un bucle de 1000 iteraciones. En cada una de ellas:

- "Elegir" 4 pares de puntos aleatoriamente. Para ello, usad rand() y cread un vector idx de índices con 4 enteros aleatorios entre 1 y Np. Usando idx, extraed las correspondientes 4 pares de coordenadas de xy1 y xy2.
- Llamar a get_proy() para esas 4 parejas y obtener la correspondiente T.
- Usar la función error_ajuste para calcular el error de esa T (calculada para sólo 4 puntos) cuando se aplica a TODOS los puntos (xy1,xy2).
- Hallar cuántos de los puntos tienen un error aceptable (menor de `0.5` píxeles). En MATLAB podemos hacer algo como: `cerca=find(error<0.5)`
- Si la longitud (length) de `cerca` (número de puntos suficientemente cerca del ajuste definido por T) es mayor que la longitud de ok (el mejor caso encontrado hasta ahora), se actualiza ok con los nuevos índices.

Sería conveniente que al crear el vector idx os aseguraseis de que los 4 índices no son iguales. Si elegís por azar 2 puntos iguales tendréis un warning al tratar de usar una matriz singular en el ajuste. Aún así el algoritmo seguramente funcionará porque al hacer muchas pruebas esos malos resultados no se usarán.

4) Al terminar el bucle tendremos en ok los índices de los puntos que caen cerca del ajuste más "popular": esos serán los "inliers". La transformación T que devuelve la función se calcula con get_proy usando SOLO los datos de esos índices ok. La longitud final del vector ok (nº de "inliers") se devuelve en Nok.

Adjuntad código de vuestra función ransac().

Para verificar la función, haced load data_ransac. Son un conjunto de 40 parejas de coordenadas (xy1)+(xy2). Entre ellas hay 20 parejas buenas, relacionadas a través de la siguiente transformación proyectiva:

```
T = [0.7 0.3 100;
     -0.2 0.75 200;
      0.001 -0.02 1];
```

El problema es que las otras 20 parejas son “outliers” que estropean el verdadero ajuste. Hallar el ajuste (con get_proy) usando todos los datos y volcad la matriz T obtenida. Hacer un plot del error cometido para cada uno de los puntos (la salida de error_ajuste). Adjuntad la gráfica obtenida.

Repetir ahora el ajuste pero ahora usando la función ransac(). ¿Cuántos puntos han sido detectados como "inliers" por ransac? Volcad la nueva matriz T de ajuste, que debe ser mucho más parecida a la ideal. Adjuntad el nuevo gráfico de los errores usando la nueva matriz de transformación. Interpretarlo.

LAB 5: (determinar la conectividad entre las imágenes).

Combinando find_matches() y ransac() vamos a escribir una función que compare todas las imágenes, encuentre posibles parejas y decida qué imágenes están conectadas y a través de qué transformaciones: `function [Q P]=find_QP(s)`

La función recibe el array de estructuras s con los "keypoints" de las imágenes y devuelve dos matrices Q y P (ambas de tamaño NF x NF):

- Q es una matriz con la información de la conectividad entre las imágenes. El valor de Q(i,j) es el número de parejas (confirmadas con RANSAC) halladas entre las imágenes i y j. Obviamente Q es simétrica, Q(i,j)=Q(j,i).

- P es un array de celdas donde **P{i,j}=T(i→j)**, la matriz T que convierte las **coordenadas de la i-ésima imagen en las de la j-ésima imagen**. En P{j,i} estará la correspondiente **transformación inversa** que pasa de coordenadas de la imagen j a coordenadas sobre la imagen i.

El proceso es el siguiente:

1. Inicializar las matrices Q y P: Q=zeros(NF) y P=cell(NF).

2. Hacer un doble bucle en i=1:NF y en j=i+1:NF. En cada paso:

   - Usar find_matches(s{i},s{j}) para encontrar las posiciones (xy1),(xy2) de los posibles emparejamientos entre ambas imágenes.
   - Si el número de pares encontrados por find_matches() es mayor de 10, usar los datos de xy1, xy2 para alimentar el algoritmo RANSAC.
   - Si RANSAC confirma al menos 8 parejas, usad sus resultados Nok y T para rellenar `Q(i,j),P{i,j}`. Usando las propiedades de P y Q indicadas antes rellenar `Q(j,i)` y `P{j,i}` sin necesidad de cálculos adicionales.
   - Ir acumulando en una variable el número de parejas propuestas por find_matches() (si superan el umbral pedido de 10) para todos los pares de imágenes y en otra las aceptadas finalmente por RANSAC.

La 1ª fila de la matriz Q me sale: **0 38 0 0 24 96 24 17 0 0 0 0**

La matriz que transforma la 1ª imagen en la 2ª es:

```
 0.9464   -0.1414   56.1619
 0.0104    0.8384  370.5177
-0.0000   -0.0002    1.0000
```

Adjuntar código de find_QP() y volcar la matriz Q de "conectividad" obtenida.
Volcad la matriz de transformación P entre coordenadas de la imagen 1 --> 5 y la que pasa de coordenadas de la imagen 7 a la 1.
¿Qué porcentaje de parejas ha aceptado RANSAC sobre las detectadas iniciamente?

Podéis hacer `save QP_data Q P` para guardar las matrices Q y P y no tener que repetir los pasos anteriores en los siguientes ejercicios (recuperar luego los datos con `load QP_data`).

LAB 6: En este apartado usaremos la información de la conectividad de la matriz Q y las transformaciones guardadas en P para obtener las transformaciones a aplicar a cada imagen para llevarla a un sistema de referencia COMÚN (el de una imagen definida como imagen base o "ancla"). Lo primero es determinar cuál será nuestra imagen ancla. Interesa que esté por el centro del mosaico (para minimizar las deformaciones a aplicar al resto de las imágenes). Un posible criterio es escoger la imagen que esté conectada con el mayor número de las otras imágenes. Escribir un código para calcular (a partir de la matriz de conectividad Q) el índice de la imagen más conectada con otras. Si hay empate podéis usar el número de puntos de conexión para desempatar. Adjuntar el código usado.
¿Qué imagen habéis escogido como ancla? ¿Con cuántas imágenes está conectada?

Escribir la función **`function T=enlaza(Q,P)`**, que reciba las matrices Q, P, y devuelva un array de celdas T{} de tamaño 1xNF con las transformaciones a aplicar a las NF imágenes para llevarlas al sistema de referencia de la imagen ancla. Dentro de la función, lo primero es usar el código anterior para determinar la imagen ancla.

En la matriz P están ya calculadas las transformaciones entre todos los pares de imágenes conectadas entre sí. Se trata de referir TODAS esas transformaciones con respecto de la imagen definida como ancla. Usar el siguiente proceso para obtener la lista de transformaciones T{} buscada:

- Reservar espacio para un array de celdas T de tamaño 1 x NF. Usando una variable lógica de MATLAB, etiquetar todas las imágenes como no procesadas: **`proc = false(1,NF);`**

- La transformación de la imagen ancla en sí misma es obviamente la identidad, por lo que podemos hacer T{ancla}=eye(3) (matriz identidad 3x3). Marcad la imagen ancla como ya procesada haciendo **`proc(ancla)=true`**.

- Mientras queden imágenes sin procesar repetir:
  1. Crear una lista de imágenes ya procesadas con **`usadas=find(proc)`** y otra con las no usadas: **`sin_usar=find(not(proc)).`** Extraer una submatriz de Q con A=Q(sin_usar,usadas), cuyas filas son las de las imágenes no usadas y cuyas columnas correspondan a las ya procesadas.
  2. Buscar la posición del máximo de esa submatriz. La fila de dicho máximo corresponde a la imagen no colocada con más puntos comunes con alguna de las ya colocadas (columna del máximo). Para hallar la posición del máximo: **`[M,pos]=max(A(:)); pos=pos(1); [i,j]=ind2sub(size(A),pos);`**
  3. Las entradas **`i,j`** de las listas **`sin_usar`** y **`usadas`**, respectivamente, nos darán los índices de la nueva imagen (new) a enlazar y la imagen (ref) (de las ya colocadas) con la que está enlazada.
  4. Construir T{new} combinando la transformación entre las coordenadas de la imagen new-->ref (guardada en la correspondiente casilla de la matriz P) con la transformación de ref al ancla (ya calculada y guardada en T{ref}, al ser la imagen ref una de las que ya están procesadas).
  5. Marcar la imagen new como procesada.

Al terminar tendremos en T{} la lista con las transformaciones entre cada una de las imágenes y el ancla. El volcado siguiente muestra el progreso de este proceso de colocación para mi caso y los resultados finales para T{8} y T{7}:

```
Ancla = img # 5     Image  8 enlaza con la  5 (en  66 puntos)  T{8} = 0.775  0.0009 267.623
                    Image 10 enlaza con la  8 (en  60 puntos)        -0.079  0.8603  81.532
                    Image  2 enlaza con la  5 (en  53 puntos)       -0.0002 -0.000  0.9382
                    Image  4 enlaza con la  5 (en  52 puntos)
                    Image  7 enlaza con la  8 (en  52 puntos)  T{7} = 0.722  -0.144  316.91
                    Image 12 enlaza con la  7 (en  89 puntos)        -0.060   0.745  311.36
                    Image  6 enlaza con la  7 (en  68 puntos)       -0.0002 -0.0002  0.9168
                    ...
```

Adjuntar el código de enlaza() y los resultados para las matrices T{1},T{2},T{3}.

LAB 7: Ya solo queda montar el mosaico completo aplicando las transformaciones T{k} a las distintas imágenes. En este apartado usaremos un par de funciones del proyecto anterior: `rango_uv()` y `aplica_T()`.

Lo primero que hay que hacer es determinar cuál será el tamaño final del mosaico para reservar una imagen de ese tamaño (que luego rellenaremos).

La función `[RU,RV]=rango_uv(size(im),P)` nos da el rango RU y RV de coordenadas destino (sobre el mosaico) al aplicar una transformación P a una imagen de un cierto tamaño. Si usamos P=T{k} sabremos el rango de coordenadas destino donde cae la k-ésima imagen. Usando esta función determinaremos el rango máximo del mosaico tanto en la coordenada u (u_min, u_max) como en la v (v_min, v_max):

- Sabemos que la imagen ancla se queda en su sitio, y como es de 600 x 900, podemos inicializar u_min=1, u_max=900 y v_min=1, v_max=600.

- Luego hacemos un bucle (k=1:NF) barriendo las imágenes. En cada paso se llama a rango_uv con el tamaño de cada imagen y la T{k} correspondiente. La imagen deformada cubre el rango de RU(1) a RU(end) en la coordenada u y desde RV(1) a RV(end) para la coordenada v. Actualizar los valores de u_min, u_max, v_min, v_max si los valores obtenidos exceden los márgenes que tenemos hasta ahora (p.e. si RU(1)<u_min o RU(end)>u_max).

Dar los valores finales obtenidos para u_min, u_max, v_min, v_max

Observaréis que el resultado incluye coordenadas negativas (tanto en u como en v). Esto es debido a que la imagen ancla (que normalmente está por el centro del mosaico) mantenía sus coordenadas originales (1:ancho) x (1:alto). Por lo tanto, una imagen que caiga a su izquierda tendrá coordenadas u negativas (y lo mismo pasará con la coordenada v de las imágenes que estén por encima del ancla).

Estas coordenadas negativas nos molestan a la hora de montar el mosaico, ya que en MATLAB la posición de los píxeles de una imagen tienen que ser siempre >=1.

Afortunadamente la solución es sencilla. Basta sumar a todas las coordenadas un desplazamiento dU=(1-u_min), dV=(1-v_min), de tal forma que la coordenada más baja en u y v sea ahora igual a 1. Los valores de u_max y v_max (tras sumarles los desplazamiento dU, dV) corresponden ahora a las dimensiones (ANCHO y ALTO) del mosaico final. Dar los desplazamiento dU, dV a usar así como el ANCHO/ALTO del mosaico.

Para incorporar este desplazamiento final en las transformaciones T{k} calculadas antes debemos crear una matriz de translación correspondiente al desplazamiento por (dU, dV) y aplicarla a todas las transformaciones T{k} previamente calculadas. Tras este paso, la matriz T{7} modificada queda como:

```
 0.6120 -0.2590  871.79
-0.2335  0.5628 1185.41
-0.0002 -0.0002    0.92
```

Adjuntad matriz de transformación T{1} una vez incorporada la translación (dU,dV).

El último paso es reservar una imagen (ALTO x ANCHO x 3) usando zeros() donde se guardará el mosaico final. Luego basta aplicar todas las transformaciones T{k} a las sucesivas imágenes e ir insertando las imágenes deformadas en las coordenadas adecuadas del mosaico final.

Barrer todas las imágenes (k=1:NF). En cada paso:

1. Leer la k-ésima imagen en im y convertirla a double con im2double().

2. Para la transformación T{k}, calcular el rango destino (RU,RV) de la imagen deformada usando **`rango_uv()`**

3. Usar **`aplica_T`** para obtener la imagen deformada en las coordenadas (RU,RV) y colocar dicha imagen en dichas coordenadas en el mosaico final.

Adjuntad mosaico resultado (imshow).

Veréis que aparecen algunas zonas negras en el mosaico final, debido a los NaN que devuelve la función aplica_T() en aquellos puntos que caen fuera de la imagen. Este problema es igual al que teníamos en el proyecto anterior al insertar una película dentro de otra. Resolverlo como se indicó entonces y adjuntad el mosaico final.

Recortar el mosaico eliminando las zonas negras de alrededor. Adjuntad resultado.

Además de ir incluyendo el código pedido en el fichero de respuestas, adjuntar un **directorio con TODO el código necesario (SIN incluir las imágenes) para que funcione vuestro programa. Entregadlo en un .7z o similar junto con esta hoja de respuestas (en formato pdf).**

---------- Hasta aquí, 70% de la nota ----------

Fusión de exposiciones (30%)

Este último apartado está (intencionadamente) menos detallado que los de las otras prácticas. Se trata de daros mayor libertad para mostrar lo que habéis aprendido, saber usar y combinar las rutinas que habéis desarrollado en otros LABs, etc.

Si necesitáis aclaraciones sobre los detalles del algoritmo de fusión podéis consultar las referencia http://www.cs.technion.ac.il/~ronrubin/Projects/fusion/index.html

En clase vimos con detalle una aplicación de imágenes HDR (High Dynamic Range) que planteaba la relación existente entre la radiancia de la escena y la exposiciones capturadas por el sensor para distintos tiempos de exposición. Este enfoque suponía resolver un sistema de cientos o miles de ecuaciones, pero tenía la ventaja de que recuperaba (salvo un factor) la información sobre la radiancia real de la escena. Si medíamos de forma objetiva la radiancia de un punto podíamos convertir nuestros resultados en radiancias físicas (cantidad de "luz" presente en la escena).

Si nuestro objetivo es simplemente mezclar fotos con distinta exposición y combinar en una imagen las partes que mejor se ven de unas u otras, hay métodos más sencillos. En esta práctica vais a implementar uno de esos métodos, denominado "fusión de exposiciones". Es bastante más sencillo de implementar y tiene la ventaja de usar la pirámide laplaciana que ya tenemos de otro laboratorio.

Partimos de Nf=3 imágenes suministradas (gala_1.jpg, gala_2.jpg, gala_3.jpg) con diferentes exposiciones (aquí no es necesario conocer exactamente sus tiempos de exposición). El método es sencillo: se calculan las pirámides laplacianas de todas las imágenes (usando NL=5 o 6 para el número de niveles o la "profundidad") y luego se combinan en una única pirámide (según las reglas descritas a continuación). Colapsando esta pirámide mezcla obtendremos la imagen resultante.

Para combinar las Nf pirámides (de profundidad NL) en una única el proceso es:

- Para combinar el último nivel NL de la pirámide (la versión pequeña de las imágenes) promediaremos los 3 niveles NL de las 3 pirámides que tenemos.

- El resto de los niveles (de 1 a NL-1) corresponden al detalle de las imágenes. La idea es que queremos conservar la información de aquella imagen con el máximo detalle (las zonas muy expuestas o demasiado oscuras tienen menos detalle que las expuestas correctamente). Una imagen con más detalle en una cierta escala tendrá un valor absoluto más alto en el correspondiente nivel de la pirámide laplaciana. Para imágenes en color se puede usar como criterio la suma de los valores absolutos de la tripleta rgb correspondiente.

- Barrer los niveles de 1 a NL-1 y dentro de cada nivel barrer todos los píxeles. Chequear cual de las 3 pirámides tiene un valor absoluto mayor para el píxel considerado (con alguno de los criterios anteriores) y usar los contenidos de ese píxel para rellenar el correspondiente nivel/píxel de la pirámide mezcla.

Una vez obtenida la pirámide mezcla la convertimos de vuelta a una imagen usando la transformación inversa inv_lap(). **Es importante que en la rutina inv_lap() quitéis la conversión final a uint8 de forma que devuelva una imagen de tipo "double".** Adjuntad código usado para la fusión de las pirámides.

La razón de no convertir a byes es porque, al ser la pirámide a invertir una mezcla de varias imágenes, no se garantiza que sus valores estén en el rango [0-255]. Por eso, en vez de aplicar directamente uint8() (que pondría a 0 los valores menores de 0 y a 255 los mayores de 255), usaremos diversas técnicas de las que conocemos para reescalar la imagen obtenida a partir de la pirámide mezcla.

Lo primero es reescalar la imagen resultante entre 0 y 1. Para ello calcular el valor mínimo v0 y máximo v1 de la imagen. Una vez calculados v0 y v1, usadlos para reescalar la imagen final entre 0 y 1, como vimos en el tema de histogramas.

En esta imagen final esperamos encontrar combinado el detalle de las diferentes imágenes, tanto en las zonas de sombra como en los claros. El problema es que en estas imágenes HDR este detalle puede perderse de nuevo durante el proceso de visualización. Una forma de mantener este detalle es hacer una ecualización local del histograma. Una ecualización aumenta el contraste de una imagen, y el hecho de ser local hace que lo resalte tanto en las zonas claras como oscuras. En MATLAB podemos usar la función:

```
I=adapthisteq(I,'ClipLimit',clip);
```

que implementa dicho algoritmo sobre una imagen 2D. El parámetro clip es el % de píxeles saturados (valores 0 o 1) admitidos en la imagen final. Un valor adecuado es del orden del 1% (0.01) o un poco más bajo. Cuanto más alto más "dramático" (y menos realista) será el resultado.

Esta ecualización se aplicará solo a la componente BW de la imagen. Convertir la imagen al espacio HSV (Hue, Saturación, Value) y aplicar la ecualización al canal de luminancia. Ya que estamos en el espacio HSV podemos también elevar un poco la saturación de color (LAB3), elevándola a una potencia p del orden de 0.6 o 0.7.

Adjuntad el código usado para re-escalar la imagen y aplicarle la ecualización adaptativa (y opcionalmente el resalte de la saturación).

Adjuntad imagen final resultante, acompañada por un zoom de la zona de la cresta.

Se apreciarán imágenes dobles. El problema es que estas fotos se tomaron sin un trípode, por lo que entre ellas hay pequeños desplazamientos y/o giros.

Esto nos indica que en cualquier proceso realista de fusión de imágenes hay que considerar un paso previo de registro de imágenes. Se trata de determinar la transformación (afín o proyectiva) que existe entre ellas y corregirla, de forma que las imágenes estén correctamente alineadas antes de empezar a mezclarlas. Pero para arreglar esto podemos usar las técnicas que hemos desarrollado en la primera parte de este proyecto.

Se trata de:

1. Usar SiftWin32 para obtener puntos significativos en las 3 imágenes y leer los datos con fc_info_puntos(). Indicad el nº puntos encontrados en cada imagen.

2. Considerar la primera imagen (la de exposición media) como la referencia y hallar los posibles emparejamientos entre puntos de las imágenes 2 y 3 con los de la imagen 1. A partir de ellos determinar las transformaciones T2 y T3 proyectivas que pasan de coordenadas de las imágenes 2 y 3 a la 1.
   Indicad el nº de emparejamientos encontrados y volcad las matrices T2 y T3.

3. Aplicar la función aplica_T() del LAB 5 sobre las imágenes 2 y 3 usando las transformaciones T2 y T3 para que las imágenes resultantes estén alineadas con la primera. Como las modificaciones serán pequeñas, usad la función sin argumentos adicionales, de forma que se calculen sobre el soporte original.

4. Al aplicar la transformación a las imágenes se moverán un poco y aparecerán algunas zonas negras en los bordes. Para quitarlas, eliminaremos una franja de píxeles alrededor de las tres imágenes. Es importante que el tamaño final de las imágenes recortadas (en ambas dimensiones) sea múltiplo de 32 o 64. Esto nos permitirá aplicar sobre ellas una pirámide laplaciana con NL=5 o 6 niveles respectivamente (en nuestra implementación de la pirámide laplaciana tenemos problemas al intentar reducir un nivel si su tamaño es impar).

Una vez que tenemos las 3 imágenes alineadas y recortadas, repetir la aplicación del algoritmo de fusión de antes: hallar las pirámides, fusionarlas, volver atrás, normalizar entre 0 y 1 la imagen resultante, aplicar la ecualización adaptativa del histograma al canal de luminancia, reforzar la saturación de color, ...

Adjuntad la nueva imagen resultante. Haced un zoom en la parte de la cresta y adjuntarlo. Debe de haber desaparecido el problema que había de la falta de alineamiento de las imágenes.

Cuando se normaliza la imagen entre 0 y 1, en vez de usar los valores máximos y mínimos de TODA la imagen, se puede trabajar en cada canal por separado de forma que al final cada canal R,G,B esté normalizado entre 0 y 1. Así se suelen obtener resultados más "dramáticos", aunque seguramente con colores "falseados", sobre todo si luego se fuerza la saturación de color. También podéis probad con varios valores de % de saturación (clip) al hacer la ecualización del histograma.

Usar la normalización independiente por canales y experimentar con los parámetros. Adjuntad alguno de vuestros resultados que os guste, indicando los parámetros usados.

Adjuntad dentro de vuestro .7z o similar otro directorio con vuestro código fusion.m junto con el resto de las funciones necesarias para que funcione.