

Proyecto (entrega definitiva en 10 días)



1. MANEJO de VIDEOS en MATLAB

En este apartado vamos a aprender como leer y acceder a la información de un video en MATLAB para escribir una sencilla aplicación de tracking. Para leer el video usaremos el comando `VideoReader()` que recibe el nombre del fichero de video (avi, mp4, etc.) y crea un objeto de tipo video del que luego extraeremos la información que necesitaremos:

```
obj=VideoReader('peli.mp4'); fps=get(obj,'FrameRate'), NF=get(obj,'NumFrames')
```

Un video es una sucesión de imágenes (o frames). Este video tiene $NF=614$ frames, lo que a un ritmo de 25 frames/segundo (fps) nos da una duración total $T \sim 25$ seg. Podemos acceder al k-ésimo frame con `frame=read(obj,k)`; (aunque según vuestra versión de Matlab la función puede ser `readframe`).

Cada frame es una image que se puede visualizar usando `imshow()`. Por ejemplo, con el bucle: `for k=1:150, frame=read(obj,k); imshow(frame); drawnow; end` se leen los primeros 150 frames del video y se van visualizando uno tras otro, lo que nos muestra unos 6 segundos del video en una ventana de MATLAB. Se trata de una hoja en blanco en la que hay 4 puntos (cruces) negras, que es movida durante la película. Al mover la hoja las cuatro marcas cambian de posición dentro de cada frame. Nuestro objetivo será identificar los 4 puntos y mantenerlos localizados a lo largo del video, actualizando su posición (en píxeles) dentro de cada frame.

Para que sea más sencillo, la identificación inicial la haréis manualmente. Cargar el primer frame del video y visualizarlo con `imshow()`. Con el menú de la figura () usad el zoom para ampliar la imagen 3 o 4 veces y ver las marcas más grandes. Luego podéis usar el cursor de datos () para pinchar en el centro de las 4 marcas (**empezando por la esquina superior izquierda y siguiendo en el sentido horario**). Usar la información que nos da el cursor de datos (posición X,Y) para rellenar la siguiente tabla. Como indicación os dejo la posición que he obtenido para el punto.

Pos	1er	2º	3er	4º
X	540			
Y	370			

Usaremos el script suministrado `video.m` como el esqueleto del código. Abrirlo en el editor y **completar las líneas iniciales donde se asignan los valores de la tabla a los vectores X,Y** (las coordenadas X e Y de los 4 puntos).

El script carga el primer frame y superpone 4 puntos amarillos en las posiciones indicadas por X e Y. Si los habéis identificado correctamente deben caer sobre las marcas del papel. Si pulsáis cualquier tecla, un bucle como el que hemos visto antes

muestra el resto del video con un contador de frames. Obviamente, los círculos que se han pintado enseguida dejan de caer sobre los puntos cuando éstos se mueven.

Inciso sobre el código: Para que la visualización sea más eficaz, al llamar a `imshow()` y a `plot()` por primera guardo los objetos gráficos creados (`im_obj, pp_obj, ...`). Así, en cada paso del bucle en vez de hacer nuevas llamadas a `imshow` y `plot`, solo hay que actualizar la información de los objetos gráficos: la propiedad '`Cdata`' para la imagen y las propiedades '`Xdata`', '`Ydata`' para los `plot`.

Para el tracking usaremos el hecho de que las marcas son negras sobre el fondo blanco para seguirles la pista. Además, como entre cada dos frames pasa muy poco tiempo ($\sim 1/30$ seg.), las marcas no se pueden haber movido mucho desde la última posición conocida. Por lo tanto, la búsqueda de la nueva posición puede limitarse a una zona pequeña ($\pm R$ píxeles) alrededor de la antigua posición (en el frame previo). Este parámetro se define como un paso previo (fuera del bucle donde barremos cada frame) junto con un par de matrices `dx`, `dy` con las coordenadas (`x,y`) de esta zona referidas a su centro:

`r=(-R:R); dx=ones(length(r),1)*r; dy=dx';`

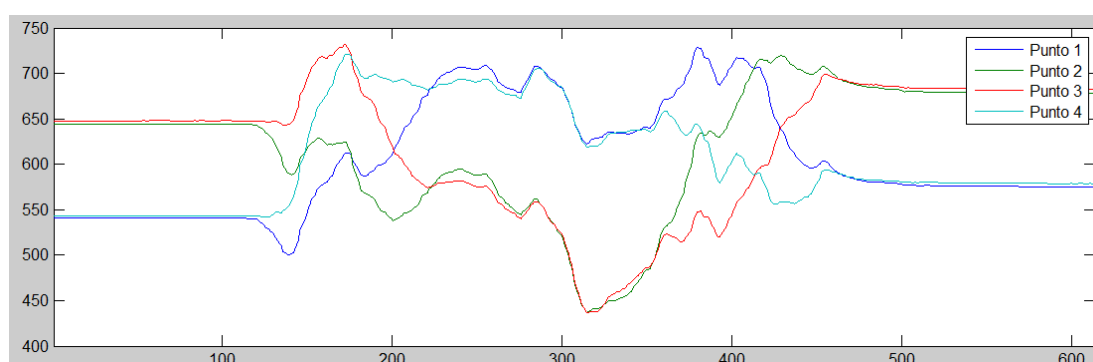
Para cada frame hay que repetir el proceso de búsqueda para los cuatro puntos por lo que estará dentro del bucle (`j=1` a `4`) que barre los 4 puntos (indicado por el comentario **% Actualizar posiciones X,Y**). El algoritmo paso a paso sería:

1. Antes de procesar los 4 puntos (bucle en `j=1:4`) convertir el `k`-ésimo frame a una imagen en niveles de gris usando `rgb2gray()`, ya que nuestro algoritmo solo necesita una versión en "blanco y negro" del frame para trabajar. El resto de las modificaciones tienen lugar dentro del bucle (`j=1:4`).
2. Para la `j`-ésima marca la zona a explorar está centrada en las coordenadas `x=round(X(j))` e `y=round(Y(j))`. Del correspondiente frame quedarnos con una subimagen `S` con el rango de coordenadas `(x+r)` e `(y+r)`, donde `r=(-R:R)`. Es importante recordar que las coordenadas `Y` (verticales) corresponde a las filas y las `X` (horizontales) a las columnas. Convertir esta subimagen `S` a tipo `double` con la función `im2double()`.
3. Calcular el valor mínimo de la subimagen `S` con `S0=min(S(:))` y calcular sus diferencias con respecto a ese valor mínimo: `d = abs(S-S0)`. A partir de `d` calculad `w=exp(-50d)` para obtener una matriz de pesos cuyos valores sean más altos cuanto más cerca estemos del mínimo (`d~0`). Para que `w` sea una buena matriz de pesos dividirla por la suma de todos sus valores. La suma de los valores de una matriz puede calcularse con `sum(A(:))`. Comprobad que la suma de los valores de la matriz `w` es 1 tras hacer esta normalización.
4. Multiplicar (punto a punto) la matriz de pesos obtenida (`w`) con la matriz `x+dx` con las coordenadas de los píxeles de la subimagen y sumar todos los valores de la matriz resultante (como se indica en el apartado anterior). El resultado es la estimación de la coordenada `x` del centro de la marca en este frame y la usaremos para actualizar `X(j)`. Repetir con la coordenada `y` y actualizar `Y(j)`.

Al terminar el bucle ($j=1:4$) en los vectores X, Y estarán las posiciones actualizadas de los 4 puntos. Si todo va bien el video debe mostrar ahora los círculos amarillos siguiendo la posición de las marcas a lo largo del tiempo. [Adjuntad el código dentro del bucle en j con la actualización de las posiciones X\(j\),Y\(j\)](#)

El seguimiento de los puntos debería ahora funcionar. [Adjuntad una captura con el frame 400 de la película.](#)

En el código solo se guardan (en los vectores X,Y) las coordenadas de los 4 puntos para el último frame. Inicializar al principio del script dos matrices U y V (de tamaño 4 x NF) y usadlas para guardar (en cada columna) las coordenadas X,Y de los 4 puntos a lo largo de todo el video. Al terminar podremos ver la evolución de las coordenadas en función del tiempo. La figura adjunta (obtenida haciendo `plot(U')`) muestra mis resultados para las coordenadas x.



[Adjuntad la misma gráfica pero ahora mostrando la evolución de las coordenadas Y de los 4 puntos según vuestros resultados.](#)

Si (x_1, y_1) son las coordenadas del 1er punto e (x_2, y_2) las del segundo en un frame, calculad la distancia entre ambos píxeles usando:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Haced un plot de dicha distancia a lo largo de todos los frames de la película. [Adjuntad la gráfica obtenida.](#) Obviamente la distancia entre ambos puntos en el papel no cambia. Los cambios observados son un efecto de la perspectiva debido al cambio de posición y orientación de la hoja respecto a la cámara (por ejemplo, la distancia sobre el frame aumenta cuando el dibujo se acerca a la cámara). Veremos en el siguiente tema cómo estas diferencias pueden usarse para deducir la posición/orientación de la hoja con respecto de la cámara.

Extraer las coordenadas X del 1er punto y mostrad en un plot la diferencia dX entre la coordenada X en un frame y en siguiente. [Adjuntad el plot.](#) Esta gráfica nos indica el máximo movimiento detectado entre frames. Si el radio de búsqueda R se hace menor que los desplazamientos observados es muy posible que el tracking se pierda. [Volver a correr el programa con R=10 y volver a adjuntar una captura de la película \(con los puntos superpuestos\) para el frame #400.](#) ¿Qué sucede en este caso?

2. CREACIÓN de MOSAICOS

El objetivo es crear imágenes formadas por un mosaico de pequeñas imágenes. Se parte de una imagen objetivo (target) típicamente con una alta resolución junto con una colección de imágenes pequeñas. Se trata de construir una versión de la imagen "target" como un collage/mosaico superponiendo las imágenes pequeñas.



Una imagen como la de la izquierda (del videojuego "Life is Strange") sería la imagen target. A la derecha se muestra el mosaico (4032 x 7488 píxeles), creado por los productores del juego a partir de imágenes enviadas por los usuarios. En la parte derecha se muestran un par de niveles de zoom en la zona de una de las caras para que podáis apreciar el detalle subyacente. El mosaico completo lo podéis encontrar en <http://twinfinite.net/2016/01/life-is-strange-mosaic/>.

En este proyecto exploraremos varias estrategias para crear este tipo de imágenes. Para que sea más sencillo trabajaremos en B/W y con imágenes del mismo tamaño, pero no sería muy complicado expandirlo a casos más generales.

La colección de imágenes usadas para componer el mosaico (retratos B/W de 288 píxeles de alto y 192 de ancho) están en el directorio ./retratos. El script lee_jpgs suministrado lee todas las imágenes de este directorio y las guarda en un array de tamaño 288 (alto) x 192 (ancho) x 360 (número total de imágenes L). Tras ejecutar el script podemos acceder a los contenidos de la k-ésima imagen con `imgs(:, :, k)`. Las imágenes están ya convertidas a tipo double con valores entre 0 y 1.

2.1 Primer mosaico

Primero montaremos todas las imágenes en un mosaico para visualizarlas sin tratar de que se parezcan a la imagen target. Reservad espacio para una imagen de 3456 x 5760 con `mosaico=zeros(3456,5760);` Como cada foto pequeña es de 288x192 en esta imagen entrarán $(3456/288) \times (5760/192) = 12 \times 30 = 360$ imágenes (que son todas las imágenes de la colección). La idea es ir rellenando el mosaico por filas (recorriéndolo primero de izquierda a derecha y luego de arriba a abajo) e ir colocando en él las sucesivas imágenes de la colección.

Lo más sencillo es hacer un doble bucle barriendo primero las filas ($j=1:12$) y dentro de él, un 2º bucle barriendo las columnas ($i=1:30$). Inicializad un par de

variables $rx=(1:192)$ y $ry=(1:288)$ con el rango de posiciones del mosaico donde colocar las fotos. Cada vez que añadamos una imagen se incrementa rx por 192 para colocar la siguiente a su derecha. Al final de una fila incrementar ry por 288 y resetear rx a $(1:128)$ para procesar la siguiente fila. Dentro de ambos bucles se trata de rellenar los contenidos de `mosaico(ry,rx)` con la n -ésima imagen de la colección (incrementad n cada vez que uséis una imagen). [Adjuntad código e imagen resultante](#). La imagen adjunta corresponde a la esquina superior izquierda de la imagen que debéis obtener.



De cara al resto de la práctica es conveniente tener definidos los tamaños de la imagen destino y de las imágenes de la colección y usar dichos valores al hacer los bucles. De esta forma será más sencillo modificar el programa si decidimos usar un tamaño diferente para la imagen target o para las fotos de la colección.

Ahora hay que elegir una imagen como "target". Al montar un mosaico se pierde resolución, por lo que interesa una imagen que no tenga un detalle muy fino, figuras muy pequeñas, etc. He elegido el retrato adjunto (cargadla de `target.jpg`), también en blanco y negro. Antes de hacer nada convertirla a una imagen con valores de tipo `double` con `im2double()`.



Esta imagen target tiene el mismo tamaño (3456 x 5760) que la creada antes, con 12 filas (3456/288) de 30 imágenes (5760/192) cada una, por lo que podremos usar como punto de partida el mosaico creado antes.

Obviamente el mosaico anterior no se parece para nada a la imagen "target" ya que sus fotos fueron colocadas secuencialmente una tras otra. Ahora ejecutaremos un programa que será muy similar al del apartado anterior, con la diferencia de que ahora se trata de escoger la foto de la colección más parecida al correspondiente trozo de la imagen target. Con los mismo bucles de antes barremos las 12x30 posiciones del target, pero ahora, en cada paso:

1. Extraemos la sub-imagen 288x192 de la imagen target correspondiente a las coordenadas (rx,ry) . Esta será la imagen a la que deseamos parecernos con una de las imágenes de la colección.
2. Para esa posición tenemos ahora un candidato previo, los contenidos ya existentes en `mosaico(ry,rx)` (alguna de las imágenes de la colección). Vamos a cuantificar una medida de la diferencia entre las dos imágenes. Para ello calcularemos la resta entre ambas imágenes, luego se calcula el valor absoluto (`abs`) de dicha resta y finalmente hallamos su media con `mean2()`. Este error `e0` será el error de partida y solo se aceptará una foto si su error con respecto a la imagen target es menor que éste.

3. Haremos un bucle (de $n=1$ a L) comparando la sub-imagen deseada del target con las $L=360$ imágenes de la colección. Para que el proceso sea un poco más rápido no esperaremos a encontrar la foto de la colección con el menor error. Si el error de una cierta foto cumple que $e/e_0 < U$ (umbral) la foto es aceptada y se abandona la búsqueda (aunque fuese posible encontrar un mejor ajuste con alguna foto todavía no comparada). Si por ejemplo $U=0.4$, una imagen es aceptada inmediatamente si su error es menor del 40% del error de partida. Si tras barrer todas las fotos ninguna verifica que $e/e_0 < U$ se usará aquella con el menor error (obviamente, siempre que mejore el error inicial e_0).
4. Tras terminar la búsqueda colocamos la imagen elegida (por superar el umbral o por ser la mejor de todas) en el rango de posiciones (r_y, r_x) del mosaico (de donde se extrajo la sub-imagen target).

Adjuntad script con el código usado. Ejecutadlo (con un umbral $U=0.4$) y adjuntad (imshow) la imagen mosaico resultante.

2.2 Penalización para igualar el uso de las diferentes imágenes

Esta técnica sencilla (cuyo resultado debe ser similar al de la imagen adjunta), además de su baja resolución, tiene algunos problemas. En particular se observa que ciertas imágenes aparecen muchas veces sobre todo en zonas homogéneas (cara o fondo). Esta es una consecuencia de que el algoritmo favorece a las primeras imágenes de la colección, que son siempre comparadas antes.



Para cuantificar el problema, inicializad en el script un vector `usos=zeros(1,360)`; que usaremos para llevar la cuenta de las veces que cada imagen de la colección es usada en el mosaico. Cada vez que la n -ésima imagen de la colección sea escogida, incrementad el contador `usos(n)` en 1. **Al terminar, haced un plot del vector usos y adjuntadlo. ¿Cuántas imágenes son usadas más de 20 veces? ¿Qué tanto por ciento del total de imágenes usadas (360) representan dichas imágenes?**

Para reducir este efecto y conseguir un mosaico más variado vamos a añadir una penalización para que cada vez que una imagen es escogida ya no se la vuelva a considerar durante un cierto número de comparaciones.

Lo implementaremos con un vector `timeout (1 x 360)` inicializado a ceros y una variable `PENALTY=50` con la penalización escogida:

- En el bucle ($n=1$ a 360) donde se hace la comparación con todas las imágenes verificar (antes de hacer nada) si `timeout(n)>0`. Si ese es el caso es que la imagen está todavía "castigada" y no se la considera (usar `continue` para saltar al siguiente paso y no tener que hacer más cálculos).

- Tras la comparación si p.e la imagen 3 es finalmente escogida (bien porque $e/e_0 < U$ o porque su error es menor que la de cualquier otra imagen) haremos $\text{timeout}(3) = \text{PENALTY}$ para que en las siguientes pruebas no se use.
- Antes del bucle ($n=1:360$) que barre todas las imágenes, restad 1 al vector timeout para que tras cumplir su “castigo” el valor de timeout para esa imagen ya no sea positivo y pueda ser considerada de nuevo.

Adjuntad modificaciones del código, el mosaico obtenido y la gráfica final del vector usos con el nº de veces que cada imagen es usada. ¿Cuál es ahora el número máximo de veces que una imagen es usada?

2.3 Combinación de diferentes escalas

El mosaico resultante evita el carácter repetitivo del anterior, al usar de forma más equitativa las imágenes de la colección, pero aún presenta el problema de su baja resolución, al ser las fotos usadas relativamente grandes comparadas con la imagen target. Una posibilidad sería reducir el tamaño de las fotos a la mitad (144×96) y repetir el proceso (ahora se necesitarían 24×60 fotos para cubrir todo el mosaico).

Esta técnica puede combinarse con el resultado anterior, para obtener un mosaico que será una combinación de imágenes con dos tamaños.

- Reducir las fotos de la colección a la mitad usando `imags=imresize(imags,1/2);`
- El punto de partida es ahora el mosaico obtenido en el apartado anterior y es el que usaremos para determinar el error de partida que una de las nuevas fotos reducidas debe superar para ser incorporada.
- La penalización usada (PENALTY) se multiplicará por 2 para reflejar el hecho de que ahora tenemos más trozos del mosaico a rellenar. Si antes dejábamos pasar 50 pruebas antes de considerar una imagen de nuevo, ahora serían 100.
- El UMBRAL aumenta a $U = \sqrt{U}$. Por ejemplo si antes se usó $U=0.4$ ahora sería $U=0.63$. Se trata de reflejar el hecho de que cada vez será más difícil mejorar los ajustes ya encontrados y seremos menos exigentes aceptando un parecido.
- El resto del código es esencialmente el mismo que antes (teniendo en cuenta las nuevas dimensiones). Si vuestro código estaba correctamente escrito (sin asumir un tamaño fijo para las imágenes usadas) debería funcionar sin problemas para la nueva colección de imágenes reducidas. [Adjuntad el mosaico resultante.](#)

En esta segunda pasada, de los $24 \times 60 = 1440$ trozos que probamos, ¿qué porcentaje es sustituido por una de las nuevas fotos reducidas? De ellas, ¿cuántas se aceptan con el criterio de que el error sea menor que $U \cdot e_0$?

Este proceso puede repetirse con tamaños sucesivamente más pequeños de las fotos (72x48, ...), usando siempre como punto de partida el mosaico anterior. En cada paso el algoritmo trata de colocar versiones cada vez más pequeñas de las fotos en el mosaico (siempre que se mejore el parecido a la imagen target respecto al contenido existente hasta ahora). Volver a ejecutar el código otras 2 veces, reduciendo cada vez las fotos a la mitad (tamaños de 72x48 y 36x26). Recordad en cada escala hacer $PENALTY = 2 * PENALTY$ y $U = \sqrt{U}$ para actualizar la penalización y el nivel umbral usados.

Adjuntad mosaico final combinando las fotos con las 4 escalas distintas. Extraer la zona del mosaico correspondiente al rango de filas de 700 a 1500 y columnas 2200 a 3500. Adjuntad una captura de esa zona. En la última pasada (con las imágenes más pequeñas), ¿qué porcentaje de "trozos" del mosaico son substituidas?

2.4 Modificando las imágenes de la biblioteca

En el algoritmo anterior no se nos permitía modificar las imágenes de la "biblioteca" de los retratos. Simplemente se buscaba una parecida al trozo de la imagen target que queríamos cubrir y se la colocaba en su lugar.

En este apartado vamos a permitir cambiar el nivel de la imagen antes de usarla. La idea es que, en vez de `imgs(:, :, k)` podremos usar `(imgs(:, :, k) + a)` para rellenar el mosaico, donde a es un valor que nos permite oscurecer ($a < 0$) o abrillantar ($a > 0$) la imagen de la librería antes de ser usada.

Para ello, antes de comparar entre el trozo de mosaico que se trata de reemplazar y una imagen de la biblioteca `im`, calculemos el valor de a como $a = \text{mean2}(\text{trozo} - im)$.

Luego, al estimar el error lo calculemos comparando trozo con `(im + a)`. Al final, si la imagen resulta elegida colocaremos `(im + a)` en la correspondiente zona del mosaico en vez de simplemente `im` como hacíamos antes.

Adjuntad código y el mosaico obtenido tras correr el nuevo algoritmo (solo para la primera escala, usando las imágenes de los retratos en su tamaño original).

Al igual que antes, este algoritmo puede volverse a ejecutar reduciendo como antes el tamaño de las fotos de la colección. Ejecutad código para tres escalas adicionales (bajando como antes hasta un tamaño de fotos de 36x24). Adjuntad mosaico final. Adjuntad un detalle del mosaico correspondiente al rango de filas (700:1500) y de columnas (2200:3500).