

Proyecto (para entregar en grupos de dos)

En este primer proyecto se trata de familiarizarnos con aspectos básicos del manejo de imágenes: lectura de imágenes, extraer partes de ellas, acceso a los planos de color, operaciones básicas, etc.

Introducción

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

Una imagen como la adjunta sería la imagen target (del videojuego "Life is Strange"). Abajo tenéis un mosaico (de tamaño de 4032 x 7488 píxeles), creado por los productores del juego a partir de más de 1000 imágenes sobre el juego creadas por los "fans". A la derecha se muestran un par de niveles de zoom en la zona de una de las caras para apreciar el mosaico subyacente. El mosaico completo lo podéis encontrar en <http://twinfinite.net/2016/01/life-is-strange-mosaic/>.



En este proyecto explicaremos varias formas para crear estas imágenes. Para que sea más sencillo nos limitaremos al caso blanco y negro, con imágenes del mismo tamaño, etc., pero no sería muy complicado expandirlo a casos más generales.

En primer lugar necesitamos una colección de imágenes para componer el mosaico. Usaremos las imágenes en el directorio `./retratos/` que corresponden a pequeños retratos en blanco y negro de 288 píxeles de alto por 192 de ancho.

El script `lee_jpgs` suministrado lee todas las imágenes del directorio y las guarda en un array de tamaño 288 (alto) x 192 (ancho) x 520 (número total de imágenes N). Una vez ejecutado el script y leídas las N=520 imágenes podemos acceder a los contenidos de la k-ésima imagen usando `imags(:, :, k)` en MATLAB.

En primer lugar haced `mosaico=uint8(zeros(2880,4800));` que reserva espacio para una imagen de 2880 x 4800 píxeles, que usaremos para mostrar los primeros 250 retratos, organizados en 10 filas de 25 imágenes cada una. Así tendremos un alto de $10 \times 288 = 2880$ píxeles y un ancho de $25 \times 192 = 4800$ píxeles. La imagen debe ser como la adjunta, con más filas y columnas.



Lo más sencillo es hacer un doble bucle barriendo filas ($k=1:10$) y dentro de cada fila las columnas ($j=1:25$). Inicializar el rango $ry=(1:288)$ e incrementarlo por 288 en cada fila. Previo al barrido de cada fila hacer $rx=(1:192)$ e incrementarlo por 192 en cada columna. En cada paso, se trata de rellenar el contenido de `mosaico(ry,rx)` con la n-ésima imagen de la colección, `imags(:, :, n)`. [Adjuntad código y e imagen resultante.](#)

A continuación borrar el contenido de `mosaico` poniendo a ceros sus valores (SIN USAR BUCLES). Vamos a rellenar de nuevo `mosaico` con las imágenes, pero ahora poniéndolas en posiciones aleatorias. Para ello, hacer un bucle barriendo todas las imágenes ($n=1:520$) y en cada paso escoger aleatoriamente una posición $x0$ e $y0$, para colocar la n-ésima imagen en el rango $ry=y0+(1:288)$, $rx=x0+(1:192)$ de la imagen compuesta.

La única dificultad es asegurarnos de no salirnos del tamaño de la imagen destino (2880 de alto y 4800 de ancho). Para los aleatorios usaremos `rand` que nos da un número entre 0 y 1. Si lo multiplicamos por $(4800-192)$ y hacemos un `floor()` del resultado, tendremos un número entero entre 0 y 4608. Este será nuestro $x0$, lo que asegura que el rango destino $x0+(1:192)$ no se sale de la imagen compuesta.

Repetir para obtener aleatoriamente el valor de $y0$. Es importante hallar primero los tamaños de la imagen destino y de las imágenes de la colección, usando dichos valores en el programa en vez de escribir directamente $(4800-192)$ ó $(1:192)$. De esta forma el programa se podrá usar con otros tamaños de imágenes.

[Adjuntad código y e imagen resultante.](#)

Creación de mosaicos

Una vez que disponemos de las N=520 imágenes de la colección, el siguiente punto es escoger una imagen "target". Como en el mosaico se pierde resolución conviene que sea una imagen que no tenga un detalle muy fino, figuras muy pequeñas, etc. He elegido el retrato adjunto (cargadla de `target.jpg`), también en B/W:



La imagen target tiene un tamaño de 4608 x 7680. Como las imágenes usadas son de 288x192, el mosaico tendrá 16 (4608/288) filas de 40 (7680/192) imágenes cada una, para un total de 640 imágenes. Usando el comando de antes volver a reservar una matriz mosaico (ahora de 4608 x 7680) para guardar el resultado. El programa será muy similar al usado en el apartado anterior (primer caso), con la diferencia de que antes se usaban las imágenes de la colección por orden, mientras que ahora se trata de escoger la imagen más parecida al trozo de la imagen target.

Para hacer más rápido el proceso de búsqueda nos interesa tener una versión reducida (en un factor $F=4$) de cada imagen de la colección. Esto se puede hacer usando una sola orden: `reds=imresize(imags,1/F)`, que crea un nuevo array `reds`, similar a `imags`, pero en una versión de tamaño reducido ($288/F \times 192/F \times 520$).

El proceso básico es sencillo. Como antes, barreremos las 16 x 40 posiciones del target, escogiendo la imagen más parecida de la colección, que usaremos para ir rellenando el mosaico:

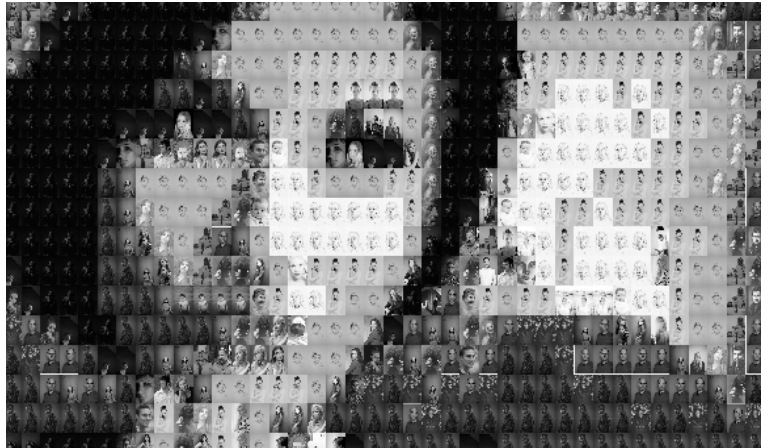
1. Extraer la sub-imagen 288x192 de la imagen target ampliada. Lo más sencillo es usar como antes una variable `ry=(1:288)`, que se incrementará en saltos de 288. En cada fila una segunda variable `rx=(1:192)` se incrementa por 192 en cada paso. La sub-imagen a comparar sería `target(ry,rx)`. Para agilizar el proceso de comparación, esta subimagen se reduce usando `imresize` con un factor $1/F=0.25$.
2. Una vez extraída y reducida la subimagen objetivo, hacemos un bucle y la comparamos con las $N=520$ imágenes de la colección (en sus versiones reducidas, `reds`, que hemos calculado previamente). Para determinar la más parecida calculamos la diferencia entre ambas imágenes (recordad pasar a `double` antes de hacer la resta), luego calculamos el valor absoluto (`abs`) de dicha diferencia (`abs`) y finalmente hallamos su media con `mean2()`. Ese será el error `e` para esa imagen de la colección, que se comparará con el menor error encontrado hasta entonces. Si es mejor se actualiza dicho error mínimo y se guarda el índice de la imagen para el que se ha hallado dicho mínimo.

3. Al terminar el bucle colocamos la imagen más parecida encontrada (en su versión original de 288x192) en el mismo rango de posiciones de la imagen destino (mosaico) de la que se extrajo de la imagen target (ry,rx).

Adjuntad script con código. Usando imshow, adjuntad la imagen mosaico resultante.

----- Hasta aquí 25% de la nota -----

Esta técnica sencilla (cuyo resultado debe ser similar a la imagen adjunta) no da resultados demasiado buenos. Además de la baja resolución, un problema del algoritmo anterior es que en zonas homogéneas (cara o fondo) es muy posible que la misma foto sea siempre la escogida. Esto da lugar a zonas muy repetitivas en el mosaico. Como podemos observar, algunas de las imágenes se terminan usando mucho más que otras.



Para cuantificar el problema, inicializad en el script un array veces (1x520) lleno de ceros donde llevaremos la cuenta de las veces que cada imagen de la colección se usa en el mosaico. Cada vez que la n-ésima imagen de la colección sea escogida, incrementad el contador veces(n) en 1.

Tras terminar, haced un plot del vector veces y adjuntadlo.

Adjuntad las 2 imágenes más usadas. ¿Cuántas veces se usan cada una de ellas?

Para disminuir este efecto hay varias posibilidades. Una muy sencilla es usar el contador anterior para añadir una penalización en función del número de veces que se haya usado ya una imagen. Por ejemplo, tras calcular el error de la n-ésima imagen lo multiplicaríamos por el siguiente factor:

$$f = \left(1 + \frac{4 \cdot \text{veces}(n)}{(\max(\text{veces}) + 1)} \right)$$

De esta forma, una imagen que no haya sido usada previamente (veces(n)=0) nos da un factor=1, no sufriendo ninguna penalización. La imagen más popular hasta ahora (veces(n)=max(veces)), se penalizará con un factor $f \sim 1 + 4 \sim 5$, lo que baja mucho sus probabilidades de ser escogida.

Adjuntad nuevo mosaico y la gráfica de veces.

¿Qué imagen (#?) es ahora la más usada? ¿Cuántas veces?

Adjuntad código de vuestro script.

----- Hasta aquí 40% de la nota -----

El mosaico resultante evita el carácter repetitivo del anterior, al usar de forma más equitativa las imágenes de la colección, pero aún presenta algunos problemas. Veamos cómo mejorarlo. En primer lugar, podríamos tener más flexibilidad si no estuviésemos limitados a colocar todas las imágenes del mosaico una al lado de la otra, sino que pudiéramos ponerlas en cualquier posición (por supuesto, siempre dentro de la imagen "contenedora"). Esto es muy similar a lo que hicimos en el 2º caso del primer apartado donde elegíamos una posición aleatoria x_0, y_0 para colocar nuestras imágenes.

La idea es combinar ambos códigos: elegir una posición aleatoria (x_0, y_0) pero en vez de poner una imagen cualquiera como hicimos entonces, elegiremos la imagen de la colección que mejor se ajusta a la imagen target en dicha posición. Además, ahora no tenemos que partir de cero, podemos usar el mosaico obtenido antes como punto de partida. En esta situación sólo usaremos una imagen de la colección si podemos superar a lo que ya tenemos. El algoritmo es el siguiente:

1. El punto de partida es la imagen objetivo (target) y el resultado (mosaico) anterior. También inicializamos el vector veces a ceros como antes.
2. Inicializar un contador de aciertos a 0 y hacer un bucle de p.e. 200 iteraciones. En cada paso:
 - Escoger una posición aleatoria (x_0, y_0) como hicimos antes, asegurándonos de no salirnos del tamaño de las imágenes (4608 x 7680). El rango objeto de estudio será $ry=y_0+(1:288)$, $rx=x_0+(1:196)$.
 - El objetivo es parecernos a la subimagen $target(ry, rx)$. Comparar dicha imagen con lo que ya tenemos en $mosaico(ry, rx)$ y calcular su error (como lo hacíamos antes). Este será el error mínimo E_{min} a reducir.
 - Hacer un bucle ($n=1:520$) barriendo las imágenes de la colección. Calcular su error (incluyendo ahora la penalización por el número de veces que ha sido usada la imagen n). Guardarlo si es mejor (más pequeño) que el error mínimo que ahora tenemos.
 - Tras terminar el bucle, **si hemos superado el error de partida**, substituir los contenidos de $mosaico(ry, rx)$ por los de la imagen ganadora, incrementar el contador de aciertos y la casilla correspondiente del vector veces para esa imagen.
3. Si tras las 200 iteraciones los casos aceptados son más de un 5% (10) volver al paso 2 y hacer otra tirada de 200 pruebas aleatorias. Por el contrario, si el porcentaje de éxito baja de ese 5% es que ya es difícil encontrar sustituciones que reduzcan el error, por lo que terminamos el proceso.

Adjuntad código e imagen resultado. ¿Cuántas tiradas de 200 habéis hecho antes de que tanto por ciento de substituciones aceptados baje del 5%? ¿Qué tanto por ciento de aceptación tuviste en la última tirada de 200 pruebas?

Observaréis que al permitir colocar imágenes en cualquier posición rompemos la regularidad asociada a la malla rectangular de los mosaicos anteriores.

A pesar de estas mejoras, el principal problema sigue siendo la baja resolución obtenida, debido al tamaño relativamente grande de las imágenes de la colección.

Esto se puede solucionar reduciendo las imágenes de la colección por un factor 2 (siendo ahora de 144x96) y volviendo simplemente a correr el código anterior (no os olvidéis de recalcular también las imágenes reducidas `reds` tras cambiar el tamaño de las imágenes de la colección). Recordad como esto podía hacerse con una sola orden usando `imresize()`.

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. Notad que al usar imágenes más pequeñas es más fácil hallar buenos ajustes y se necesitan más iteraciones para bajar del 5% de aceptados.

Adjuntad el mosaico final resultante (para 144x96) ¿Cuántas tiradas de 200 hay que hacer ahora para que el tanto por ciento de substituciones aceptados baje del 5%? Adjuntad código de vuestro script para la parte aleatoria.

La ventaja de este enfoque es que puede repetirse con tamaños sucesivamente más pequeños (144x96, 72x48, ...), usando siempre como punto inicial el mosaico anterior. En cada paso el algoritmo coloca versiones cada vez más pequeñas de las imágenes a lo largo y ancho del mosaico (siempre que se mejore el parecido a la imagen "target" respecto a lo que teníamos hasta ahora). Al hacerse las imágenes más pequeñas ajustarán con mejor precisión los detalles del "target" y, como antes, veréis que tarda más en bajar del 5% que pusimos como umbral de parada.

Adjuntad mosaico final después de ejecutar el script para (72x48) y (36x24).

¿Cuántas iteraciones habéis necesitado en cada paso?

Es posible que para el tamaño más pequeño (36x24) se necesiten muchas iteraciones. Si os tarda mucho subir el umbral de salida a un 10% o poned un tope de unos 100 bucles de 200 pruebas, para un total de 20000.

De la última imagen seleccionar la zona del ojo y la ceja de la derecha de la foto y mostrad un detalle de ella.

Haceros una foto vuestra en primer plano, pasarla a B/W usando `rgb2gray()` y hacer un mosaico a partir de ella usando vuestro programa. Adjuntad la imagen original y el mosaico resultante.

----- Hasta aquí 75% de la nota -----


MANEJO de VIDEOS en MATLAB

En este apartado vamos a ver como leer y acceder a la información de un video en MATLAB para escribir una sencilla aplicación de tracking. Para leer el video se usa el comando `VideoReader()` que recibe un fichero de video (avi, mp4, etc.) y crea un objeto tipo video del que luego podemos extraer la información que necesitamos:

```
obj=VideoReader('color.mp4'); fps=get(obj,'FrameRate'), NF=get(obj,'NumberOfFrames')
```

Sabemos que un video consiste en una sucesión de imágenes (frames). Este video tiene $NF=749$ frames, lo que a $fps=29.96$ frames por segundo da una duración total de $T=NF/fps \sim 25$ seg. Para leer el k-ésimo frame podemos usar la función `read`: `frame=read(obj,k)`; (según la versión de Matlab la función puede ser `readframe`).

Una vez extraído un frame es como una image, pudiendo visualizarlo con `image()`. Con un bucle como: `for k=1:200, frame=read(obj,k); image(frame); drawnow; end` leemos los 200 primeros frames del video y los mostramos uno tras otro, lo que nos muestra el video en una ventana de MATLAB. El video consiste en la imagen de una hoja en blanco con cuatro puntos de distintos colores formando un cuadrilátero. La cámara no está fija por lo que al moverse y cambiar de orientación durante el video, los 4 puntos 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.

Aunque podría automatizarse, para que sea más sencillo, la identificación inicial la vais a hacer manualmente. Para ello cargar el primer frame del video, visualizarlo con `imshow()` y usar el cursor de datos de la figura () para pinchar sucesivamente en los 4 puntos (**empezando por el punto negro y siguiendo en el sentido de las agujas del reloj: rojo, verde y finalmente cyan**). Usar la información que nos da el cursor de datos (posición X,Y) para rellenar la siguiente tabla. Como indicación os dejo rellenos los valores que yo obtengo para el punto negro.

Info	punto negro	Punto rojo	Punto verde	Punto cyan
X	490			
Y	320			
R	13			
G	10			
B	12			

El script suministrado `track_video.m` será el esqueleto de vuestro código. Abrirlo en el editor **y completar las líneas del inicio, terminando de dar valores a los vectores X,Y con los datos de la tabla** (las coordenadas X e Y de los 4 puntos).

Al correr el script se carga el primer frame y se superponen 4 puntos amarillos en las posiciones indicadas por X, Y. Si los habéis marcado bien deben caer sobre las 4 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 (arriba/izquierda). Los círculos pintados enseguida dejan de caer sobre los puntos al moverse éstos.

Nota sobre el código: Para que la visualización sea más eficaz, al llamar a `imshow()` y a `plot()` por primera vez se guardan los objetos gráficos creado (`im_obj, pp_obj, tt`) y en cada paso del bucle en vez de hacer nuevas llamadas a `imshow` y `plot`, simplemente se actualiza la información de dichos objetos gráficos: la propiedad '`Cdata`' para la imagen y las propiedades '`Xdata`', '`Ydata`' para el `plot`.

Para hacer el tracking de los puntos, usaremos su diferente color para identificarlos y seguirles la pista. Para ello se usará la variable `color(3x4)` donde guardaremos las 3 componentes RGB de los colores de los 4 puntos. Terminar de completar sus valores en el script con las componentes RGB de los tres puntos restantes a partir de los datos de la tabla: cada columna debe ser el color de uno de los 4 puntos.

Usando esta información podremos detectar que píxeles son probablemente de la marca y calcular el centroide de su posición `x` e `y`, lo que nos dará la nueva posición en cada frame. El lugar para añadir este código de "seguimiento" está marcado por el comentario indicando **% Bucle actualizando posiciones X,Y de las 4 esquinas** dentro del bucle (`for k=1:Nf`) que barre los `Nf` frames. Si actualizamos los valores de las coordenadas `X` e `Y`, para que reflejen las nuevas posiciones en cada frame, veremos cómo los puntos amarillos acompañan a los marcas durante todo el video.

En una aplicación de tracking (una vez que se dispone de una primera hipótesis) se usa el hecho de que entre dos frames pasa muy poco tiempo ($\sim 1/30$ seg), por lo que los puntos no se habrán movido mucho desde la posición anterior. Por lo tanto, la búsqueda de la nueva posición de las marcas puede limitarse a una subimagen pequeña (de \pm RAD píxeles) alrededor de la posición anterior. En el script hemos definido este radio `RAD=25`. Una vez definido este parámetro, como paso previo (fuera del bucle) creamos un par de matrices `dx`, `dy` con las coordenadas de una subimagen referidas a su centro:

```
dx=ones(2*RAD+1,1)*(-RAD:RAD);
dy=(-RAD:RAD)'*ones(1,2*RAD+1);
```

Este proceso de búsqueda lo repetiremos para todos los puntos, dentro del bucle desde `j=1:4` para detectar y actualizar la posición de los 4 puntos. En cada punto la posición de partida vendrá dada por `X(j),Y(j)`. El algoritmo paso a paso sería:

1. Las componentes del color correspondiente a la `j`-ésima marca son (`r0,g0,b0`) siendo `r0=col(1,j)`; `b0=...` `g0=...` Para la marca `j`, el centro de la subimagen a explorar está en las coordenadas `x=round(X(j))`, `y=round(Y(j))`, y el rango de coordenadas a explorar será `rx=x+(-RAD:RAD)` y `ry=y+(-RAD:RAD)`.
2. Extraer la subimagen del frame actual en el rango de coordenadas anteriores y extraer su componente roja, guardandola como `R`. Repetir con los canales verde y azul (`G`, `B`). No olvidaros convertir a `double` para poder hacer cuentas.
3. Calcular `dr=(R-r0)/T`, la diferencia entre el canal rojo y la componente roja `r0` de referencia, dividida por `T=10`. Repetir con el canal verde y azul (`g0`, `b0`) para obtener `dg` y `db`. `T` es un umbral que indica cuánto lo que tenemos que apartarnos del color de referencia antes de que `dr`, `dg`, `db` sean > 1 .

4. Calcular: $w = \exp(-(dr^2 + dg^2 + db^2))$. La matriz w tendrá valores altos (~ 1) para aquellos píxeles donde $dr, dg, db \sim 0$, esto es, donde el color de dichos píxeles (en los tres canales) sea parecido al color de referencia (r_0, g_0, b_0) buscado.
5. Normalizar w dividiéndola por la suma de sus valores, que podéis calcular con $\text{sum}(w(:))$. Así obtenemos una matriz de pesos, que refleja la probabilidad de que los píxeles de la subimagen pertenezcan a la marca buscada.
6. Multiplicar (punto a punto) la matriz $(x+dx)$ de las coordenadas de los píxeles de la subimagen por la matriz de pesos w y sumar todos sus valores. Para sumar los valores de una matriz A sin tener que usar bucles haced $\text{sum}(A(:))$. El resultado será la estimación de la coordenada x del "centro" de la marca en ese frame. Guardar dicha posición en $X(j)$. Repetir con las coordenada y 's, guardando el resultado en $Y(j)$.

De esta forma, al terminar el bucle (for $j=1:4$) tendremos en los vectores X, Y las posiciones actualizadas de los 4 puntos. Si todo va bien el video debe mostrar ahora las círculos amarillos siguiendo la posición de las marcas a lo largo de los frames.

[Adjuntad código del bucle en j con la actualización de las posiciones X,Y](#)

Muy posiblemente observaréis que tras unos cuantos frames los puntos amarillos se vuelven inestables, moviéndose mucho y estando cerca de perder el "tracking". [¿En qué rango de frames es más aparente este comportamiento?](#) En mi caso, a pesar de estos problemas, el tracking aguanta hasta el final del video. Dependiendo de lo cuidadosos que hayáis sido al elegir las posiciones y colores iniciales, esta falta de estabilidad puede causar que alguno o varios de los puntos pierdan su marca. [Indicad si éste es o no vuestro caso. Si es así adjuntad una foto de un frame donde se haya perdido el tracking de alguno de los puntos.](#)

----- **Hasta aquí 90 % de la nota** -----

El problema es que durante todo el video se ha usado como identificador de las marcas su color original (guardado en col). Sin embargo, el color capturado en el video puede cambiar debido a cambios de iluminación, un ángulo distinto de observación, etc. Tras un rato el color puede ser lo suficientemente diferente del original como para perder el "tracking".

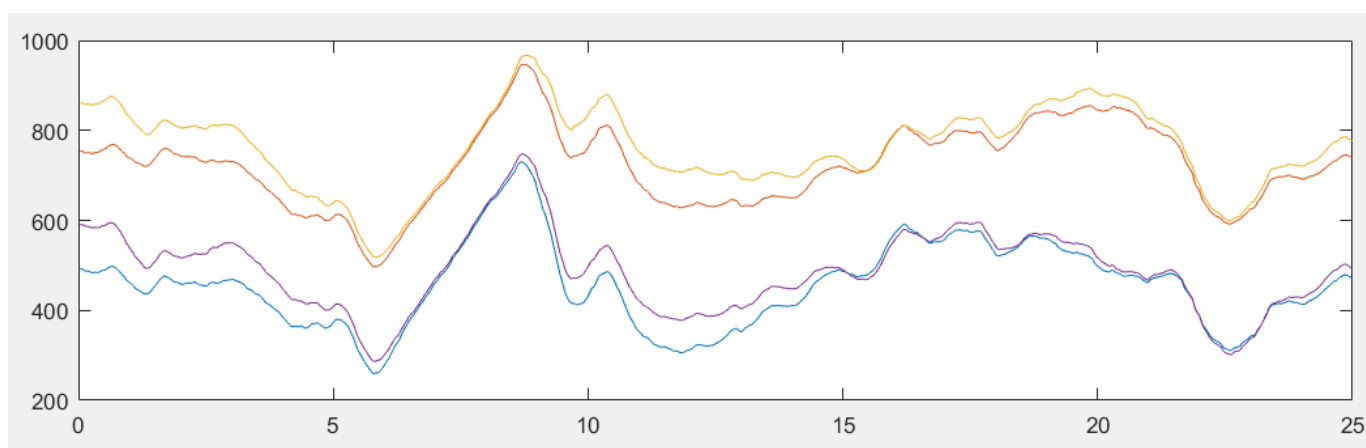
La solución es actualizar también el color de referencia al mismo tiempo que se actualiza la posición del punto. De hecho, la misma matriz de pesos w usada para calcular la posición puede usarse para recalculer el color de referencia.

Si multiplicamos los valores del canal rojo (R) por la matriz de pesos w y sumamos todos sus valores como hicimos antes para las coordenadas x, y obtenemos la media del canal rojo para los puntos que hemos detectado como parte de la marca. Usad ese valor para actualizar el color de referencia $col(1,j)$ (la componente roja del punto j). Repetir con los planos de color G y B , para completar la actualización de $col(2,j)$ y $col(3,j)$. Esto debe hacerse tras la actualización de $X(j), Y(j)$ para actualizar el color de los 4 puntos.

Adjuntad vuestro código con la actualización del color de referencia. Si ejecutáis de nuevo el script, el tracking debería verse más estable.

Buscad al final del bucle buscad la orden `set(pp_obj, 'Xdata'...);`. Comentadla y sustituirla por `hold on; plot(X,Y, 'y.');` `hold off` para ir dejando un "rastro" de por dónde se han ido moviendo los puntos. [Adjuntad foto final.](#)

En el código actual solo guardamos (en X,Y) las coordenadas de los 4 puntos en el último frame. Inicializar dos nuevos vectores U y V (NF x 4) y usarlos para guardar las coordenadas {x,y} de los puntos a lo largo de todo el video. Al final del bucle podremos superponer en un plot la evolución de ambas coordenadas frente al tiempo (en seg) para los 4 puntos. La figura adjunta muestra mi resultado para la coordenada U (x's). Observar como la trayectoria seguida por las 4 marcas, aunque similar, no es idéntica. En el tema siguiente veremos cómo estas diferencias pueden usarse para deducir la posición de la cámara.



Adjuntad vuestros propios "plots" correspondientes a las coordenadas V (y's') para el caso SIN y CON estabilización. Indicad sobre los plots en qué tiempo del video se notan más los efectos de la estabilización en la trayectoria.

Adjuntar código completo de vuestro script track_video con todas las modificaciones indicadas