



**POLITÉCNICA**

# **Panorámicas y Características de la Imagen**

**Fundamentos de Análisis de Imágenes**

**Eder Tarifa Fernandez - 21c011**

**David Hernando González - 21c27**

**Mario García Berenguer - 21c025**

## **Parte 1: Detección de objetos planos basada en la homografía.**

### **Descripción del problema**

En este primer problema, partimos de dos imágenes; una imagen de una escena y otra de un objeto que aparece también en la escena. El objetivo es que dada una tercera imagen de otro objeto, del mismo tamaño que el original, podamos reemplazarlo en la escena, haciendo parecer que este nuevo objeto pertenece a ella.

### **Resolución del problema**

Para realizar nuestro objetivo, lo primero que hemos hecho ha sido detectar y describir los puntos característicos de nuestras imágenes de partida. Para ello hemos utilizado diferentes algoritmos de detección de esquinas y manchas.

A continuación, hemos utilizado estos descriptores para buscar las correspondencias que existen entre nuestras dos imágenes, lo que conocemos como *matches*. Para ello también hemos utilizado diferentes estrategias de búsqueda de correspondencias.

Una vez tenemos las correspondencias entre las dos imágenes, hemos utilizado el algoritmo RANSAC para conseguir la matriz que corresponde a la homografía a realizar a la imagen de nuestro objeto para que adquiera la forma y ubicación con la que se representa en la imagen de la escena. Esta homografía es la clave de nuestro problema, ya que, aplicándola a una nueva imagen de otro objeto, se realizan todas las transformaciones necesarias para insertarla en la escena.

Por último, tenemos que crear una máscara con la forma de la nueva imagen transformada. Esta máscara tiene que tener todos los píxeles que aparecen en la posición del objeto a insertar con valor 0 (color negro) y todos los demás en 255 (color blanco), para que se elimine la información que aparece en la posición de la escena original que queremos insertar el nuevo objeto.

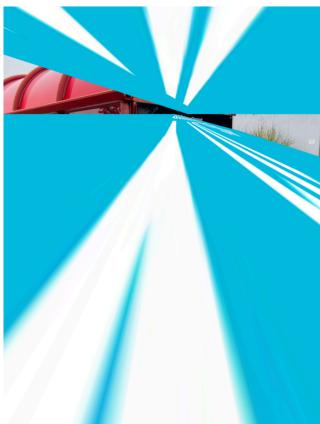
Para terminar, después de aplicar esta máscara a la escena original, combinamos la escena modificada con nuestra tercera imagen transformada y conseguimos el resultado que buscábamos.

## Problemas que hemos tenido para alcanzar la solución

El primer problema que nos ha surgido ha sido al aplicar la homografía a la imagen del nuevo objeto, se aplicaba correctamente pero obteníamos una imagen más pequeña de lo que salía en la escena. Para solucionarlo nos hemos dado cuenta de que esta nueva imagen tenía un *shape* diferente al de la imagen con el objeto original. Esta era la razón por la que se realizaba la homografía pero el resultado era diferente. Por tanto, hemos modificado la imagen para que tuviera el mismo tamaño que la original.

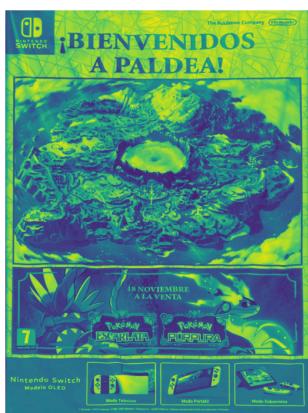
Seguidamente, hemos tenido problemas con la máscara que teníamos que aplicar a la imagen original, y la raíz del problema ha sido que queríamos crear esta máscara a partir del resultado de aplicar la homografía a la nueva imagen. En este punto ya teníamos la forma de la máscara y solo teníamos que cambiar el valor de los píxeles. Por tanto hemos decidido poner todos los píxeles que pertenecían al objeto en negro y todos los demás en blanco. Esto lo realizamos usando cv2.threshold y operaciones como cv2.bitwise\_not, sin embargo no obteníamos en resultado deseado, ya que, en la parte del objeto había píxeles que se quedaban con valores como [255, 255, 0], por ejemplo, lo que hacía que tomarán colores como rojo o azul, y no el blanco o negro que buscábamos. Por tanto, hemos creado una máscara de otra forma totalmente distinta; partiendo de una imagen totalmente blanca, le hemos aplicado la homografía que habíamos creado, obteniendo una máscara totalmente blanca en la parte del objeto y totalmente negra en el resto de la imagen, por tanto, solo hemos tenido que aplicar cv2.bitwise\_not a la máscara para obtener el resultado que buscábamos.

A la hora de utilizar diferentes algoritmos, en la detección de características usando Fast o Mser y usando el descriptor Sift, hemos tenido el problema de que al aplicar *Brute Force* y cualquier estrategia de correspondencias, obtenemos el error “trainDescCollection[iIdx].rows < IMGIDX\_ONE in function cv::BFMatcher::knnMatchImpl” que parece ser que las imágenes son demasiado grandes para aplicar estos algoritmos, ya que hemos comprobado el tipo y tamaño de los descriptores y puntos característicos y no había ningún problema. Además, al aplicar *Best radius matches* obtenemos el siguiente fallo tanto al usar *Brute Force* como *FLANN* “Failed to allocate 50373356000 bytes in function ‘cv::OutOfMemoryError’” en todos los algoritmos menos con Orb, vemos que es un problema de almacenamiento porque obtenemos demasiados resultados en los descriptores. Hemos intentado utilizar *Radius match* con Orb, pero incluso poniendo una distancia máxima de 100 no encontramos ninguna correspondencia. Al poner una distancia como 300 conseguimos que funcione, pero el resultado es erróneo:



Por todo esto, ya sea por un mal resultado o por unos descriptores demasiado grandes, se nos ha hecho imposible utilizar *Radius match*.

Antes de comentar los resultados, destacamos que no hemos utilizado el algoritmo Harris ya que al leer las imágenes en *GRAYSCALE* obtenemos este resultado, algo que vemos claramente que no está en blanco y negro.



Aún así, hemos probado a hacer la detección de esquinas con este algoritmo pero los resultados son los mismo que las imágenes que se muestran arriba, algo que tiene sentido, ya que, el algoritmo de detección de esquinas de Harris se basa en ver cómo cambian los niveles de gris de una ventana. Además, intentamos aplicar la homografía y obtenemos algo sin sentido.

Por otro lado, después de leer la imagen con *GRAYSCALE*, le hemos aplicado *cv2.COLOR\_GRAY2RGB* y en este caso sí que nos aparecen las imágenes en blanco y negro, algo que no tiene ningún sentido porque las transformamos de gris a rgb. Sin embargo, obtenemos errores de código que, como nos podíamos esperar, nos dicen que las imágenes no tienen el formato adecuado.

Por todo esto, hemos decidido no utilizar este algoritmo para este ejercicio.

## Resultados

Hemos utilizado diferentes detectores de características, descriptores y estrategias de correspondencias. Además, hemos intentado utilizar tanto *Brute Force* como *FLANN*, pero por algunos problemas previamente mencionados no ha sido posible y hemos utilizado la única alternativa posible. En las columnas de la tabla, se muestra en primera posición el algoritmo de detección de características y, en la segunda, el descriptor. En el caso de las filas, tenemos el algoritmo de búsqueda de correspondencias y la estrategia de emparejamiento, en segundo lugar. Estos han sido los resultados obtenidos en cuanto al tiempo de ejecución (en segundos) y las abreviaciones usadas en la tabla:

Sift: S      Fast: F      Mser: M      Orb: O  
Brute Force: BF      FLANN: FL  
Best match: BM      Best K matches: KNN

		S	S	F	S	M	S	O	O
BF	BM	12.824		-		-		0.278	
BF	KNN	12.252		-		-		0.254	
FL	BM	3.878		6.599		8.550		-	
FL	KNN	3.647		6.388		8.518		-	

Ahora pasamos a ver los resultados obtenidos para cada caso:

### SIFT

**BF-BM**



**BF-KNN**



**FL-BM**



**FL-KNN**



### FAST-SIFT

**FL-BM**



**FL-KNN**



**MSER-SIFT**

**FL-BM**



**FL-KNN**



**ORB**

**FL-BM**



**FL-KNN**



En primer lugar y lo que vemos a primera vista, los resultados que conseguimos con Orb no son los que buscábamos. Si vemos los tiempos obtenidos, Orb es el más rápido, pero los resultados están mal, por tanto, los tiempos son irrelevantes.

Por otro lado, en el caso de Sift, en ambos casos en los que se utiliza *Best K matches* no se obtiene un resultado muy bueno en las esquinas del objeto. En los otros dos casos de Sift, utilizando *Best matches* tanto con *Brute Force* como con *FLANN* conseguimos un buen resultado.

Por último, en los casos en los que usamos Fast o Mser como detectores de características, nos pasa lo mismo con Sift, *Best K matches* no nos da un buen resultado del problema. Sin embargo, los resultados con FLANN y *Best match* son los que buscábamos.

En conclusión, para este problema no debemos usar ni Orb ni la estrategia de correspondencias *Best K matches*, ya que nos dará una mala solución al problema. Por otro lado, teniendo en cuenta los tiempos de los algoritmos y que los demás resultados para el problema son buenos, nos quedamos con el detector y descriptor de características Sift, con el algoritmo FLANN y la estrategia de búsqueda de correspondencias *Best Match*, combinación con la que obtenemos una buena solución al problema en 3.878 segundos.

## **Parte 2: Une cinco imágenes para crear una panorámica.**

### **Descripción del problema**

En este segundo ejercicio partimos de cinco imágenes consecutivas horizontalmente con partes repetidas entre imágenes. El objetivo es conseguir una panorámica de la escena superponiendo las cinco imágenes de izquierda a derecha.

### **Resolución del problema**

Para resolver este problema, lo primero que hemos hecho nuevamente ha sido extraer los puntos clave y descriptores de las cinco imágenes mediante el algoritmo SIFT. Consecutivamente, hemos utilizado estos descriptores para buscar las correspondencias que existen entre cada par de imágenes consecutivas.

Con todo esto, nos hemos apoyado nuevamente en el algoritmo RANSAC para conseguir las cuatro matrices que corresponden a las homografías de los cuatro pares de imágenes, con la pequeña variación de que hemos agregado un desplazamiento para mejorar la precisión de la estimación de la homografía.

Lo siguiente que hemos hecho ha sido crear una función que superpone dos imágenes consecutivas. Lo primero que hace esta función es crear una máscara que detecta los píxeles negros de la imagen1, después crea la máscara inversa para quedarse con el contenido de la imagen, y gracias a esta última máscara inversa extrae su contenido (área que no es negra). Gracias a bitwise\_and extrae el fondo de imagen2 que coincide con la región negra de imagen1. Y por último, combina las áreas seleccionadas de imagen1 e imagen2, colocando el contenido de la imagen1 sobre el fondo relevante de imagen2, permitiendo que el contenido de una imagen se superponga sobre la otra.

Lo siguiente que se hace, es una vez hemos indicado cuál es la imagen central (en nuestro caso 2), calculamos las transformaciones necesarias de cada imagen según sus homografías y del lado de la imagen al que pertenezcan, multiplicando para las más alejadas por la matriz intermedia.

Y por último, aplicamos estas homografías a las imágenes sobre un fondo negro y las vamos superponiendo de izquierda a derecha.

## **Problemas que hemos tenido para alcanzar la solución**

Lo cierto es que nos han surgido varios problemas a la hora de realizar este ejercicio, el primero fue un problema con la proyección, ya que no se hacía como queríamos y no sabíamos si era por la homografía o por la función ‘warpPerspective’ que no la estábamos implementando bien, por lo que creamos la lista ‘transformations’, el objetivo de estas transformaciones es asegurar que todas las imágenes se proyecten y alineen correctamente en relación con la primera imagen. Cada transformación se aplica en orden, construyendo gradualmente la panorámica al incorporar cada imagen y ajustarla para que se integre de manera coherente con las anteriores.

Otro desafío al crear nuestra imagen panorámica fue el de la aparición de espacios negros entre las fotos proyectadas. Identificamos que este problema estaba relacionado con la superposición de imágenes, y para solucionarlo, implementamos una función llamada ‘overlay’. Esta función utiliza máscaras para fusionar las imágenes de manera suave. La máscara se crea mediante el método cv2.inRange para identificar los píxeles negros en la primera imagen. Luego, aplicamos operaciones bitwise para combinar las áreas válidas de ambas imágenes, eliminando así los espacios negros y superponiendo las fotos de manera efectiva. Gracias a esta función conseguimos crear una imagen panorámica final sin interrupciones visuales.

Y por último, otro problema fue el de eficiencia computacional, ya que este ejercicio puede ser computacionalmente muy costoso, por lo que intentamos optimizar el código lo máximo posible para que su tiempo de ejecución fuera mínimo.

## Resultados

Basándonos en el problema anterior, hemos decidido realizar este problema utilizando como único descriptor Sift, ya que con el descriptor binario Orb no conseguimos la solución que queremos para el problema. Así mismo, hemos utilizado los mismos detectores y estrategias de correspondencias que en el ejercicio anterior. De la misma forma, hemos utilizado *Brute Force* o *FLANN* dependiendo en caso, como se ha explicado en el primer problema. En las columnas de la tabla, se muestra en primera posición el algoritmo de detección de características y, en la segunda, el descriptor. En el caso de las filas, tenemos el algoritmo de búsqueda de correspondencias y la estrategia de emparejamiento, en segundo lugar. Estos han sido los resultados obtenidos en cuanto al tiempo de ejecución (en segundos) y las abreviaciones usadas en la tabla:

Sift: S      Fast: F      Mser: M  
Brute Force: BF      FLANN: FL  
Best match: BM      Best K matches: KNN

		S	S	F	S	M	S
BF	BM	16.569		-		-	
BF	KNN	15.297		-		-	
FL	BM	13.893		19.589		17.147	
FL	KNN	12.147		18.989		16.19	

Ahora pasamos a ver los resultados obtenidos para cada caso:

### SIFT

#### BF-BM

#### BF-KNN



**FL-BM**



**FL-KNN**



**FAST-SIFT**



**FL-BM**

**FL-KNN**



**MSER-SIFT**



**FL-BM**

**FL-KNN**



En los resultados gráficos de este ejercicio, vemos que quitando el caso de Mser-Sift y Sift-Sift usando KNN con *FLANN*, obtenemos un buen resultado en todos los casos. En estos dos casos vemos que la parte en la que aparece el autobús en la imagen no queda bien conectada.

En los demás resultados, vemos que todos nos podrían servir como uno bueno del ejercicio. Sin embargo y al igual que nos ocurría en el ejercicio anterior, el resultado que obtenemos con Sift-Sift y *Brute Force* con KNN es el más rápido de todos. Por tanto, en caso de tener que quedarnos con un único resultado, sería con este, por ser uno bueno y el más eficiente.

## **Parte 3: Unir 7 imágenes en una nueva proyección para crear una panorámica**

### **Descripción del problema**

Este problema era similar al propuesto en el ejercicio 2. Sin embargo, al tener tantas imágenes para crear la panorámica no podíamos proyectar sobre un plano, ya que se vería muy distorsionado el resultado, debíamos proyectar sobre una superficie curva.

Este ejercicio no hemos conseguido completar al 100% el resultado mostrado por el profesor. Sin embargo, hemos intentado alcanzar hasta él por dos caminos distintos, obteniendo dos resultados bastante óptimos y distintos.

### **Resolución del problema (Parte 1)**

Para resolver este ejercicio en primer lugar hemos tenido que definir una función llamada *cylindricalWarp* que nos permite proyectar las 7 imágenes que tenemos sobre una superficie cilíndrica, de forma que no se distorsionen tanto al intentar juntarlas.

Una vez disponíamos de las 7 imágenes proyectadas en superficies cilíndricas, hemos creado un bucle para montar las imágenes de derecha a izquierda. En este bucle nos encargamos de obtener los keypoints y los descriptores usando el SIFT, ya que nos parece un algoritmo muy fiable al ser invariante. También se calculan los mejores matches, a través del método de fuerza bruta y aplicando KNN como estrategia en lugar de bestmatches. Esto nos ha permitido obtener mejores resultados a la hora de encontrar las coincidencias. Nos quedamos con aquellos matches cuya distancia a su vecino más cercano sea menos de la mitad. Una vez hemos calculado los puntos que coinciden podemos calcular la homografía y aplicarla, modificando los parámetros de tamaño para permitirnos que ambas fotos quepan en la misma imagen. También modificamos la imagen de destino para que muestre las dos imágenes superpuestas.

Cuando todo el bucle es ejecutado, obtenemos una panorámica del conjunto de las imágenes.

## **Problemas que hemos tenido para alcanzar la solución**

Durante este ejercicio hemos tenido un montón de problemas que han provocado que no hayamos sido capaces de llegar a la mejor solución.

En primer lugar, hemos tenido un problema al aplicar la homografía a la imagen, y es que si intentábamos aplicar la homografía a una imagen colocada a la izquierda de otra, esa imagen se veía recortada y al final no se obtenía una panorámica. Hemos intentado solucionarlo añadiendo márgenes a las imágenes antes de aplicar la homografía para que los puntos que se mandaran fuera de la imagen se mandaran dentro en su lugar, pero no ha funcionado y se seguía recortando la imagen. Como solución hemos decidido aplicar las homografías a las imágenes de la derecha. Esto ha provocado que las imágenes tiendan a estar muy pequeñas en la parte derecha y mucho más grandes en la izquierda, como ya veremos en los resultados. Para intentar apaciguar este efecto hemos diseñado una última transformación aplicada a la imagen final, que intenta estirar la imagen para apaciguar este efecto de zoom.

En segundo lugar, hemos tenido de nuevo otro problema con las homografías. En este caso viene dado por los puntos que se utilizaban para calcularlas. La razón de este problema es que los métodos que estábamos usando para obtener los matches no generaban suficientes puntos, los matcheaba mal o generaba muchos matches en una zona en concreto pero el resto de las imágenes no obtenemos ninguno. Es por esto que nos hemos visto obligados a no usar bestmatch como método, sino KNN, ya que nos ha proporcionado más coincidencias sobre las que calcular la homografía y así obtener una mejor proyección.

También relacionado con el primer problema, hemos tenido muchos problemas al construir la panorámica se nos dibuja con una líneas negras entre medias de las imágenes. Esto es debido a que, como ya hemos comentado, cuando aplicamos la homografía se elimina la parte izquierda de la foto a la que se la aplicamos. Estas rayas negras debería ser parte de la foto, pero al eliminarse partes por la homografía se quedan como márgenes negros. Si decidimos eliminarlos y juntar las fotos el resultado es que hay una parte de la foto que no está bien acoplada con el resto. Por esto hemos tenido que modificar las imágenes al aplicar la transformación cilíndrica para evitar que se dejará ningún margen, y así que la homografía no trabajara con ninguno, lo que permite que las imágenes se acoplen mucho mejor.

## **Resultados (Parte 1)**

En este ejercicio hemos necesitado muchas transformaciones en las imágenes para poder llegar a una buena solución.

En primer lugar, hemos necesitado aplicar la proyección cilíndrica, a la que más tarde le hemos tendido que eliminar los bordes negros que dejaba:



En segundo lugar hemos ido aplicando las homografías y proyectando sobre las imágenes. Hemos comenzado por las imágenes de la derecha:



Vemos que a medida que añadimos más imágenes, la zona de la derecha va disminuyendo su tamaño:



Hasta finalmente obtener la panorámica entera:



Claramente el tamaño del lado izquierdo no está nivelado con el tamaño del lado derecho, por lo que aplicamos una última transformación para nivelar un poco la panorámica:



Y este sería el resultado final de nuestra panorámica. Claramente está mejor representada que el resultado final de la aplicación de las homografías. El resultado se podría mejorar si en lugar de tener que empezar por una de las esquinas lo pudiéramos hacer por el medio y acoplar imágenes a ambos lados, pero como ya hemos comentado, esto no es posible ya que la homografía corta nuestra imagen.

### Resolución del problema (Parte 2)

En esta segunda solución, hemos optado por una técnica más parecida al segundo ejercicio de la práctica. De forma que hemos extraído los puntos clave y descriptores de las siete imágenes proyectadas sobre superficies cilíndricas de la solución anterior mediante el algoritmo SIFT. Consecutivamente, hemos utilizado estos descriptores para buscar las correspondencias que existen entre cada par de imágenes consecutivas y hemos utilizado el algoritmo RANSAC para conseguir las seis matrices que corresponden a las homografías de los seis pares de imágenes.

Con todo esto, calculamos las transformaciones necesarias para cada imagen con el objetivo de que se superpongan adecuadamente unas a otras y las mostramos. Este es el resultado obtenido:



## Optional: Eliminate the condition of naming images in increasing order

### **Descripción del problema**

El objetivo de este problema es crear una panorámica superponiendo imágenes nuevamente, pero en este caso sin especificar el orden. Este enfoque añade una complejidad adicional, ya que no conocemos la secuencia correcta de las imágenes y debemos determinar la alineación óptima de manera dinámica.

### **Resolución del problema**

Para la resolución de este problema, lo primero que hemos hecho nuevamente ha sido extraer los puntos clave y descriptores de las cinco imágenes mediante el algoritmo SIFT. Consecutivamente, hemos utilizado estos descriptores para buscar las correspondencias que existen entre cada par de imágenes posibles entre todos.

Con todo esto, nos hemos apoyado nuevamente en el algoritmo RANSAC para conseguir las diez matrices que corresponden a las homografías de los diez pares de imágenes.

Una vez tenemos todos los 10 pares de imágenes con sus matches, vamos a suponer que los 4 con más inliers significará que son imágenes consecutivas al tener mayor número de matches. Obtenemos el índice de estos 4 pares mediante la función ‘argsort’ y los mostramos.

Lo siguiente que hacemos es intentar identificar cuál es el orden correcto de las imágenes. Para ello, creamos un diccionario en el que las claves son el índice de la imagen y los valores el número de veces que aparece en los 4 mejores pares de imágenes calculados anteriormente. De esta forma, podemos identificar cuáles serán las 2 imágenes de los extremos y cuáles las 3 centrales, ya que la de los extremos sus valores serán 1, al sólo aparecer en un par y las centrales 2, al aparecer en 2 pares. Añadimos las 2 imágenes laterales a una lista, al igual que los índices pertenecientes al par de imágenes en el que aparecen estas imágenes laterales. Ambas listas las usaremos a continuación.

Sabemos cuáles son las dos imágenes laterales, pero no cuál es la izquierda y cuál la derecha, por lo que apoyándonos en las listas anteriores creamos un código que vaya comparando las dos imágenes izquierdas de los pares a los que pertenecen con las dos imágenes laterales, de forma que cuando coincidan tendremos identificada nuestra imagen izquierda. Lo mismo para la derecha.

De esta forma, tendríamos identificadas la primera y última imagen, pero nos faltarían las centrales. Para identificar cuál es cada una vamos a aplicar la lógica, de forma que el índice de la imagen que se encuentra entre la central y la de la izquierda del todo, será la que aparezca en el par de imágenes junto a la de la izquierda del todo, ya que esta imagen solo aparecía en un par de imágenes y sólo puede ser con esta imagen. Lo mismo se aplicaría para la que está entre la central y la de la derecha del todo.

Por lo tanto, tendríamos identificadas a todas las imágenes menos la central. Para saberlo, simplemente iteramos sobre los índices de las imágenes y le asignamos el que no esté asignado todavía por ninguna de las otras 4. Ya con los índices, los ordenamos de izquierda a derecha.

Y por último, aplicamos las homografías calculadas en el ejercicio anterior, que las reutilizamos (ya que serían las mismas que si las volviésemos a calcular con las imágenes en orden) para evitar el exceso coste computacional, a las imágenes sobre un fondo negro y las vamos superponiendo de izquierda a derecha.

## Resultados

Los resultados que hemos ido obteniendo a lo largo de este ejercicio son los siguientes:

Los matches correctos de los 10 pares de imágenes posibles:



Cabe destacar que se ve a simple vista cuáles son los 4 correctos, pero nosotros hemos preferido hacerlo de una forma automática para casos más dudosos.

Los 4 pares de imágenes consecutivas:



Las imágenes ya ordenadas:



Y el resultado final:



## **Problemas que hemos tenido para alcanzar la solución**

El mayor desafío al que nos hemos enfrentado en este ejercicio ha sido el de saber la posición de cada imagen, para calcularlo hemos ido probando y fallando numerosos diccionarios, listas, bucles... hasta que hemos dado con la tecla para alinear las imágenes sea cuál sea el orden de entrada.