# Implementación de panoramas lineales con proyección en superficies cilíndricas o esféricas y mezcla de colores usando el algoritmo de Burt-Adelson

**Braulio Valdivielso Martínez, Gema Correa Fernández**

[1]Departamento de Ciencias de la Computación e I.A – Escuela Técnica Superior de Ingenirías Informática y de Telecomunicaciones (UGR)

{gecorrea, bvaldivielso}@correo.ugr.es

***Resumen.*** *Este documento describe la confección de panoramas lineales con proyección en superficies cilíndricas o esféricas con transiciones suaves entre las imagenes usando el algoritmo de Burt-Adelson. Implementaremos ambas proyecciones, tanto la cilíndrica como la esférica y una vez implementadas, elegiremos la que menos deformación global proyecte para así crear la panorámica. Para conseguir una interpolación suave entre los colores de imágenes tomadas en condiciones de luminosidad potencialmente distintas adaptaremos el algoritmo de Burt-Adelson a la confección de panoramas de una forma que no hemos visto reflejada en la literatura. Para realizar el proyecto nos hemos basado principalmente en los documentos [BA83a; BA83b; Sze10].*
***Palabras-clave****: Burt-Adelson, panorama, proyección, esférica, cilíndrica.*

## 1. Definición del Problema

Nuestro proyecto tiene como objetivo la creación de panoramas de alto ángulo de apertura y con transiciones suaves entre las imágenes. Como vimos en la práctica 2, la creación de panoramas consiste en la concatenación *inteligente* de imágenes adyacentes, obteniendo así una imagen que contenga a todas las anteriores. Sin embargo, algunos problemas a los que la técnica vista en clase no podía enfrentarse son los siguientes:

1. La construcción de panorámicas con gran ángulo de apertura: las deformaciones que resultan de proyectar sobre una colección de imágenes de una escena determinada no son muy notables cuando se tiene una sola imagen o un mosaico con un ángulo de apertura moderado. Una vez se tienen imágenes representando una apertura de más de $120^o$ aproximadamente las distorsiones comienzan a arruinar la calidad del resultado [Lev]. Tomaremos un enfoque alternativo para sobreponernos a este problema que consiste en la proyección de las imágenes tomadas sobre una superficie bien cilíndrica o bien esférica, y posteriormente utilizaremos las proyecciones habituales de estas superficies sobre el plano que representa nuestra imagen final.

2. La fusión de imágenes con distinta luminosidad: mientras que la técnica vista en clase era geométricamente satisfactoria, pudimos comprobar que no se comportaba de manera ideal desde el punto de vista fotométrico. Entonces, al unir las

imágenes adyacentes, podían encontrarse pequeñas (aunque notables) variaciones en los colores en las fronteras entre imágenes colindantes. Para resolver este problema usaremos el algoritmo de Burt-Adelson, fusionando los colores en los límites de las imágenes que componen el mosaico, realizando así una transición más suave.

De acuerdo con esta exposición, el problema que tratamos de resolver es la creación de panorámicas a partir de imágenes que representen un alto grado de apertura y tomadas en condiciones de luminosidad potencialmente distintas. Como hemos dicho antes, usaremos la proyección cilíndrica o esférica y el algoritmo de Burt-Adelson para intentar solventar estas dificultades. En las siguientes secciones comentaremos detalladamente cada uno de los componentes de nuestra solución.

## 2. Implementación del Problema

En este apartado, explicaremos las técnicas usadas para la implementación del problema, así como los resultados experimentales obtenidos y las posibles valoraciones o conclusiones alcanzadas. Comenzaremos realizando la proyección de una imagen sobre una superficie cilíndrica o esférica y escogiendo una de ellas para la creación de la panorámica. Posteriormente trataremos la implementación del algoritmo de Burt-Adelson para corregir las variaciones de color que podamos encontrar en la frontera entre las imágenes.

### 2.1. Proyección en Superficies Cilíndricas o Esféricas

La técnica usada en la segunda práctica para la realización de los mosaicos deja de producir resultados satisfactorios cuando se enfrenta a mosaicos con un amplio ángulo de apertura. En su momento no notamos especialmente las deformaciones introducidas por nuestro algoritmo, pero basta con tomar otro conjunto de imágenes para que estas se vuelvan evidentes.

**Figura 1. Mosaico deformado por la técnica de la práctica 2**

El ejemplo de la figura 1 es especialmente bueno para comprobar las deformaciones porque podemos tomar como referencia la línea blanca, representando el círculo central del terreno de juego. La geometría original de la imagen se ha perdido (difícilmente podría decirse que la línea blanca proviene de una circunferencia a menos que se estuviera acostumbrado a este tipo de deformaciones) y más importantemente aun: seguir extendiendo el mosaico para incluir más imágenes a los lados, introduciría deformaciones más drásticas. Nótese que estas imágenes representan una panorámica de menos de 180º. Para obtener este mosaico hemos utilizado nuestro algoritmo de panorámicas (con Burt-Adelson incluido) sin realizar previamente ningún tipo de proyección sobre las imágenes originales.

A lo largo del trabajo veremos cómo las proyecciones cilíndrica y esférica ayudarán a mejorar los resultados. Las imágenes originales para realizar nuestros experimentos las hemos obtenido de [Sti].

En la figura 2 podemos observar dos de estas imágenes originales.

(a)

(b)

**Figura 2. Ejemplo de imágenes originales usadas a lo largo de este proyecto**

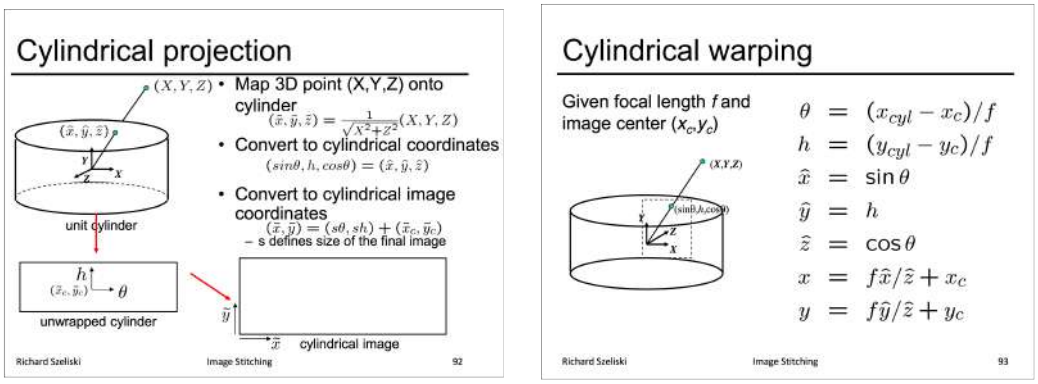
Para poder extender la panorámica a ángulos superiores proyectaremos las imágenes individualmente sobre una superficie cilíndrica (en primer lugar). El objetivo será realizar el emparejamiento entre las imágenes situadas en la superficie del cilindro y una vez se tengan así *desenrollar* el cilindro para obtener una superficie plana. Ha de observarse que por esta proyección, arcos de circunferencia centradas en el punto donde se sitúa la cámara se transformarán en líneas rectas al proyectar la superficie del cilindro sobre el plano. En esta línea observamos la figura 2 a la que le hemos realizado una proyección cilíndrica evidenciando la transformación entre arcos de circunferencia y segmentos rectos. En el siguiente apartado discutiremos qué parámetros determinan dichas proyecciones y qué se ha tenido en cuenta para encontrar la utilizada en la figura 2. Por lo que la forma de manejar este problema es deformando las imágenes mediante una proyección cilíndrica o esférica. En la figura 3 podemos ver una de las imágenes de la figura 2, en dónde la línea del centro del campo aparece aproximadamente recta.

**Figura 3. Proyección cilíndrica sobre la figura 2. Obsérvese que la circunferencia del centro del campo aparece recta.**

A continuación, explicaremos como construir las imágenes proyectadas sobre un cilindro o una esfera [Urt]. Como podemos ver en [Bla], nuestro objetivo será convertir una imagen sobre un plano a una superficie cilíndrica o esférica, jugando con las posibles distancias focales. Más adelante profundizaremos en estos conceptos.

### 2.1.1. Proyección Cilíndrica

Usaremos esta proyección si la cámara está nivelada y solo tenemos rotación alrededor de su eje vertical (que es un modelo que aproxima la mayor parte de panorámicas que se toman), por lo que solo deberemos considerar variaciones en un ángulo. Para realizar la proyección sobre la superficie cilíndrica acudimos a la información que nos proporcionan los apuntes de teoría [Bla] en la figura 4.




(a) Proyección cilíndrica (b) Deformación cilíndrica

**Figura 4. Coordenadas para la proyección sobre superficie cilíndrica**

Así que haciendo uso de la figura anterior y, complementando tal información con las fórmulas obtenidas en [Sze10], podemos obtener ya las coordenadas para la proyección sobre una superficie cilíndrica:

$$\text{Coordenada } X: \quad \boldsymbol{x' = r \cdot tan^{-1} \frac{x - x_{centro}}{f} + x_{centro}}$$

$$\text{Coordenada } Y: \quad \boldsymbol{y' = r \cdot \frac{y - y_{centro}}{\sqrt{(x + x_{centro})^2 + f^2}} + y_{centro}}$$

Siendo:

- $(x, y)$: las coordenadas de la proyección original
- $(x', y')$: las nuevas coordenadas para la proyección cilíndrica
- $(x_{centro}, y_{centro})$: las coordenadas del centro de la imagen
- $r$: el radio del cilindro
- $f$: la distancia focal. Este parámetro junto con $r$ controlará lo drástico de la deformación de la proyección. Si además utilizamos la recomendación de [Sze10] y hacemos $r = f$ tenemos que este será el único parámetro libre.

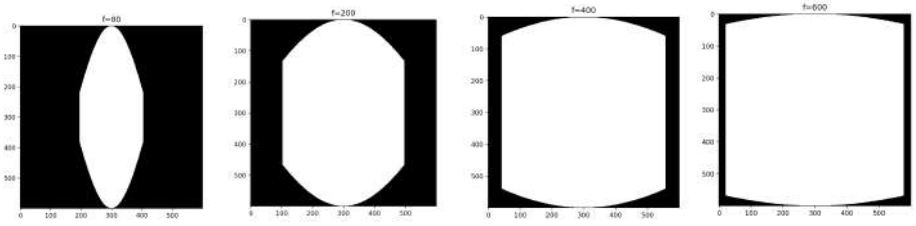
Nuestra implementación es `cylindrical_warp`: función que proyecta una imagen sobre una superficie cilíndrica. Esta función tiene por parámetros de entrada una imagen y la distancia focal. Dentro de la función definimos `convert_coordinates`, función que implementa las fórmulas explicadas anteriormente y que a partir de una distancia focal y las coordenadas del pixel $(x', y')$ en la imagen proyectada nos devuelve las coordenadas $(x, y)$ en la imagen original. Creamos un lienzo negro del mismo tamaño que la imagen original y actualizamos cada pixel de coordenadas $(x', y')$ con el valor del pixel en las coordenadas $(x, y)$ dadas por nuestra función `convert_coordinates` en la imagen original. El código es equivalentemente explicativo:

```
def cylindrical_warp(img, f=1200):
    """Performs a cylindrical warp on a given image"""

    # obtener las coordenadas de la proyeccion cilindrica
    # libro Szeliski: formula 9.13 y 9.14
    def convert_coordinates(new_point, new_shape, f, r):

        # x' = r * arctang((x - x_c) / f) + x_c
        # y' = r * ((y - y_c) / ((x - x_c)**2 + f**2)) + y_c

        y, x = (
            new_point[0] - new_shape[0]/2,
            new_point[1] - new_shape[1]/2
        )

        new_y = r * (y / sqrt(x**2 + f**2))
        new_x = r * np.arctan(x / f)

        return (
```

```
            floor(new_y) + new_shape[0]//2,
            floor(new_x) + new_shape[1]//2
        )

    # nos creamos un lienzo para pegar la
    # proyeccion cilindrica de la imagen
    new_img = np.zeros(img.shape, dtype=np.uint8)

    # recorremos la imagen de entrada (matriz)
    for row_index in range(len(img)):
        for col_index in range(len(img[0])):

            # convertimos las coordenadas de la imagen
            y, x = convert_coordinates(
                (row_index, col_index),
                img.shape[:2],
                f,
                f )

            new_img[y, x] = img[row_index, col_index]

    return new_img
```

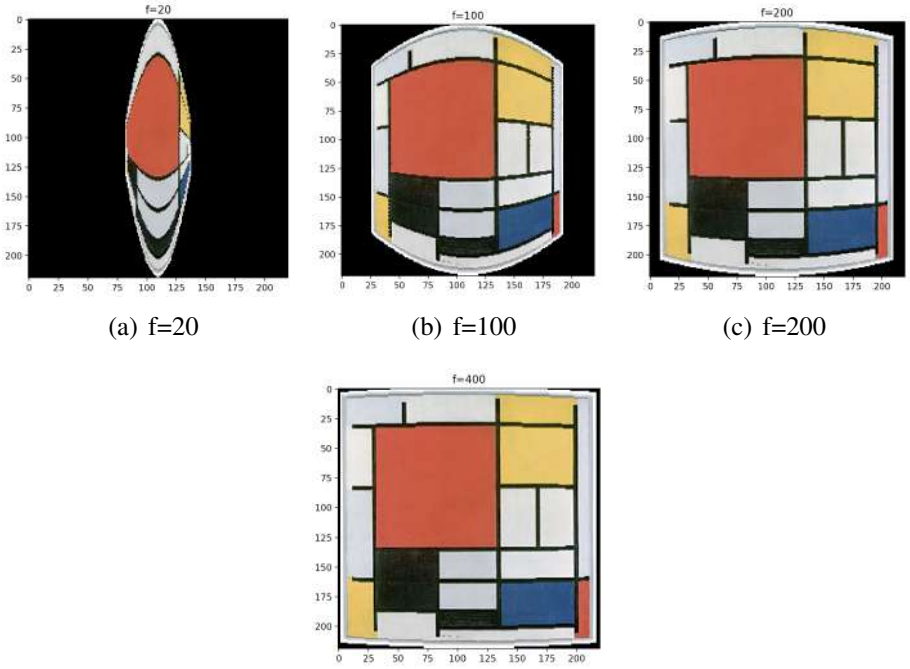
Ya solo falta comprobar el resultado obtenido, para ello visualizaremos varios ejemplos con dicha proyección usando distintas distancias focales. Tales visualizaciones las encontramos en las figuras 5 y 6. Notamos que como estamos proyectando sobre un cilindro, es lógico que la parte del centro de la imagen sea la que más se ensanche. Además comprobamos que conforme más lejos se encuentre la cámara más ancho será la visión del cilindro donde proyectamos.

En las figuras 5 y 6 podemos ver respectivamente una imagen completamente blanca y un cuadro de Mondrian a la que le hemos aplicado proyecciones cilíndricas con distintas distancias focales para apreciar la influencia de la distancia focal en el resultado final. Entenderemos distancia focal, como el campo de visión. Por lo que al variar la distancia focal conseguiremos un menor o mayor acercamiento, modificando así el campo de visión. Entonces, al aumentarla nos acercaremos y al reducirla nos alejaremos. La distancia focal ideal para nuestro ejemplo del campo de fútbol que vimos en la figura 3 es la que transforma el arco de circunferencia en una recta. Esta distancia focal depende de la cámara utilizada para realizar la foto, pero nosotros la hemos obtenido probando manualmente hasta encontrar un valor que nos pareció adecuado.



(a) f=80 (b) f=200 (c) f=400 (d) f=600

**Figura 5. Imagen 600x600 proyectada sobre superficie cilíndrica**



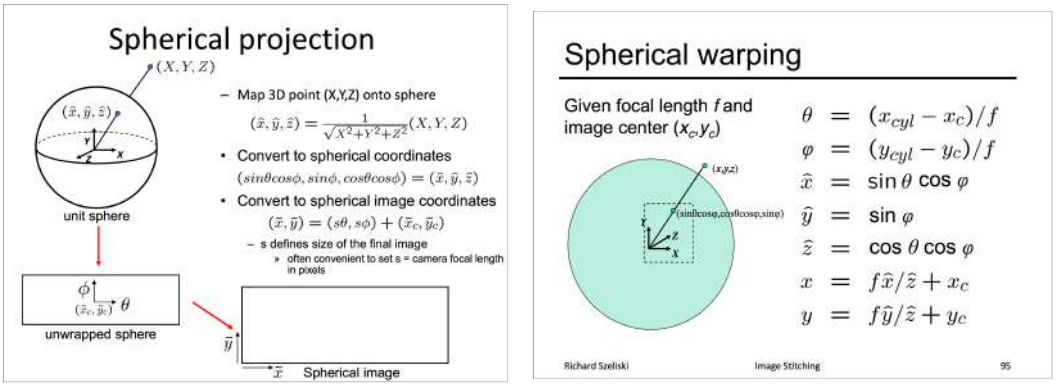

(a) f=20 (b) f=100 (c) f=200

(d) f=400

**Figura 6. Imagen 220x219 proyectada sobre superficie cilíndrica**

### 2.1.2. Proyección Esférica

Para la proyección esférica usaremos la misma idea que para la proyección cilíndrica, simplemente que ahora las fórmulas para obtener las coordenadas de tal proyección serán distintas [Bla]. De igual manera, calcularemos la correspondencia entre coordenadas deformadas y mapeadas [Urt]:




(a) Proyección esférica (b) Deformación esférica

**Figura 7. Coordenadas para la proyección sobre superficie esférica**

Indistintamente, haremos uso de la figura previa y completaremos tal información

con las fórmulas obtenidas en [Sze10]. Con esto, obtenemos las siguientes coordenadas:

$$\text{Coordenada } X: \quad \boldsymbol{x' = r \cdot tan^{-1} \frac{x - x_{centro}}{f} + x_{centro}}$$

$$\text{Coordenada } Y: \quad \boldsymbol{y' = r \cdot tan^{-1} \frac{y - y_{centro}}{\sqrt{(x + x_{centro})^2 + f^2}} + y_{centro}}$$

Siendo:

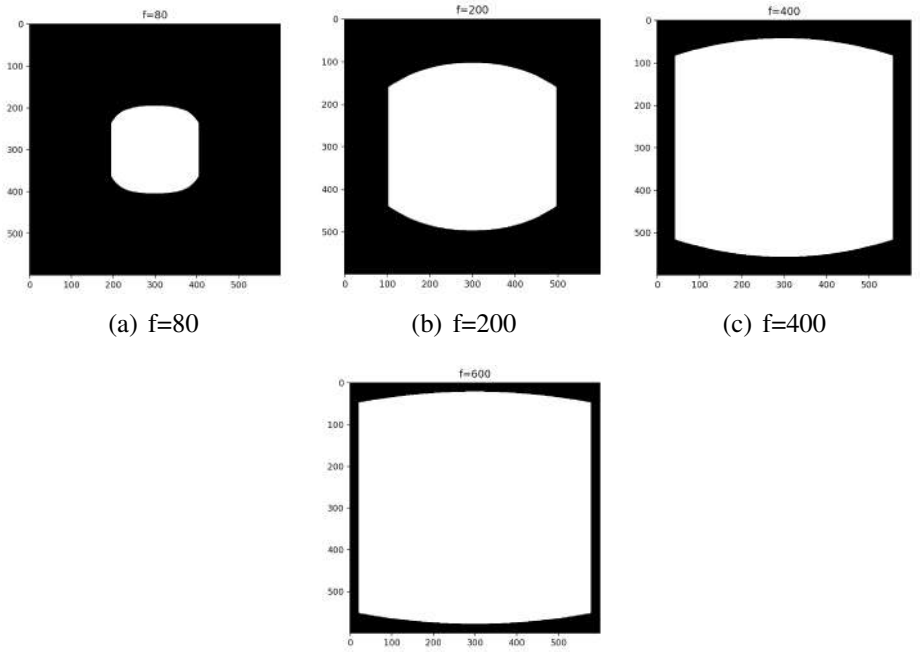
- $(x, y)$: las coordenadas de la proyección original
- $(x', y')$: las nuevas coordenadas para la proyección esférica
- $(x_{centro}, y_{centro})$: las coordenadas del centro de la imagen
- $r$: el radio de la esfera.
- $f$: la distancia focal. Este parámetro junto con $r$ controlará lo drástico de la deformación de la proyección. Si además utilizamos la recomendación de [Sze10] y hacemos $r = f$ tenemos que este será el único parámetro libre.

De forma análoga a `cylindrical_warp` tenemos `spherical_warp`, que proyecta una imagen sobre una esfera y cuyos parámetros de entrada son una imagen y la distancia focal a usar dentro de la proyección. Tal como hicimos para `cylindrical_warp`, dentro de esta función definimos otra función `convert_coordinates` que implementa las fórmulas explicadas anteriormente para obtener las coordenadas en la imagen original de un punto proyectado. Una vez tenemos esta función creamos un lienzo negro sobre el que proyectaremos la imagen. Para cada pixel de coordenadas $(x', y')$ obtenemos las coordenadas en la imagen antes de proyectar $(x, y)$ mediante `convert_coordinates` y actualizamos el valor del pixel $(x', y')$ conforme al valor en la imagen original. Una vez más, y como recomendación en [Sze10], usaremos el mismo valor para la distancia focal que para el radio del cilindro ($r = f$) ya que da menos distorsión.

```
def spherical_warp(img, f=1200):
    """Performs a spherical warp on a given image"""

    # obtener las coordenadas de la proyeccion esferica
    # libro Szeliski: formula 9.18 y 9.19
    def convert_coordinates(new_point, new_shape, f, r):

        # x' = r * arctang((x - x_) / f) + x_
        # y' = r * arctang( ((y - y_c) / ((x - x_centro)**2 + f**2)) )
           + y_centro

        y, x = (
            new_point[0] - new_shape[0]/2,
            new_point[1] - new_shape[1]/2
        )

        new_y = r * np.arctan( y / sqrt(x**2 + f**2))
```

```
        new_x = r * np.arctan(x / f)

        return (
            floor(new_y) + new_shape[0]//2,
            floor(new_x) + new_shape[1]//2
        )

    # nos creamos un lienzo para pegar la
    # proyeccion esferica de la imagen
    new_img = np.zeros(img.shape, dtype=np.uint8)

    # recorremos la imagen de entrada (matriz)
    for row_index in range(len(img)):
        for col_index in range(len(img[0])):

            # convertimos las coordenadas de la imagen
            y, x = convert_coordinates(
                (row_index, col_index),
                img.shape[:2],
                f,
                f
            )

            new_img[y, x] = img[row_index, col_index]

    return new_img
```

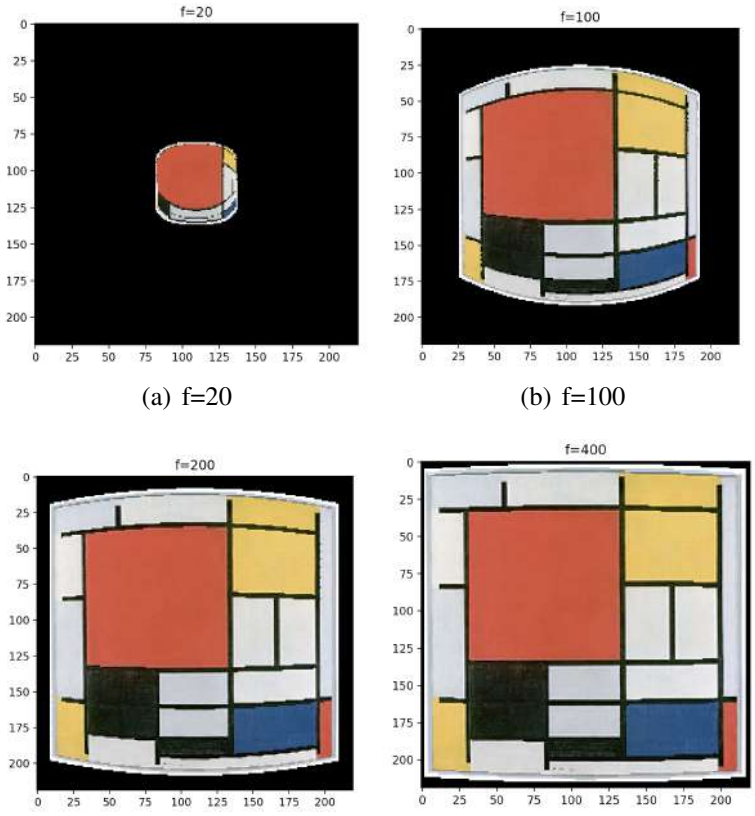
Ya solo falta comprobar el resultado obtenido. Para ello visualizaremos varios ejemplos con dicha proyección usando distintas distancias focales.




(a) f=80 (b) f=200 (c) f=400

(d) f=600

**Figura 8. Imagen 600x600 proyectada sobre superficie esférica**



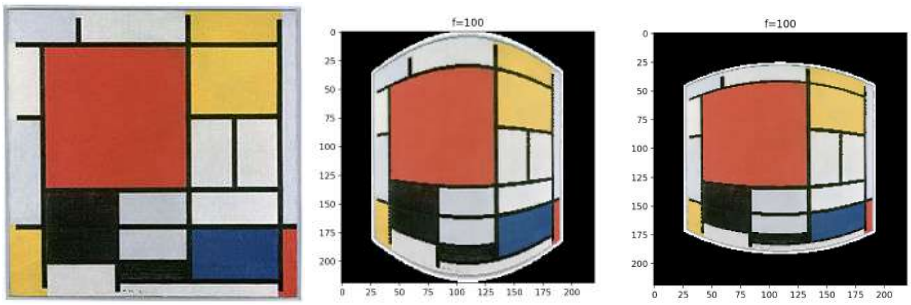

(a) f=20 (b) f=100

(c) f=200 (d) f=400

**Figura 9. Imagen 220x219 proyectada sobre superficie esférica**

Como se aprecia en las figuras 8 y 9, al proyectar sobre una esfera, obtenemos prácticamente la misma salida, sin embargo, comprobamos que esta imagen produce menos distorsión global en la imagen. A continuación, explicamos las diferencias más detalladamente.De igual manera, obtenemos la misma conclusión para la utilización de la distancia focal.

### 2.1.3. Proyección Cilíndrica vs. Proyección Esférica




(a) imagen original (b) proyección cilíndrica (c) proyección esférica

**Figura 10. Comparación proyección cilíndrica y esférica**

Como comentamos, la proyección esférica produce menos deformación global (figura 10). Para una misma distancia focal, dicha imagen, proporciona dos salidas distintas (b) y (c), donde ambas producen deformaciones sobre la imagen original. Sin embargo, la imagen (b), produce una mayor deformación global a la imagen debido a que estira verticalmente los objetos a medida que se acercan a los polos norte y sur. Y sin embargo, ese problema no lo tenemos con la (c). Pero este problema, disminuría conforme aumentamos la distancia focal. Sin embargo, si tuviéramos que decantarnos por una proyección con la que generar una panorámica para aplicar el algoritmo de Burt-Adelson, elegiríamos la esférica. Pero indistintamente podríamos haber escogido la otra proyección.

## 2.2. Algoritmo de Burt-Adelson

El otro componente fundamental de nuestro proyecto es la interpolación suave entre imágenes utilizando el algoritmo de Burt-Adelson. Este algoritmo lo utilizaremos para conseguir transiciones suaves entre las imágenes que componen el panorama. Trataremos de exponer en primer lugar la utilización del algoritmo de Burt-Adelson de forma independiente a la construcción de panoramas de acuerdo con los recursos [BA83a; BA83b] y posteriormente concretaremos su utilización en nuestro proyecto.

Tal como se expone [Opec] en el problema de *Image Blending* (combinar dos imágenes de forma que las transiciones entre ellas sean suaves) merece la pena considerar las distintas bandas de frecuencia de las imágenes con las que trabajamos. Las bandas de frecuencias más altas de nuestra imagen pueden ser combinadas abruptamente sin que apenas notemos la transición. Sin embargo, si tratamos de realizar una mezcla tan abrupta en las bandas de frecuencias más bajas de las imágenes en seguida notaremos la transición. Por este motivo Burt y Adelson decidirán trabajar con las pirámides laplacianas de las imágenes en cuestión para obtener la interpolación deseada.

Aprovechamos para introducir el ejemplo que presentaremos a lo largo de esta sección, que se encuentra en toda la literatura al respecto de *Image Blending*: la manzana y la naranja. Para poner en perspectiva lo bien que funciona el algoritmo de Burt-Adelson mostramos en la figura 11 el resultado de realizar la mezcla de imágenes de forma ingenua. Hemos definido una máscara binaria que determina de qué imagen hay que coger cada píxel en la imagen final y no hemos realizado ningún proceso más de interpolación. La frontera entre ambas imágenes salta a la vista.

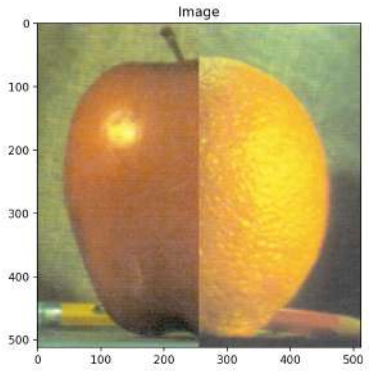



**Figura 11. *Image Blending* sin utilizar Burt-Adelson**

Supongamos ahora que queremos mezclar mediante el algoritmo de Burt-Adelson dos imágenes $A$ y $B$ de las mismas dimensiones. Sea $M$ una máscara binaria que especifica qué parte de la imagen final queremos que provenga de $A$ y cuál de $B$: interpretaremos un $1$ en la máscara como que tal pixel en la imagen final ha de provenir de $A$ y un $0$ como que tal pixel provendrá de $B$. Nótese que la máscara binaria ha de tener por tanto las mismas dimensiones que $A$ y que $B$.

El algoritmo se basa en la combinación de las pirámides laplacianas de dos imágenes según esta determinada máscara binaria. Veremos que se tomarán pirámides guaussianas de una máscara que en principio contiene regiones conexas y disjuntas de 1s y 0s. Esto provocará que en cada versión de más baja frecuencia (en los sucesivos niveles de la pirámide gaussiana) encontremos saltos más suaves de la zona de 1s a la zona de 0s, sirviendo de interpolación suave para las componentes de más baja frecuencia de las pirámides laplacianas de las imágenes $A$ y $B$, de acuerdo con la intuición que vimos en [Opec].

Los pasos a seguir para la construcción de la mezcla son:

1. Construir las pirámides laplacianas $LA$ y $LB$ de $k$ niveles de las imágenes $A$ y $B$ respectivamente.
2. Construir la pirámide gaussiana $GM$ de $k$ niveles de la máscara $M$.
3. Formar una pirámide combinada de $LA$ y $LB$, usando los nodos de $GR$ como pesos. Es decir, para cada $k$, $i$ y $j$:

$$LS_k(i,j) = GM_k(i,j,) * LA_k(i,j) + (1 - GM_k(i,j)) * LB_k(i,j)$$

4. Interpretando la suma $LS$ de pirámides laplacianas como una pirámide laplaciana en sí misma, podemos recomponer la imagen *original* colapsando la pirámide.

El código de la función con la que implementamos el algoritmo es tan sencillo como nuestra exposición inicial:

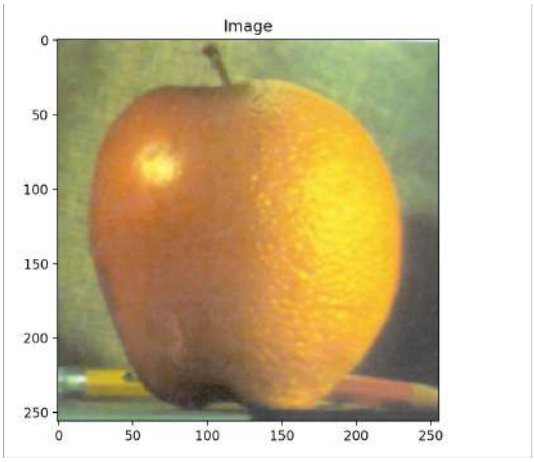
```
def blend(imgA, imgB, mask):
    """Forma una piramide combinada con A y B,
    usando los nodos de la mascara como pesos"""

    levels = 7
    gpMask = compute_gaussian(mask, levels)

    lAs = compute_laplacian(imgA, levels)
    lBs = compute_laplacian(imgB, levels)

    lSs = []

    # LS_k(i,j) = GM_k(i,j,)*LA_k(i,j) + (1-GM_k(i,j))*LB_k(i,j)
    for lA, lB, G in zip(lAs, lBs, gpMask):
        lSs.append(G*lA + (1-G)*lB)

    return spline(lSs)
```

Para explicar el proceso mostramos las imágenes de la manzana y la naranja junto con la máscara que usamos para la mezcla.

(a) (b) (c)

**Figura 12. Imagenes para explicar el funcionamiento de Burt-Adelson**

El resultado final lo encontramos en 13. Nótese la diferencia con 11. Las comparaciones son odiosas.




(a)

**Figura 13. *Image Blending* usando Burt-Adelson**

A continuación detallaremos la implementación de cada uno de los pasos que hemos descrito anteriormente.

**Nota:** si la imagen leída es en color, deberemos aplicar el algoritmo Burt-Adelson, por cada canal. Realizaremos el ejemplo con la imagen leía en escala de grises.

### 2.2.1. Paso 1 - Paso 2: Construcción de Pirámides

Implementamos la construcción de las pirámides gaussianas y laplacianas de la imagen tal como vimos en la primera práctica. Las pirámides gaussianas, son un tipo de *collage*, en el cual se representa la misma imagen con distintas escalas. Para la construcción

de tales pirámides deberemos hacer un redimensionado y un alisamiento, ambas tareas serán llevadas a cabo por la función `pyrDown` de OpenCV, la cual difumina una imagen y disminuye la resolución de la misma. En la figura 14 vemos un ejemplo de ello, en donde al aumentar de nivel la franja que separa el blanco del negro se vuelve más difusa.

```python
def compute_gaussian(img, levels=7):
    """Realiza una piramide gaussiana para una imagen dada"""

    # guardamos el primer nivel de la piramide
    pyramid = [img]
    acc = img

    for _ in range(levels):
        # redimensionar y alisar
        acc = cv2.pyrDown(acc)
        pyramid.append(acc)

    return pyramid
```

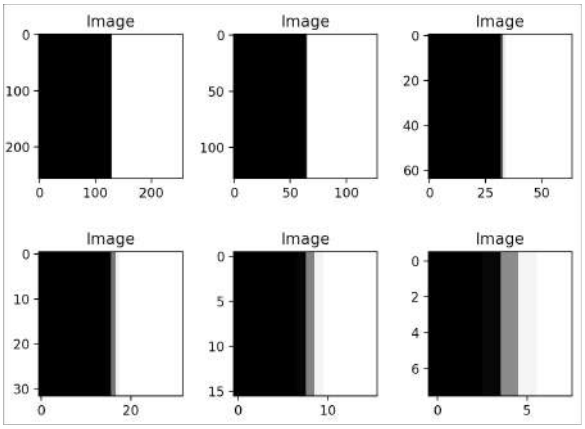
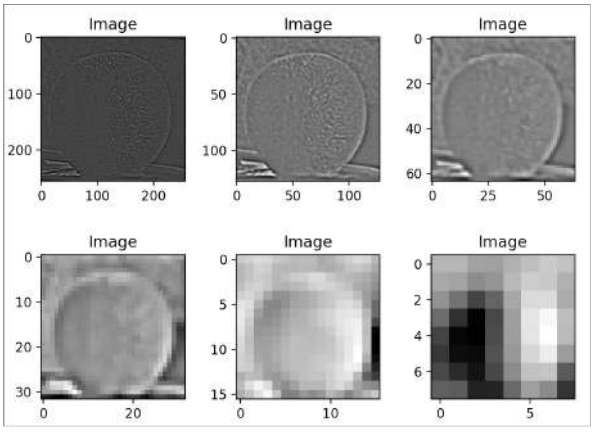



**Figura 14. Pirámide gaussiana para la máscara**

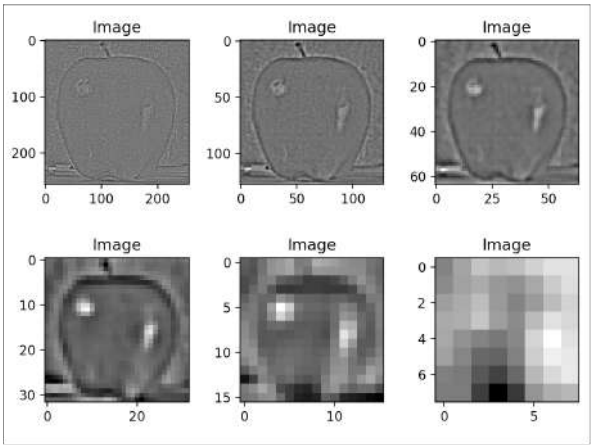
Utilizamos la pirámide laplaciana para poder trabajar con las distintas bandas de frecuencia de la imagen de forma sencilla. La implementación es análoga a la que se ha visto en clase: se obtiene la pirámide gaussiana y se define la componente $k$-ésima de la laplaciana como el componente $k$-ésimo de la gaussiana restándole el componente $k+1$-ésimo escalado hacia arriba. Tal como se comentó en clase, es fácil interpretar la implementación como obtener qué información de la imagen se está perdiendo con cada filtrado y muestreo. En la figura 17 vemos un ejemplo de la pirámide laplaciana para las imágenes de la manzana y de la naranja.

```python
def compute_laplacian(img, levels=7):
    """Realiza una piramide laplaciana para una imagen dada"""

    # obtener piramide gaussiana
    laplacian = compute_gaussian(img, levels)
```

```
    pyramid = [laplacian[levels-1]]

    for i in range(levels-1,0,-1):
        GE = cv2.pyrUp(laplacian[i])
        height, width = laplacian[i-1].shape
        L = cv2.subtract(laplacian[i-1],GE[:height, :width])
        pyramid.append(L)

    return pyramid[::-1]
```




(a) naranja




(b) manzana

**Figura 15. Pirámides laplacianas para las imágenes**

### 2.2.2. Paso 3 - Formar una pirámide combinada

Mezclamos los valores de las pirámides laplacianas de las imágenes, formando una nueva pirámide laplaciana, mediante la fórmula:

$$LS_k(i,j) = GM_k(i,j,) * LA_k(i,j) + (1 - GM_k(i,j)) * LB_k(i,j)$$

donde

1. **GM**: pirámide gaussiana de la máscara
2. **LA**: pirámide laplaciana de la imagen A
3. **LB**: pirámide laplaciana de la imagen B
4. **LS**: pirámide laplaciana mezclada

Nótese cómo esta fórmula es la media ponderada de ambas pirámides usando la pirámide gaussiana de la máscara como pesos. La implementación se comprende ahora de forma sencilla:

```
def blend(imgA, imgB, mask):
    """Forma una piramide combinada con A y B,
    usando los nodos de la mascara como pesos"""

    levels = 7
    gpMask = compute_gaussian(mask, levels)

    lAs = compute_laplacian(imgA, levels)
    lBs = compute_laplacian(imgB, levels)

    lSs = []

    # --- Inicio Paso 3 --- #
    # LS_k(i,j) = GM_k(I,j,)*LA_k(I,j) + (1-GM_k(I,j))*LB_k(I,j)
    for lA, lB, G in zip(lAs, lBs, gpMask):
        lSs.append(G*lA + (1-G)*lB)
    # --- Fin Paso 3 --- #

    # recomponemos la laplaciana lSs
    return spline(lSs)
```

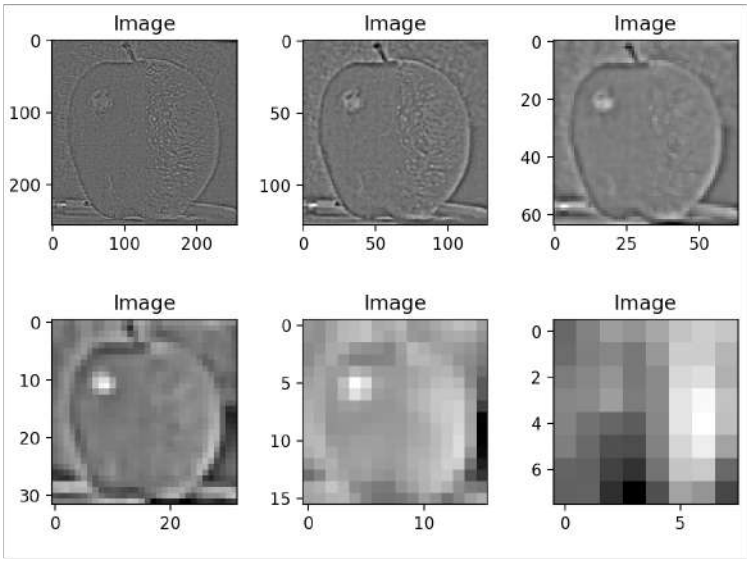



**Figura 16. Mezcla final en un canal**

Como se aprecia en la figura 16, obtenemos una pirámide laplaciana en donde cada nivel está formado aproximadamente por la mitad del nivel correspondiente a la primera imagen y la mitad de la otra. Gracias a la utilización de la pirámide gaussiana de la máscara las mezclas de las componentes laplacianas presentan interpolaciones más suaves conforme más bajas son las frecuencias que representan. Por último, solo nos queda recomponer esa pirámide en una imagen, obteniendo así la imagen solución. En la subsección siguiente, explicaremos cómo la función `spline(lSs)` realiza esta recomposición.

### 2.2.3. Paso 4 - Recomponer la imagen *original* colapsando la pirámide.

A partir de la pirámide que hemos sintetizado en el Paso 3, recuperamos una imagen utilizando el procedimiento habitual de reconstrucción de una imagen a partir de su pirámide laplaciana. Este método no va más allá de revertir los pasos en la construcción de la pirámide y empezando por el nivel más bajo de la laplaciana (que coincide con el de la gaussiana) escalamos hacia arriba y sumamos al siguiente nivel de la laplaciana. Este procedimiento lo realizamos inductivamente hasta obtener la imagen correspondiente.

```
def spline(pyramid):
    """Recomponer la imagen colapsando la piramide"""

    prev = pyramid[-1]

    for img in reversed(pyramid[:-1]):
        height, width = img.shape[:2]
        prev = cv2.pyrUp(prev)
        prev = img + prev[:height, :width]

    return prev
```

Ya tenemos terminado el algoritmo para cada canal de la imagen. Solo queda transformar la imagen original a `float` (es necesario para operar con algunas funciones de OpenCV) y llamar a nuestro algoritmo por cada canal. Finalizamos recomponiendo los tres canales en una sola imagen.

```
n_to_float = lambda img: img.astype(np.float)/255

def burt_adelson(imgA, imgB, mask):
    imgA = n_to_float(imgA)
    imgB = n_to_float(imgB)
    mask = n_to_float(mask)

    blended_channels = [
        blend(
            imgA[:, :, channel],
            imgB[:, :, channel],
            mask[:,:, channel]
        )
        for channel in range(len(imgA[0,0]))
```

```
    ]

    return np.array(blended_channels).transpose(1, 2, 0)
```

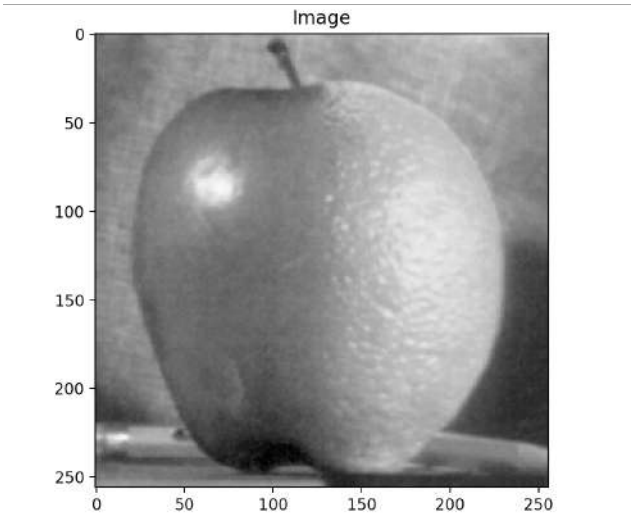



**Figura 17. Pirámides laplacianas para las imágenes**

Y como se observa se obtiene la misma imagen que la figura 13, pero ahora está en escala de grises.

### 2.3. Creación de Panoramas Lineales

Como ya tenemos el algoritmo Burt-Adelson implementado, solo nos queda implementar la creación de panoramas. Para ello, nos hemos basado en la implementación que realizamos en la *Práctica 2* para la generación de un panorama [Oped; Opea; Opeb]. Siguiendo con el trabajo, una vez que tenemos las imágenes proyectadas, debemos saber donde estarán colocadas en nuestro panorama, por tanto, realizaremos un mosaico de $N$ imágenes proyectadas sobre una superficie esférica. Primero, obtendremos con el descriptor-detector `SIFT`, los *keypoints* y descriptores de cada imagen para nuestro vector de imágenes. Luego, mediante fuerza bruta calcularemos los vectores de correspondencias entre las imágenes y una vez obtenidos, obtendremos las correspondencias entre las imágenes usamos un objeto de tipo `BFMatcher`. Para establecer tales correspondencias usaremos el criterio `BruteForce+crossCheck`, diciéndole a su constructor que use la normal $L2$ y que use `cross-check` (validación cruzada). Así que si tenemos $n$ imágenes, obtendremos $n - 1$ correspondencias, una por cada pareja de imágenes adyacente. Todo esto, estará metido un bucle, el cual recorra la lista de imágenes. Por tanto, los keypoints de la imagen $i$ estarán en correspondencia con los keypoints de la imagen $i + 1$. Ahora procedemos a calcular la matriz de homografía para cada par de imágenes y guardamos las homografías de imágenes adyacente.

```
def compute_homographies(imgs):
```

```
    """Calcula las homografias entre una lista de imagenes
       contiguas"""

    # descriptor SIFT
    sift = cv2.xfeatures2d.SIFT_create()

    # calculamos las correspondencias
    matcher = cv2.BFMatcher(
        normType=cv2.NORM_L2,
        crossCheck=True
    )

    keypoints = []
    descriptors = []
    homographies = []

    # obtenemos todos los keypoints y descriptores
    # asociados de las imagenes
    for img in imgs:
        kps, desc = sift.detectAndCompute(img, mask=None)
        keypoints.append(kps)
        descriptors.append(desc)

    keypoints = np.array(keypoints)
    descriptors = np.array(descriptors)

    # calculamos las homografias entre cada par
    # de imagenes contiguas
    for i in range(len(imgs) - 1):
        matches = matcher.match(descriptors[i], descriptors[i+1])

        keypoints_izq = np.array([
            keypoints[i][match.queryIdx].pt
            for match in matches
        ])

        keypoints_der = np.array([
            keypoints[i+1][match.trainIdx].pt
            for match in matches
        ])

        homography, _ = cv2.findHomography(
            srcPoints=keypoints_izq,
            dstPoints=keypoints_der,
            method=cv2.RANSAC,
            ransacReprojThreshold=1
        )

        homographies.append(homography)

    return homographies
```

Una vez que hemos calculado todas las homografías, podemos pasar al siguiente paso, que es el de comenzar a construir el mosaico. Lo primero de todo, creamos un lienzo negro cuyas dimensiones dependerán del número de imágenes. A continuación, seleccio-

namos la imagen del centro de nuestro vector de imágenes, y la trasladamos al centro de la imagen con fondo negro creada previamente. Luego iremos pegando las demás imágenes con la función `warpPerspective()` para sus respectivas imágenes y homografías.

Comenzamos definiendo algunas funciones que nos serán útiles durante la implementación del algoritmo. En primer lugar `translation_matrix` nos devuelve la matriz que representa a la homografía de traslación que lleva al punto $(0, 0)$ al $(x, y)$. Definimos también *perspective* que abstrae `warpPerspective` sobre algunos parámetros que dejaremos fijo a lo largo de todo el algoritmo.

```
def translation_matrix(x, y):
    """Devuelve la matriz de una traslacion"""
    return np.array([
        [1, 0, x],
        [0, 1, y],
        [0, 0, 1]
    ], dtype=float)
```

```
def perspective(src, M, size):
    b = np.zeros(size)
    """Realiza una homografia sobre una imagen de fondo negro"""
    return cv2.warpPerspective(
        src=src,
        M=M,
        dst=b,
        dsize=size[::-1],
        borderMode=cv2.BORDER_CONSTANT
    )
```

Una vez definido esto, tenemos que coser las imágenes que aparecen a la derecha y a la izquierda de la imagen trasladada al centro. Deberemos tener en cuenta, que no podemos crear un bucle y meter todas las imágenes de golpe en el mosaico, ya que hay que tener en cuenta las que están a la derecha o izquierda de la imagen trasladada. Por tanto, debemos crear primero la parte derecha y luego la parte izquierda:

- Para la parte izquierda: recorremos las imágenes desde la imagen a la izquierda de la del centro hasta la primera de todas. Llevaremos un acumulador que represente la homografía que se utilizará para coser la siguiente imagen. En cada iteración multiplicamos la homografía que lleva de la imagen $k$ a la $k+1$ por el acumulador. El acumulador inicialmente tendrá un valor igual a la matriz de traslación de la imagen central al centro del mosaico.
- Para la parte derecha: recorremos las imágenes desde la del $centro + 1$ hasta la última imagen de nuestro vector. Nos creamos un bucle que vaya multiplicando las homografías, ya que la imagen $i$ será la homografía $i$ al centro del mosaico. Sin embargo, ahora tendremos que hacer la inversa de las homografías (trabajamos con matrices regulares 3x3), ya que si antes la homografía $i$ llevaba la imagen $i$ a la imagen $i + 1$, ahora la inversa lleva la imagen $i + 1$ a la $i$. Esto hace que la imagen se cosa en el lugar que le toca.

Estas consideraciones servirían para implementar una versión básica del algoritmo de generación de panorámicas, pero el hecho de querer utilizar Burt-Adelson para realizar el

*blend* entre las sucesivas imágenes nos obligará a mantener el control de más información de cómo se están concatenando las imágenes.

La opción natural para incorporar Burt Adelson a nuestro algoritmo de generación de panoramas y la que se nos ocurrió en primer lugar consistía en llevar mediante la homografía calculada previamente, la imagen $k$-ésima a un canvas que estuviera completamente negro. Por otro lado mantendríamos en otro canvas de fondo negro todas las imágenes que ya han sido cosidas. Una vez tuviéramos estos dos lienzos podríamos realizar una interpolación entre ambos. El problema que quedaría resolver sería cómo elegir la máscara que seleccionaría qué parte de la imagen proviene del lienzo acumulado y cuál proviene del lienzo correspondiente a la nueva imagen que se trata de coser. Es imposible elegir esta máscara sin bordear zonas negras de alguno de los dos lienzos, algo de lo que no había que preocuparse en el ejemplo de la manzana y la naranja puesto que no había zonas negras en la imagen. Estas zonas negras junto con los filtros gaussianos provocan que las regiones que colindan ambas imágenes se *mezclen* también con estas zonas negras, resultando en un ligero oscurecimiento de los bordes de las imágenes. Este oscurecimiento no es muy notable en sí mismo pero teniendo en cuenta que el algoritmo de Burt-Adelson acabará siendo aplicado en repetidas ocasiones sobre la imagen central (cada vez que se cose una nueva imagen) la diferencia entre los bordes de varias imágenes del mosaico serán llamativos, dando al traste con la intención de realizar una interpolación suave en primer lugar.

Viendo que no conseguimos hacer funcionar ideas similares a la que tratamos en el párrafo anterior decidimos tomar otro camino. En lugar de realizar la interpolación entre los lienzos al completo trataríamos de seleccionar una región rectangular que estuviera contenida dentro de la intersección del lienzo acumulado (en el que se encuentran ya las imágenes anteriormente cosidas) con el lienzo a coser. En esa región rectangular habremos conseguido evitar las regiones negras que fastidiaban la interpolación con la idea ingenua. En la figura 18 mostramos las regiones de las que hablamos:

(a) lienzo-acumulado (b) lienzo-nuevo (c) mascara-acumulada

(d) mascara-nueva (e) mascara-interseccion

**Figura 18. Pirámides laplacianas para las imágenes**

El siguiente objetivo será encontrar la mayor región rectangular que se encuentre dentro de la máscara que representa la intersección de ambas imágenes. En esa región será donde realicemos la interpolación de Burt-Adelson. Resulta que encontrar la mayor región rectangular que se encuentra dentro de un polígono convexo (nuestra máscara intersección es un polígono convexo) no es un problema trivial de acuerdo con [Smi]. En cualquier caso fuimos capaces de encontrar una solución que aunque no es matemáticamente correcta (no da la respuesta precisa en la totalidad de los casos) ha funcionado bien en todos los ejemplos que hemos tratado. La implementación de nuestra solución para encontrar tal rectángulo se encuentra en `compute_largest_rectangle` y lo que hace es fijar la esquina superior izquierda del rectángulo 10 unidades hacia abajo y 20 hacia la derecha del punto más arriba a la izquierda de la máscara. Con las otras esquinas del rectángulo se opera de forma análoga.

```
def compute_largest_rectangle(mask):
    non_empty_cols = (mask > 0).any(axis=(0, 2))
    non_empty_rows = (mask > 0).any(axis=(1, 2))

    l = np.argwhere(non_empty_rows)
    first_row = l[0][0]
    last_row = l[-1][0]

    l = np.argwhere(non_empty_cols)
    first_col = l[0][0]
    last_col = l[-1][0]

    room = -10
    return first_row+room, last_row-room, first_col+room-20,
        last_col-room+20
```

Una vez tenemos una región rectangular de intersección entre las imágenes a las que queremos hacer interpolación, computamos una máscara que divida esa región por la mitad, de la misma manera que se hace en el ejemplo de la manzana y la naranja. Eso lo hacemos con la funcion `get_half_mask` que recibe las esquinas del cuadrado y llena la mitad izquierda de puntos blancos y la mitad derecha de puntos negros.

```
def get_half_mask(first_row, last_row, first_col, last_col):
    # mitad izquierda
    white = np.ones((
        (last_row-first_row),
        (last_col-first_col)//2
    )) *255

    # mitad derecha
    black = np.zeros((
        (last_row-first_row),
        (last_col-first_col) - (last_col-first_col)//2
    ))

    # las pegamos
    mask_to_mix = np.hstack(
        (black, white)
```

```
    ).astype(np.uint8)
    mask_to_mix = np.array([mask_to_mix]*3).transpose(1, 2, 0)

    return mask_to_mix
```

En la figura 19 podemos observar un ejemplo las regiones de los lienzos que se le pasarán como entrada al algoritmo de Burt-Adelson para realizar la interpolación.

(a) Sección del canvas acumulado (b) Sección de la imagen a coser (c) mascara de Burt-Adelson generada (d) `blend`

**Figura 19. Regiones del panorama que se harán pasar por Burt-Adelson**

A continuación, mezclaremos el canvas acumulado, la región mezclada por Burt-Adelson y la nueva imagen a coser en un solo lienzo:

```
# en el canvas resultado podemos pegar todo lo que tenemos
# fuera de la zona donde vamos a hacer burt-adelson
canvas_[:, :first_col+3] = tmp_canvas[:, :first_col+3]
canvas_[:, last_col-3:] = canvas[:, last_col-3:]

roi = float_image_to_uint8(
       burt_adelson(
        canvas_to_mix,
        tmp_canvas_to_mix,
        mask_to_mix
    )
)

# actualizamos el canvas con la zona burt-adelson
canvas_[first_row:last_row, first_col:last_col] = roi
```

donde `first_col` y `last_col` representan las coordenadas $x$ del rectángulo de intersección donde se va a hacer Burt-Adelson y `canvas_to_mix`, `tmp_canvas_to_mix` y `mask_to_mix` son las regiones que se muestran en la figura 19.

Un ejemplo que muestra el funcionamiento de la técnica implementada lo encontramos en la figura 20 en la que comparamos una panorámica realizada sin utilizar la mezcla

de Burt-Adelson con otra en la que usamos la técnica de Burt-Adelson adaptado a nuestro caso. Como se ve, se obtiene una mejor panorámica usando el algoritmo Burt-Adelson, ya que su no utilización implica que no haya una combinación suave entre imágenes adyacentes, puesto que se ve el corte de unión de las imágenes.

(a) Panorámica con Burt-Adelson

(b) Panorama sin Burt-Adelson

**Figura 20. Ejemplos de panorámicas usando pirámides de 7 niveles**

### 2.3.1. Experimentación con todo lo desarrollado en el documento

Ahora que tenemos todos los componentes listos podemos realizar algunas pruebas de nuestro algoritmo.

En la figura 21 observamos una panorama a cuyas fotos se le ha aplicado previamente una proyección cilíndrica. Nótese también el cambio de saturación gradual que hemos impreso en las imágenes en 22 para poner de manifiesto lo suave de las transiciones.

(a) panorámica+burt-adelson de 7 niveles+proyección con `f=1200`

**Figura 21. Ejemplos de panorámicas**

(a) imagen 1 `f=1200` (b) imagen 2 `f=1200`

(c) imagen 3 `f=1200` (d) imagen 4 `f=1200`

**Figura 22. Imágenes utilizadas para la confección del mosaico en 21**

El ejemplo más sorprendente que hemos conseguido generar es sin embargo el que encontramos en la figura 23. Aquí podemos ver cómo la técnica desarrollada sirve en efecto para la confección de panorámicas con un alto ángulo de apertura. Para la obtención de las imágenes originales que hemos usado en esta panorámica hemos vuelto a recurrir a [Sti]. Compárese esa figura con la figura 24 y nótese que la deformación de la proyección cilíndrica deja fuera de la panorámica a la bicicleta de la derecha.

**Figura 23. Panorámica de una plaza utilizando proyección esférica con `f=1000` y Burt-Adelson de 7 niveles.**

**Figura 24. Panorámica de una plaza utilizando proyección cilíndrica con `f=1000` y Burt-Adelson de 7 niveles.**

## 3. Consideraciones Finales

A lo largo del documento, hemos ido explicando las diferentes conclusiones que obteníamos con cada apartado. Pero observando las figuras 23 y 20 podemos decir que hemos obtenido un buen resultado. El algoritmo Burt-Adelson sí aplica una mejora significativa a la hora de crear un panorama con imágenes con iluminación diferente.

Además, también hemos podido comprobar, que las imágenes con gran ángulo de apertura proyectadas sobre una superficie cilíndrica o esférica en comparación a una proyección lineal, obtienen una salida que nos permite visualizar de forma más sencilla una escena panorámica con gran ángulo de apertura.

Podemos decir que la proyección esférica produce una menor deformación global, cosa que hemos comprobado en la figura 10, debido a que el uso del cilindro ensancha mucho los polos norte y sur. Sin embargo, si hacemos uso de una distancia focal grande, tal diferencia no es tan característica.

Hemos encontrado numerosa dificultades que no esperábamos durante la realización de este trabajo. Uno de los problemas que más quebraderos de cabeza ha provocado ha sido trabajar sin tener una idea muy clara de qué modelo sigue la representación en número de coma flotante de de imágenes. Que librerías como `matplotlib` realicen normalizaciones automáticamente a las imágenes que le pides mostrar ha ocultado muchos problemas relacionados con operaciones con imágenes que se encontraban en distintas escalas. Encontrar la causa de estos problemas nos ha llevado mucho tiempo que hubiéramos preferido emplear en mejorar nuestros algoritmos y en encontrar ejemplos más impresionantes de nuestro trabajo.

Otra dificultad que hemos sufrido a lo largo de la implementación del trabajo es tener que mezclar el algoritmo de Burt-Adelson con la generación de panorámicas. Casi toda la literatura que se encuentra sobre el algoritmo de interpolación de Burt-Adelson se presenta en una situación donde la implementación resulta muy sencilla como en el ejemplo de la manzana y la naranja. Tener que adaptar el algoritmo a nuestro caso ha resultado no ser nada trivial y nos ha obligado a explorar muchos caminos que resultaban tras mucho trabajo no tener una salida. Creemos que la nuestra ha sido una labor en la que verdaderamente estábamos resolviendo creativamente un problema, muy lejos de limitarse a implementar un algoritmo que viniera detallado en un artículo.

## 4. Propuestas de Mejora de la Técnica y Trabajos Futuros

En este proyecto, se ha desarrollado un algoritmo de confección de panoramas lineales de imágenes proyectadas sobre superficies esférica/cilíndricas que produce transiciones suaves entre las imágenes adaptando el algoritmo de Burt Adelson. Los resultados obtenidos han sido buenos, tanto globalmente como analizando cada bloque de forma independiente. Sin embargo, el problema no se puede dar por concluido. Se nos ocurren numerosas ideas en las que podríamos seguir trabajando de haber tenido más tiempo:

1. Encontrar bajo que condiciones funciona bien la heurística de localización de la mayor región rectangular que se encuentra dentro de la intersección de dos imáge-

nes de la que hablamos en la sección de generación de la panorámica. Para ello, deberíamos estudiar si los algoritmos que se encuentran en la literatura sobre este tema podrían ser prácticos o no para este caso concreto. Un ejemplo de estos algoritmos puede ser consultado en [Smi].

2. La transformación geométrica de la imagen toma como origen de coordenadas la imagen central, por lo que el resultado final es una imagen correcta desde el punto de vista geométrico. Sin embargo, se desconoce cuál es el origen de coordenadas más adecuado para componer el mosaico, por lo que hemos hecho uso de una imagen como referencia absoluta a la hora de la creación del panorama. Ya que el punto de vista es algo subjetivo que depende de las imágenes para la composición del panorama. Más aun, algunas de los motivos que nos hicieron situar en primer lugar la imagen central en el mosaico no siguen teniendo sentido en un contexto en el que se van a aplicar proyecciones sobre una superficie cilíndrica o esférica.

3. Resultaría interesante introducir un modelo de distorsión en el problema geométrico, ya que nos permitiría aplicar en la técnica imágenes distorsionadas, generando un modelo distorsionado a partir de imágenes genéricas.

4. Podemos generalizar la noción de panorámica a sucesiones de imágenes en varios ejes. Un posible trabajo, sería en vez de haber proyectado las imágenes sobre una superficie cilíndrica o esférica, es la de crear una esfera de imágenes.

5. Nosotros usamos el algoritmo de Burt-Adelson para conseguir transiciones suaves entre las imágenes, sin embargo en la literatura encontramos distintas técnicas, como por ejemplo *la corrección de iluminación usando constancia de color*. Un posible trabajo futuro podría ser comparar el algoritmo implementado en en este documento, con otras técnicas de interpolación más actuales.

6. Recorte automático de los resultados: de haber tenido más tiempo podríamos haber implementado el recortado automático a las panorámicas (eliminar los bordes negros), pero al haber considerado otros temas del proyecto de mayor importancia se ha optado por realizar este corte de forma manual.

**Enlace de descarga de las imágenes:**

`https://consigna.ugr.es/f/HlNB26JIEETwj9HL/images.zip`

## Referencias


[BA83a] P. J. Burt y E. H. Adelson. «A multiresolution spline with applications to image mosaics». En: *ACM Transactions on Graphics* 2.4 (1983), págs. 217-236. DOI: http://persci.mit.edu/pub_pdfs/spline83.pdf.

[BA83b] P. J. Burt y E. H. Adelson. «The Laplacian Pyramid as a Compact Image Code». En: *IEEE Transactions on Communications* COM-3l.4 (1983). DOI: http://persci.mit.edu/pub_pdfs/pyramid83.pdf.

[Sze10] Richard Szeliski. *Computer Vision: Algorithms and Applications*. Springer, 2010.

[Bla] Nicolás Pérez de la Blanca. *Computer Vision: Panorama (07.panoramas.pdf)*.

[Lev] Marc Levoy. *Cylindrical Panoramas*. URL: http://graphics.stanford.edu/courses/cs178-12/applets/projection.html.

[Opea] OpenCV. *Feature Matching + Homography to find Objects*. URL: https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_feature_homography/py_feature_homography.html.

[Opeb] OpenCV. *Geometric Image Transformations*. URL: https://docs.opencv.org/2.4/modules/imgproc/doc/geometric_transformations.html.

[Opec] OpenCV. *Image Pyramids*. URL: https://docs.opencv.org/3.1.0/dc/dff/tutorial_py_pyramids.html.

[Oped] OpenCV. *Introduction to SIFT*. URL: https://docs.opencv.org/3.1.0/da/df5/tutorial_py_sift_intro.html.

[Smi] Daniel Smilkov. *Largest rectangle in a polygon*. URL: https://d3plus.org/blog/behind-the-scenes/2014/07/08/largest-rect/.

[Sti] Panorama Stitcher. *OpenPano: How to write a Panorama Stitcher*. URL: http://ppwwyyxx.com/2016/How-to-Write-a-Panorama-Stitcher/#Cylindrical-Panorama.

[Urt] Raquel Urtasun. *Computer Vision: Panorama*. URL: https://www.cs.toronto.edu/~urtasun/courses/CV/lecture08.pdf.