

Doble Grado en Ingeniería Informática y Matemáticas

VISIÓN POR COMPUTADOR
(E. Computación y Sistemas Inteligentes)

PROYECTO FINAL:

Implementar la creación de panoramas lineales con proyección en superficies cilíndricas o esféricas y mezcla de colores usando el algoritmo de Burt-Adelson.



**UNIVERSIDAD
DE GRANADA**

Alberto Estepa Fernández
Email: albertoestep@correo.ugr.es

Carlos Santiago Sánchez Muñoz
Email: carlossamu7@correo.ugr.es

8 de enero de 2020

Índice

1. Introducción	2
2. Proyección cilíndrica	3
3. Proyección esférica	7
4. Comparativa entre las proyecciones	10
5. Obtención de correspondencias	12
6. Cálculo de la homografía y construcción del mosaico	14
7. Algoritmo de Burt-Adelson	16
8. Burt-Adelson sobre el conjunto de imágenes “Yosemite”	20
9. Burt-Adelson sobre el conjunto de imágenes “Alhambra”	22
10. Conclusiones y trabajos futuros	26

1. Introducción

En el marco de la Visión por Computador encontramos la construcción de panoramas lineales. Esto consiste en la creación de imágenes panorámicas a partir de un conjunto de imágenes de un mismo lugar. Este tema tiene una profundidad y utilidad muy grande en diferentes ámbitos y nos abre un mundo dentro de la Visión por Computador.

La necesidad de combinar dos o más imágenes en un mosaico más grande ha surgido en diferentes contextos. Sin ir más lejos para fotos tomadas en el espacio, donde aparecen de estrellas y satélites que debido a sus órbitas sólo se pueden apreciar en diferentes puntos. Aquí la construcción de vistas panorámicas a partir de múltiples fotografías permite conseguir un campo de visión y nivel de detalle mucho mayor. Por ejemplo, a menudo fotografías de *Landsat* se ensamblan en vistas panorámicas de la Tierra.

Problema: nuestro problema a estudiar es el acople de las imágenes. Sabemos de la práctica 3 y lo estudiado en la teoría que este acople puede ser no satisfactorio ya que al tomar dos fotos de un mismo sitio percibimos diferencias de luces, sombras, brillos, contrastes y otros. A la hora de la creación de la vista panorámica esto se nota en la unión de las imágenes.

Solución propuesta: Vamos a usar el Algoritmo de Burt Adelson para la mezcla en la creación de la panorámica. Para ello vamos a hacer uso de dos proyecciones: proyección cilíndrica y proyección esférica.

Vamos a usar lo aprendido a lo largo del desarrollo de la asignatura pues necesitaremos filtros, construcciones de pirámides laplacianas, restauración de pirámides laplacianas, cálculo de correspondencias, cálculo de homografías más los nuevos contenidos ya mencionados.

La diferencia entre lo que se va a realizar ahora y los mosaicos que trabajamos en la práctica 3 es que al usar las proyecciones y al hacer la mezcla con el algoritmo de Burt Adelson conseguimos un suavizado en la unión de las imágenes y una construcción que parece prometedora.

El material donde encontramos detallada la explicación de Burt Adelson lo encontramos en el siguiente link [1]. Dicho material es apoyo fundamental de lo que aquí se realiza y explica.

2. Proyección cilíndrica

La proyección cilíndrica consiste en recorrer todos los píxeles de la imagen (y, x) y calcular su proyección cilíndrica (y', x') bajo la aplicación matemática que describe este fenómeno. Para ello necesitamos considerar el centro de la imagen (c_y, c_x) .

Para la transformación necesitamos dos parámetros:

- la distancia focal, f , que es la distancia desde el centro óptico
- y el factor de escalado, s , que define el escalado de la imagen para dibujarla en la original.

La fórmula para la proyección explicada es la siguiente:

$$x' = s * \arctan\left(\frac{x - c_x}{f}\right) + c_x \quad (1)$$

$$y' = s * \frac{y - c_y}{\sqrt{(x + c_x)^2 + f^2}} + c_y \quad (2)$$

El procedimiento es crear una nueva imagen en donde en el píxel (y', x') guardamos el valor que hay en el píxel (y, x) de la imagen original. Este valor puede ser una terna RGB en caso de que sea una imagen tribanda. Esto es fácil de evitar pues hemos programado esta función de manera recursiva y una vez que tenemos la proyección implementada para un canal es fácil extenderlo a tres usando las funciones `cv2.split()` y `cv2.merge()`.

Respecto a los parámetros distancia focal y factor de escala, la imagen tendrá mayor proyección cuanto más pequeños sean estos parámetros. Recíprocamente para valores grandes será más similar a la original pues tendrá menos proyección.

Vamos a trasladar todo lo explicado a código. Programamos en Python una función `cylindricalProjection()` a la cual le pasamos la imagen y los parámetros y nos devuelve la proyección cilíndrica de esa imagen. Veamos:

```
""" Proyección cilíndrica de una imagen
- img: imagen a proyectar.
- f: distancia focal.
- s: factor de escala.

def cylindricalProjection(img, f, s):
    if len(img.shape) == 3:
        # Si está en color separamos en los distintos canales
        canals = cv2.split(img)
        canals_proy = []
        for n in range(len(canals)):
            # Proyección cilíndrica
            canals_proy.append(cylindricalProjection(canals[n], f, s))
        # Mezclamos canales
        projected=cv2.merge(canals_proy)

    else:
```

```

projected = np.zeros(img.shape) # Imagen proyectada
y_center = img.shape[0]/2      # centro y
x_center = img.shape[1]/2      # centro x
# Proyectamos la imagen
for i in range(img.shape[0]):
    for j in range(img.shape[1]):
        y_proy = floor(s * ((i-y_center) / np.sqrt((j-x_center)*(j-x_center) + f*f)))
        + y_center)
        x_proy = floor(s * np.arctan((j-x_center) / f) + x_center)
        projected[y_proy][x_proy] = img[i][j]
# Normalizamos
projected = cv2.normalize(projected, projected, 0, 255, cv2.NORM_MINMAX, cv2.CV_8U)

return projected

```

Ahora vamos a probarlo sobre diferentes imágenes. La primera imagen que vamos a proyectar es del palacio de Carlos V:



Imagen 1: Palacio de Carlos V

La primera proyección la hacemos con valores distancia focal y factor de escalado de 600. El resultado queda así:



Imagen 2: Palacio de Carlos V proyectado cilíndricamente con $f = 600$ y $s = 600$

La segunda proyección la hacemos con valores distancia focal y factor de escalado de 900. El resultado queda así:

Proyección cilíndrica. $f=900$. $s=900$.



Imagen 3: Palacio de Carlos V proyectado cilíndricamente con $f = 900$ y $s = 900$

Haciendo una valoración de los resultados lo primero que debemos de darnos cuenta es lo que mencionamos anteriormente de que cuanto mayores sean los parámetros f y s más se parece a la imagen original. La *Imagen 3* se parece a la original más que la otra, la cual está más proyectada.

Todas las imágenes tienen las mismas dimensiones, ello nos permite observar que fruto de la proyección hay una zona de la imagen (las esquinas) que se queda en negro. La explicación es sencilla, nosotros antes de proyectar rellenamos la matriz con ceros y no existen píxeles (y, x) que viajen a esos de las esquinas por medio de la proyección.

La segunda imagen que vamos a proyectar es una foto de la Alhambra vista desde el Albaicín granadino que será parte de la futura vista panorámica:



Imagen 4: Alhambra

La primera proyección la hacemos con valores distancia focal y factor de escalado de 600. La segunda proyección la hacemos con valores distancia focal y factor de escalado de 800. El resultado queda así:

Proyeccion cilindrica. $f=600$. $s=600$.



Imagen 5: Alhambra proyectada cilíndricamente con $f = 600$ y $s = 600$

Proyeccion cilindrica. $f=800$. $s=800$.

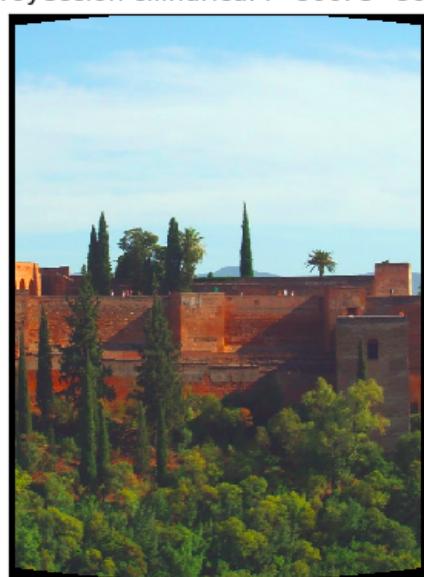


Imagen 6: Alhambra proyectada cilíndricamente con $f = 800$ y $s = 800$

Al igual que antes la *Imagen 6* es la que más parecida a la original y la que tiene mayor distancia focal y factor de escalado.

La que está más proyectada, la *Imagen 5*, es la que deja más esquina en color negro.

3. Proyección esférica

Para la proyección esférica también vamos a recorrer todos los píxeles de la imagen (y, x) calculando su proyección cilíndrica (y', x') bajo la aplicación matemática que describe dicho fenómeno. Para ello necesitamos considerar el centro de la imagen (c_y, c_x) .

Para la transformación necesitamos dos parámetros:

- la distancia focal, f , que es la distancia desde el centro óptico
- y el factor de escalado, s , que define el escalado de la imagen para dibujarla en la original.

La fórmula para la proyección explicada es la siguiente:

$$x' = s * \arctan\left(\frac{x - c_x}{f}\right) + c_x \quad (3)$$

$$y' = s * \arctan\left(\frac{y - c_y}{\sqrt{(x - c_x)^2 + f^2}}\right) + c_y \quad (4)$$

Al igual que antes el procedimiento es crear una nueva imagen en donde en el píxel (y', x') guardamos el valor que hay en el píxel (y, x) de la imagen original. Este valor puede ser una terna RGB en caso de que sea una imagen tribanda y lo evitamos usando recursividad ya que una vez que tenemos la proyección implementada para un canal es fácil extenderlo a tres usando las funciones `cv2.split()` y `cv2.merge()`.

Tenemos otra vez la conclusión de que la imagen tendrá mayor proyección cuanto más pequeños sean los parámetros distancia focal y factor de escala la imagen tiene mayor proyección y recíprocamente para valores grandes será más similar a la original.

Vamos a trasladar todo lo explicado a código. Programamos en Python una función `sphericalProjection()` a la cual le pasamos la imagen y los parámetros y nos devuelve la proyección cilíndrica de esa imagen. Veamos:

```
""" Proyección esférica de una imagen
- img: imagen a proyectar.
- f: distancia focal.
- s: factor de escalado.
"""
def sphericalProjection(img, f, s):
    if len(img.shape) == 3:
        # Si está en color sepáramos en los distintos canales
        canals = cv2.split(img)
        canals_proy = []
        for n in range(len(canals)):
            # Proyección esférica
            canals_proy.append(sphericalProjection(canals[n], f, s))
        # Mezclamos canales
        projected=cv2.merge(canals_proy)
```

```

else:
    projected = np.zeros(img.shape) # Imagen proyectada
    y_center = img.shape[0]/2      # centro y
    x_center = img.shape[1]/2      # centro x
    # Proyectamos la imagen
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            y_proy = floor(s * np.arctan((i-y_center) / np.sqrt((j-x_center)*(j-x_center)+f*f))) -
            x_proy = floor(s * np.arctan((j-x_center) / f) + x_center)
            projected[y_proy][x_proy] = img[i][j]
    # Normalizamos al tipo uint8
    projected = cv2.normalize(projected, projected, 0, 255, cv2.NORM_MINMAX, cv2.CV_8U)

return projected

```

Ahora vamos a probarlo sobre diferentes imágenes. La primera como ya sabemos es del palacio de Carlos V. Realizamos una primera proyección con valores distancia focal y factor de escalado de 600. El resultado queda así:



Imagen 7: Palacio de Carlos V proyectado esféricamente con $f = 600$ y $s = 600$

Ahora hacemos otra proyección con valores distancia focal y factor de escalado de 900. El resultado queda así:



Imagen 8: Palacio de Carlos V proyectado esféricamente con $f = 900$ y $s = 900$

Es momento de valorar los resultados de esta proyección. Confirmamos lo que ya mencionamos anteriormente de que cuanto mayores sean los parámetros f y s más se parece a la imagen original. La *Imagen 8* es más parecida a la original y la *Imagen 7* tiene una mayor proyección. De nuevo se nos quedan zonas de la imagen (las esquinas) en negro.

La segunda imagen que vamos a proyectar es la foto de la Alhambra. Para la primera proyección usamos parámetros distancia focal y factor de escalado de 600. La segunda proyección la hacemos con valores distancia focal y factor de escalado de 800. Mostramos el resultado:

Proyección esférica. $f=600$. $s=600$.



Imagen 9: Alhambra proyectada esféricamente
con $f = 600$ y $s = 600$

Proyección esférica. $f=800$. $s=800$.

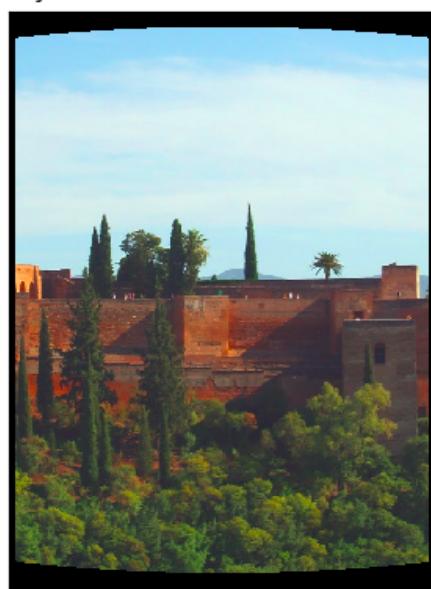


Imagen 10: Alhambra proyectada
esféricamente con $f = 800$ y $s = 800$

Al igual que antes la *Imagen 10* es la que más parecida a la original y la que tiene mayor distancia focal y factor de escalado. La *Imagen 9* está más proyectada y deja más esquina en color negro.

4. Comparativa entre las proyecciones

Si realizamos una comparativa entre las dos proyecciones, la diferencia está en que para la esférica se toma $\arctan()$ en ambas direcciones, filas y columnas en nuestro caso. Veamos:

$$\text{Proyección cilíndrica} \rightarrow y' = s * \frac{y - c_y}{\sqrt{(x + c_x)^2 + f^2}} + c_y$$

$$\text{Proyección esférica} \rightarrow y' = s * \arctan \left(\frac{y - c_y}{\sqrt{(x + c_x)^2 + f^2}} \right) + c_y$$

Observemos qué diferencia hay en los resultados debido a este fenómeno:

Imagen 5: Proyección cilíndrica. $f=600$. $s=600$.



Imagen 9: Proyección esférica. $f=600$. $s=600$.

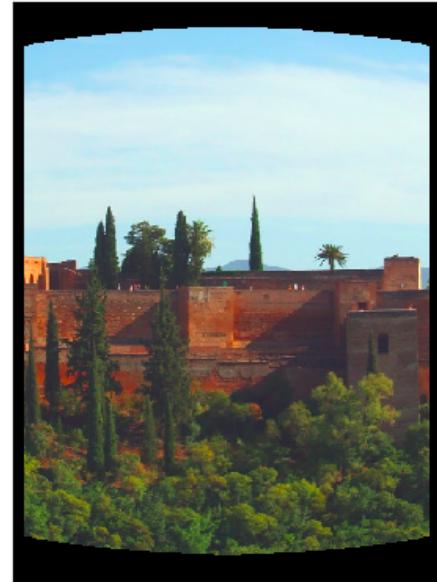


Imagen 5: Alhambra proyectada cilíndricamente con $f = 600$ y $s = 600$

Imagen 9: Alhambra proyectada esféricamente con $f = 600$ y $s = 600$

He escogido las imágenes con parámetros $f = 600$ y $s = 600$ que eran las que estaban más proyectadas para tener mayores diferencias. Efectivamente obtenemos lo que comentábamos teóricamente, la componente y' cambia entre la esférica y la cilíndrica y por eso ahora la imagen es menos alta, tiene más forma de “esfera”.

Probamos también a juntar las dos imágenes más proyectadas del otro ejemplo y ver qué resultado obtenemos:

Proyección cilíndrica. $f=600$. $s=600$.



Imagen 2: Palacio de Carlos V proyectado cilíndricamente con $f = 600$ y $s = 600$

Proyección esférica. $f=600$. $s=600$.



Imagen 7: Palacio de Carlos V proyectado esféricamente con $f = 600$ y $s = 600$

En este ejemplo es más difícil observar el cambios pues las dos imágenes son realmente parecidas. Sin embargo, hay cambios en el eje vertical ya que en la *Imagen 2* los píxeles en negro acaban en la letra “o” de la palabra “Proyección” de la parte superior y en la *Imagen 7* esto ocurre en la letra “n”.

5. Obtención de correspondencias

Para la obtención de las correspondencias vamos a usar el detector AKAZE de OpenCV. A partir de él obtendremos una lista de `keyPoints` y un descriptor por cada una de las dos imágenes.

Haremos uso de dos criterios de características aprendidos en el desarrollo de la asignatura. Los dos criterios de correspondencias son *BruteForce + crossCheck* y *Lowe-Average-2NN*.

Usamos la clase BFMatcher y al llamarla podemos activar el flag *crossCheck* que hace que la validación sea cruzada, es decir, que cada correspondencia obtenida cumple que en ambos descriptores es la mejor correspondencia del otro.

No voy a proporcionar más explicaciones de esto, pues corresponde a lo que ya se comentó en la Práctica 3 y ahora tenemos un cometido más grande. Código de los dos criterios:

```
"""Dadas dos imágenes calcula los keypoints y descriptores para obtener los matches
usando "BruteForce+crossCheck". Devuelve la imagen compuesta.
- img1: Primera imagen para el match.
- img2: Segunda imagen para el match.
- n (op): número de matches a mostrar. Por defecto 50.
- flag (op): indica si se muestran keypoints y matches (0) o solo matches (2).
    Por defecto 2.
- flagReturn (op): indica si debemos devolver los keypoints y matches o la imagen.
    Por defecto devolvemos la imagen.
"""
def getMatches_BFCC(img1, img2, n = 50, flag = 2, flagReturn = 1):
    # Inicializamos el descriptor AKAZE
    detector = cv2.AKAZE_create()
    # Se obtienen los keypoints y los descriptores de las dos imágenes
    keypoints1, descriptor1 = detector.detectAndCompute(img1, None)
    keypoints2, descriptor2 = detector.detectAndCompute(img2, None)

    # Se crea el objeto BFMatcher activando la validación cruzada
    bf = cv2.BFMatcher(normType=cv2.NORM_L2, crossCheck = True)
    # Se consiguen los puntos con los que hace match
    matches1to2 = bf.match(descriptor1, descriptor2)

    if len(matches1to2)<=n:
        n = len(matches1to2)
    # Se guardan n puntos aleatorios
    matches1to2 = sample(matches1to2, n)

    # Imagen con los matches
    img_match = cv2.drawMatches(img1, keypoints1, img2, keypoints2, matches1to2, outImg=None)

    # El usuario nos indica si quiere los keypoints y matches o la imagen
    if flagReturn:
        return img_match
    else:
        return keypoints1, keypoints2, matches1to2

""" Dadas dos imágenes calcula los keypoints y descriptores para obtener los matches
usando "Lowe-Average-2NN". Devuelve la imagen compuesta.
Si se indica el flag "improve" como True, a elegir los mejores matches.
```

```

- img1: Primera imagen para el match.
- img2: Segunda imagen para el match.
- n (op): número de matches a mostrar. Por defecto 50.
- ratio (op): Radio para la distancia entre puntos. Por defecto 0.8.
- flag (op): indica si se muestran keypoints y matches (0) o solo matches (2).
    Por defecto 2.
- flagReturn (op): indica si debemos devolver los keypoints y matches o la imagen.
    Por defecto devolvemos la imagen.

"""

def getMatches_LA2NN(img1, img2, n = 50, ratio = 0.8, flag = 2, flagReturn = 1):
    # Inicializamos el descriptor AKAZE
    detector = cv2.AKAZE_create()
    # Se obtienen los keypoints y los descriptores de las dos imágenes
    keypoints1, descriptor1 = detector.detectAndCompute(img1, None)
    keypoints2, descriptor2 = detector.detectAndCompute(img2, None)

    # Se crea el objeto BFMatcher
    bf = cv2.BFMatcher(normType=cv2.NORM_L2, crossCheck=False)
    # Escogemos los puntos con los que hace match indicando los vecinos más cercanos para la comprobación (2)
    matches1to2 = bf.knnMatch(descriptor1, descriptor2, k=2)

    # Mejora de los matches ->los puntos que cumplan con un radio en concreto
    best1to2 = []
    # Se recorren todos los matches
    for p1, p2 in matches1to2:
        if p1.distance < ratio * p2.distance:
            best1to2.append([p1])

    if len(best1to2)<=n:
        n=len(best1to2)
    # Se guardan n puntos aleatorios
    matches1to2 = sample(best1to2, n)

    # Imagen con los matches
    img_match = cv2.drawMatchesKnn(img1, keypoints1, img2, keypoints2, best1to2, outImg=None)

    # El usuario nos indica si quiere los keypoints y matches o la imagen
    if flagReturn:
        return img_match
    else:
        return keypoints1, keypoints2, best1to2

```

6. Cálculo de la homografía y construcción del mosaico

A continuación se muestra el código de la función `getHomography()` a la cual se le pasa dos imágenes y una bandera. La bandera indica qué criterio de correspondencias de los ya expuestos usar, por defecto será *Lowe-Average-2NN*. Se calculan los puntos origen y destino para posteriormente obtener la homografía entre las imágenes usando `findHomography()` de OpenCV.

```
""" Calcula la homografía entre dos imágenes.
- img1: primera imagen.
- img2: segunda imagen.
- flag (op): si vale 1 se calcular con Lowe-Average-2NN y si vale 0
    con BruteForce+crossCheck. Por defecto vale 1.
"""
def getHomography(img1, img2, flag=1):
    # Obtenemos los keyPoints y matches entre las dos imágenes.
    if(flag):
        kpts1, kpts2, matches = getMatches_LA2NN(img1, img2, flagReturn=0)
        # Ordeno los puntos para usar findHomography
        puntos_origen = np.float32([kpts1[punto[0].queryIdx].pt
                                    for punto in matches]).reshape(-1, 1, 2)
        puntos_destino = np.float32([kpts2[punto[0].trainIdx].pt
                                    for punto in matches]).reshape(-1, 1, 2)
    else:
        kpts1, kpts2, matches = getMatches_BFCC(img1, img2, flagReturn=0)
        # Ordeno los puntos para usar findHomography
        puntos_origen = np.float32([kpts1[punto.queryIdx].pt
                                    for punto in matches]).reshape(-1, 1, 2)
        puntos_destino = np.float32([kpts2[punto.trainIdx].pt
                                    for punto in matches]).reshape(-1, 1, 2)
    # Llamamos a findHomography
    homografia, _ = cv2.findHomography(puntos_origen, puntos_destino, cv2.RANSAC, 1)
    return homografia
```

Para la construcción del mosaico, que en nuestro caso es la futura vista panorámica, he implementado una función llamada `getMosaic()`. Dicha función recibe como parámetro dos imágenes y devuelve el mosaico de ellas. Toma como tamaño para el mosaico la altura de la primera imagen y como ancho la suma de los anchos.

Hay que tener un detalle importante. No devuelve el mosaico como una imagen resultado (lo cual sería parecido a lo que hacíamos en la práctica 3) sino que devuelve dos imágenes del mismo tamaño en el que en una está colocada la primera imagen en su sitio y en la otra exclusivamente la segunda imagen en su sitio. Por tanto es obvio que al llamar a `warpPerspective()` para la segunda imagen pasamos la que nos dieron como parámetro y nunca sobrescribimos el resultado anterior. El resto de las imágenes resultantes está en negro. Esto es importante para poder aplicar Burt-Adelson más adelante.

```
""" Calcula el mosaico resultante de N imágenes.
- list: Lista de imágenes.
"""
def getMosaic(img1, img2):
    width = img1.shape[1] + img2.shape[1]    # Ancho del mosaico
    height = img1.shape[0]                    # Alto del mosaico
```

```

# Homografía 1
hom1 = np.matrix([[1,0,0],[0,1,0],[0,0,1]], dtype=float)
res1 = cv2.warpPerspective(img1, hom1, (width, height))
# Homografía 2
hom2 = getHomography(img2, img1, 1)
hom2 = hom1 * hom2
res2 = cv2.warpPerspective(img2, hom1*hom2, (width, height))

return res1, res2

```

Para entender lo que estoy diciendo he parado la ejecución del algoritmo en ese punto e imprimo los resultados, los voy a mostrar para el ejemplo de la “Alhambra”.

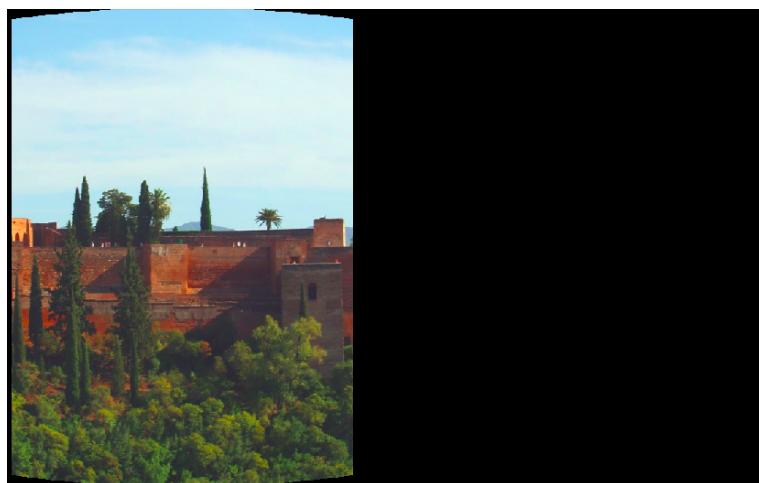


Imagen 11: Parte 1 del mosaico de la Alhambra



Imagen 12: Parte 2 del mosaico de la Alhambra

Obtenemos lo esperado, dos imágenes con el tamaño explicado en donde en cada una aparece una de las imágenes en su sitio. El *match* es obvio a simple vista.

7. Algoritmo de Burt-Adelson

Antes de la llegada de este algoritmo construimos nuestro mosaico y podíamos observar los cortes entre las uniones de fotos, ahora vamos a usar el algoritmo de Burt-Adelson para realizar un suavizado en todos los píxeles. Para ello vamos a construir pirámides laplacianas de las imágenes, realizar una mezcla en todos los niveles y reconstruir la imagen.

La explicación del algoritmo la hemos dividido en 5 grandes pasos que nos llevarán al resultado que deseamos. La función centralizadora de todo lo que se va a explicar es `BurtAdelson()` y llamará a cada función de los pasos explicados pues hemos intentado tener un equilibrio en la modularización del código.

Paso 1: Obtención del mosaico

Este paso es la función `getMosaic()` que ya hemos explicado detalladamente. A partir de aquí tendremos dos imágenes que debemos mezclar adecuadamente más adelante.

Paso 2: Limpiar zona sobrante

Es importante para nuestro algoritmo borrar las filas y columnas que estén enteras negras en ambas fotos. En la función `cleanImage()` definimos una matriz que usaremos como máscara llena de ceros. Los píxeles de la primera imagen los guardamos con el valor 1 y los de la segunda con el valor 2. Finalmente borramos en ambas imágenes aquellas filas y columnas que en la máscara sigan siendo cero. Código:

```
""" Limpia el área que no está en ninguna imagen.
- img1: primera imagen a tratar.
- img2: segunda imagen a tratar.
"""

def cleanImage(img1, img2):
    mask = np.zeros((img1.shape[0], img1.shape[1]))
    mask[np.nonzero(img1)[0:2]] = 1
    mask[np.nonzero(img2)[0:2]] = 2

    # Si la imagen está a color creamos una copia en B/N
    if len(mask.shape) == 3:
        copia_mask = cv2.cvtColor(mask, cv2.COLOR_BGR2GRAY)
    else:
        copia_mask = mask

    col = np.any(copia_mask.T != 0, axis = 1) # columnas de 0s
    raw = np.any(copia_mask.T != 0, axis = 0) # filas de 0s

    # Borramos filas y columnas sobrantes
    mask = mask[:, col][raw, :]
    img1 = img1[:, col][raw, :]
    img2 = img2[:, col][raw, :]

    return img1, img2
```

También tenemos la función `adjustImages()` que recorta las imágenes al tamaño de la más pequeña.

Paso 3: Construcción de las pirámides laplacianas

La función `laplacianPyramid()` construye una pirámide laplaciana con número de niveles n . Para ello tiene que construir una pirámide gaussiana de $n + 1$ niveles usando la función `gaussianPyramid()`.

El detalle que aquí debemos de tener es guardar el último nivel de la pirámide gaussiana en la laplaciana pues próximamente necesitaremos hacer una recuperación de la pirámide y ésto es necesario en el algoritmo de recuperación.

Para no hacer más pesada la lectura de este proyecto no muestro el código trivial de esta función.

Paso 4: Mezcla de las pirámides

En este momento poseemos dos pirámides laplacianas de dos imágenes recortadas de manera adecuada, con el mismo tamaño y que al superponerlas correctamente forman el mosaico.

Ahora es momento de la mezcla de estas dos pirámides. Consideramos la mitad de la imagen y para cada nivel de la pirámide creamos una imagen, en este caso llamada `aux`, en la que en la primera mitad guardamos la imagen de la primera mitad de la primera imagen para ese nivel de la pirámide laplaciana. De igual manera hacemos para la segunda mitad. Vamos guardando esta imagen en una pirámide, `finalLaplacian`. Mostramos código:

```
""" Para cada nivel de la Laplaciana mezcla la primera mitad de la imagen
de la primera ápirmide con la segunda de la segunda imagen.
- laplaciana1: primera ápirmide a mezclar.
- laplaciana1: segunda ápirmide a mezclar.
"""
def mixLaplacians(laplaciana1 , laplaciana1):
    finalLaplacian = []
    for i in range(len(laplaciana1)):
        aux = np.zeros(laplaciana1[i].shape , laplaciana1[i].dtype)
        mitad = laplaciana1[i].shape[1] // 2
        aux[:, :mitad, ...] = laplaciana1[i][:, :mitad, ...]
        aux[:, -mitad:, ...] = laplaciana1[i][:, -mitad:, ...]
        if laplaciana1[i].shape[1] % 2 == 1:
            # Numero de columnas impar ->media aritmética
            aux[:, mitad, ...] = (laplaciana1[i][:, mitad, ...] +
                                  laplaciana1[i][:, mitad, ...])/2
        finalLaplacian.append(aux)
    return finalLaplacian
```

Paso 5: Restauramos la pirámide

Después de hacer la mezcla poseemos una única pirámide laplaciana. Para continuar con nuestro algoritmo debemos hacer la recuperación de dicha pirámide laplaciana para obtener una sola imagen que será la vista panorámica que tanto deseamos.

Vamos a proceder a recordar el estudio teórico de como recuperar la imagen original a través de la laplaciana. Sea $[L_0, \dots, L_{n-1}]$ la pirámide laplaciana de n niveles. Sabemos que en su construcción se ha usado una pirámide gaussiana de $n+1$ niveles $[G_0, \dots, G_n]$ donde G_0 es la imagen original.

Nuestro objetivo es seguir los pasos de la construcción de la pirámide laplaciana pero en este caso al revés y para poder llevar a cabo este algoritmo necesitaremos la pirámide laplaciana, G_n y F , la función que expande tamaño del de la imagen de nivel superior a la de nivel inferior.

En la construcción usamos que $L_k := G_k - F(G_{k+1})$, por lo que despejando $G_k = L_k + F(G_{k+1}) \forall k \in [0, n-1]$. De esta manera obtendríamos G_0 , primer nivel de la pirámide gaussiana que coincide con la imagen original. Veamos código:

```
""" Restaura una ápiramide laplaciana .
- piramide: ÁPirmide a restaurar .
"""

def lapacianRestoring(piramide):
    # Cogemos el ultimo nivel de la laplaciana
    recuperacion = piramide[-1]
    # Recorremos todos los niveles
    for i in range(len(piramide) - 1):
        # Cogemos el siguiente
        siguiente = piramide[-2 - i]
        # Transformamos al formato adecuado para el upsample
        recuperacion = np.float32(recuperacion)
        # Realizamos la convolucion y aumento
        aumento = convolution(recuperacion, siguiente)
        # Recuperamos el formato
        recuperacion = np.uint32(recuperacion)
        # Realizamos la suma y recuperamos la imagen
        recuperacion = aumento + siguiente
    # Guardamos la imagen en el formato uint32
    recuperacion = np.uint32(recuperacion)
    return recuperacion
```

Burt-Adelson

Llegados a este punto procedemos a realizar el seguimiento de los 5 pasos anteriores y, si no nos hemos equivocado, obtener lo buscado. La función `BurtAdelson()` realiza este trabajo.

En primer lugar recibe tres parámetros: las imágenes a ensamblar y el número de niveles para la pirámide laplaciana, lo cual iremos variando para ver de qué manera influye en el resultado final. Realiza lo mencionado y devuelve el resultado.

```
""" Aplica el algoritmo Burt Adelson a dos áimenes
- img1: primera imagen a tratar .
- img2: segunda imagen a tratar .
- levels (op): niveles de la ápiramide laplaciana que se usa en el algoritmo . Por defecto 6 .
"""

def BurtAdelson(img1, img2, levels=6):
    res1, res2 = getMosaic(img1, img2)                                # Calulamos el mosaico
    res1, res2 = cleanImage(res1, res2)                                # Limpiamos las imágenes
```

```

res1, res2 = adjustImages(res1, res2)
lap1 = laplacianPyramid(res1, levels)
lap2 = laplacianPyramid(res2, levels)
lap_splined = mixLaplacians(lap1, lap2)
img_splined = laplacianRestoring(lap_splined)
np.clip(img_splined, 0, 255, out=img_splined)
img_splined = np.uint8(img_splined)
return img_splined

```

*# Ajustamos imágenes
Pirámide laplaciana 1
Pirámide laplaciana 1
Pirámide laplaciana combinada
Restauramos la laplaciana combinada
Normalizamos al rango [0,255]
Formato uint8 para visualización*

Extensión a N imágenes

El algoritmo anterior recibe como parámetro únicamente dos imágenes (más el número de niveles que es opcional). Ahora queremos un algoritmo que se le pase una lista de imágenes, `img_list` y realiza la construcción Burt-Adelson.

La función empieza por la imagen central de la lista y va creando dos subimágenes resultado, que son `left` y `right`. La primera va hacia desde la imagen central hacia la izquierda y la segunda hacia la derecha. Finalmente, mezclamos con `BurtAdelson(left, right)` las dos imágenes y devolvemos la vista panorámica final.

```

""" Aplica el algoritmo Burt Adelson a N imágenes
- img_list: lista de imágenes a tratar.
- levels (op): niveles de la pirámide laplaciana que se usa en el algoritmo. Por defecto 6.
- title (op): título del conjunto de imágenes.
"""

def BurtAdelson_N(img_list, levels=6, title="Álbumes"):
    print("BurtAdelson al conjunto de imágenes " + title)
    centro = len(img_list)//2
    right = BurtAdelson(img_list[centro], img_list[centro+1], levels)
    left = BurtAdelson(img_list[centro-1], img_list[centro], levels)

    for n in range(centro, -1, -1):
        left = BurtAdelson(left, img_list[n], levels)
    for n in range(centro, len(img_list)):
        right = BurtAdelson(right, img_list[n], levels)

    mosaic = BurtAdelson(left, right, levels)

    return mosaic

```

Finalizamos aquí la sección acerca todas las explicaciones relativas al algoritmo central de este proyecto. A falta de los resultados tenemos la confianza de haber seguido bien el camino y conseguir unas vistas panorámicas correctas y con un buen nivel de detalle.

8. Burt-Adelson sobre el conjunto de imágenes “Yosemite”

Como primer conjunto de imágenes nos hemos permitido usar las de “Yosemite” proporcionadas en el material de la asignatura. Vamos a empezar haciendo construcciones con proyección cilíndrica: primero una con 6 niveles de la pirámide laplaciana y luego con 9.

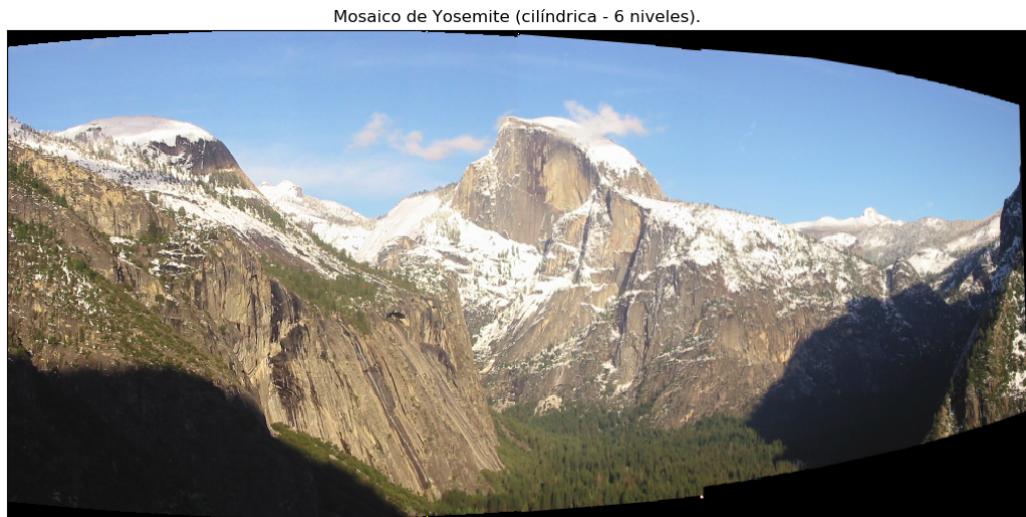


Imagen 13: Yosemite con proyección cilíndrica y 6 niveles de la pirámide



Imagen 14: Yosemite con proyección cilíndrica y 9 niveles de la pirámide

Ambas imágenes muestran que el proceso de ensamblado ha tenido éxito. Las diferencias por el cambio de nivel son nimias, apreciamos más atenuación de luces en la de 9 niveles y algo más de calidad de imagen en la de 6.

Hacemos el mismo enamblado que antes pero usando la proyección esférica. Veamos las imágenes obtenidas:

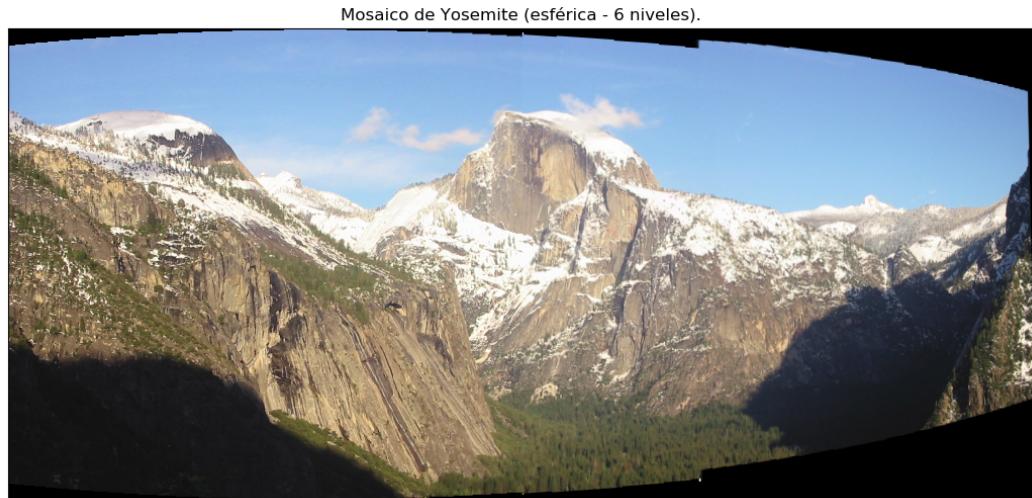


Imagen 15: Yosemite con proyección esférica y 6 niveles de la pirámide



Imagen 16: Yosemite con proyección esférica y 9 niveles de la pirámide

También es un resultado excelente y los cambios de una imagen a otra apenas se notan, han sido muy suavizados por la restauración de la pirámide laplaciana. No quiero dar más explicaciones para reservarme y confirmar juicios con el siguiente conjunto de datos propuesto por nosotros que vamos a usar, el de la “Alhambra”.

9. Burt-Adelson sobre el conjunto de imágenes “Alhambra”

Para este prueba disponemos de tres imágenes de la “Àlhambra” vista desde el Albaicín.



Imagen 17: Trozo 1



Imagen 18: Trozo 2



Imagen 19: Trozo 3

Con dichas imágenes vamos a construir la vista panorámica. En primer lugar usamos proyección cilíndrica con número de niveles para la pirámide 6 y 9.

Mosaico de la Alhambra (cilíndrica - 6 niveles).



Imagen 20: Alhambra con proyección cilíndrica y 6 niveles de la pirámide

El resultado es verdaderamente bueno. Notamos los efectos de la proyección cilíndrica para la cual se han usado parámetros distancia focal y factor de escalado de 900. El ensamblado es muy bueno porque no se notan los cortes. Exponemos el resultado con 9 niveles y seguimos haciendo más valoraciones.

Mosaico de la Alhambra (cilíndrica - 9 niveles).



Imagen 21: Alhambra con proyección cilíndrica y 9 niveles de la pirámide

Diferencias bastante pequeñas respecto de la anterior, suavizado en las nubes del cielo y poco más. Valorando las dos juntas hemos de decir que lo normal después de haber ejecutado el algoritmo es que no se noten los cambios de una imagen a otra (entre esas 3 que forman la panorámica) pues para eso hacemos las pirámides, mezclamos y restauramos. Sin embargo en este caso, hasta el empalme de una imagen con la de al lado es excelente y la parte inferior y superior en negro casan casi en el mismo sitio dificultando saber donde está la unión entre las imágenes.

Procedemos a hacer lo análogo a lo anterior pero usando la proyección esférica. De nuevo los dos niveles para las pirámides son 6 y 9.

Mosaico de la Alhambra (esférica - 6 niveles).



Imagen 22: Alhambra con proyección esférica y 6 niveles de la pirámide

Buen resultado y buen ensamblado. Destacar que la zona derecha de la imagen se ve realmente proyectada con una diferencia de tamaño con respecto a la parte izquierda considerable. Realmente da la sensación de estar en una “esfera”. En la proyección cilíndrica también estaba la zona derecha bastante proyectada pero no a este nivel, la cilíndrica está más cercana a la realidad, a si hubiéramos tomado una imagen horizontal en ese instante.

La última imagen es la de la proyección esférica con 9 niveles.

Mosaico de la Alhambra (esférica - 9 niveles).



Imagen 23: Alhambra con proyección esférica y 9 niveles de la pirámide

El ensamblado es muy bueno como ya se ha comentado. Me atrevería a decir que si en alguna de las 4 imágenes anteriores hacemos un recortado adecuado eliminando los fondos negros no podríamos distinguir donde están las uniones. Podemos hacer la prueba usando la *Imagen 20*. Veamos:



Imagen 24: Alhambra recortada

10. Conclusiones y trabajos futuros

El algoritmo programado ofrece **resultados satisfactorios** cumpliendo los requisitos que teníamos. A partir de un conjunto de imágenes nos proporciona una vista panorámica con cualquiera de las proyecciones estudiadas que tiene un nivel de detalle alto.

Estos buenos resultados tienen un **coste computacional** asumible, más o menos podemos hablar de 15 segundos por ejemplo ya que hay que contar el cálculo de la proyección, pirámides laplacianas con un número de niveles alto, mezcla y restauración así que no es un tiempo disparatado. Este coste es asumible y podemos construir todas las vistas panorámicas que queramos.

La programación es genérica y robusta por lo que lo implementado es **fácilmente accesible** por otros usuarios que necesiten de estas vistas panorámicas simplemente llamando a la función `BurtAdelson_N`. Además la terminal informa en todo momento de qué se hace. Así queda después de la ejecución de todo el programa y ejemplos:

```
carlos@carlos-Aspire-VN7-571G:~/Documentos/VC_Proyecto$ python3.6 BurtAdelson.py
----- PREPROCESANDO IMÁGENES -----
Leyendo 'imagenes/carlosV.jpg' en color
Leyendo 'imagenes/yosemite1.jpg' en color
Leyendo 'imagenes/yosemite2.jpg' en color
Leyendo 'imagenes/yosemite3.jpg' en color
Leyendo 'imagenes/al1.png' en color
Leyendo 'imagenes/al2.png' en color
Leyendo 'imagenes/al3.png' en color
Calculando las proyecciones cilíndricas de 'Yosemite'
Calculando las proyecciones cilíndricas de 'Alhambra 1'
Calculando las proyecciones esféricas de 'Yosemite'
Calculando las proyecciones esféricas de 'Alhambra 1'

----- PROBANDO PROYECCIONES -----
Proyecciones cilíndricas
Proyecciones esféricas
Pulsa 'Enter' para continuar

----- PROBANDO BURTADELSON -----
El número de NIVELES de la pirámide para Burt-Adelson es: 6
BurtAdelson al conjunto de imágenes 'Yosemite (cilíndrica - 6 niveles)'
BurtAdelson al conjunto de imágenes 'Yosemite (esférica - 6 niveles)'
BurtAdelson al conjunto de imágenes 'Alhambra (cilíndrica - 6 niveles)'
BurtAdelson al conjunto de imágenes 'Alhambra (esférica - 6 niveles)'
Pulsa 'Enter' para continuar
```

Imagen 25: Terminal

Para este algoritmo es crucial la elección de parámetros pues un número bajo de niveles de la pirámide podría dejar deformaciones y un número alto suavizaría mucho y habría pérdida de información. También hay que tener en cuenta la máscara de la convolución.

Al hilo de lo anterior también destacar la elección de la proyección y la dificultad

al escoger los parámetros distancia focal y factor de escalado pues esto varía muchísimo los resultados y nos puede quedar una imagen muy proyectada.

Por tanto, podemos considerar como una desventaja la **difícil elección de diferentes parámetros**. Otra es que el algoritmo empeora sus resultados si el número de imágenes a ensamblar es grande.

Para finalizar este proyecto vamos a exponer algunas **posibles mejoras** que a nuestro entender se podrían hacer para seguir profundizando en lo tratado:

- Establecer diferente vista panorámica en función del punto de vista del observador, así una imagen podrá estar más proyectada por la izquierda que por la derecha, al revés o igualmente proyectada si el observador está en el centro.
- Buscar un algoritmo que recorte la vista panorámica y proporcione la mayor región rectangular de dicha vista. Esto daría imágenes con buena calidad, que parecerían estar menos proyectadas como lo es la *Imagen 24*.
- Un fenómeno que sería interesante sería poder enrollar una vista panorámica en un cilindro o esfera en vez de simplemente proyectar y tener por tanto al menos dos vistas de como sería ese cilindro o esfera.

Referencias

- [1] http://www.cs.princeton.edu/courses/archive/fall05/cos429/papers/burt_adelson.pdf
- [2] <https://www.cs.toronto.edu/~urtasun/courses/CV/lecture08.pdf>
- [3] https://docs.opencv.org/3.1.0/dc/dff/tutorial_py_pyramids.html