

---

# Proyecto Final: Implementación Algoritmo Seam-Carving.

---

**UNIVERSIDAD DE GRANADA**  
**E.T.S.I. INFORMÁTICA Y TELECOMUNICACIÓN**



**UNIVERSIDAD  
DE GRANADA**

Departamento de Ciencias de la  
Computación e Inteligencia Artificial

Visión por Computador (2019-2020)

Daniel Bolaños Martínez  
José María Borrás Serrano

# Índice

<b>1. Introducción.</b>	<b>3</b>
<b>2. Implementación Seam-Carving.</b>	<b>3</b>
2.1. Generar el mapa de energía. . . . .	4
2.2. Calcular las costuras de menor energía. . . . .	6
2.3. Eliminar la costura de mínima energía. . . . .	8
2.4. Reducir imágenes. . . . .	9
<b>3. Optimización del tiempo del algoritmo.</b>	<b>10</b>
<b>4. Aumentar imágenes.</b>	<b>14</b>
<b>5. Redimensionar.</b>	<b>17</b>
<b>6. Eliminar objetos.</b>	<b>19</b>
<b>7. Amplificar contenido.</b>	<b>22</b>
<b>8. Tiempo funciones con y sin optimización.</b>	<b>23</b>
<b>9. Experimentación con imágenes.</b>	<b>24</b>
9.1. Reducir el tamaño de la imagen. . . . .	24
9.1.1. Limitaciones al reducir. . . . .	26
9.2. Aumentar el tamaño de la imagen. . . . .	28
9.3. Eliminar objetos en la imagen. . . . .	30
9.3.1. Limitaciones al eliminar objetos. . . . .	32
9.4. Ampliar objetos en la imagen. . . . .	33
9.4.1. Limitaciones al ampliar contenido. . . . .	34
<b>10. Conclusión.</b>	<b>35</b>

## 1. Introducción.

Para cambiar el tamaño de una imagen, existen dos alternativas básicas que se suelen usar normalmente: una de ellas es escalar la imagen de acuerdo con la altura y el ancho especificados y la otra realizar un recorte a la imagen para obtener una nueva imagen con las dimensiones requeridas.

Sin embargo, el escalado simple no conserva la perspectiva del contenido a menos que las nuevas dimensiones sean proporcionales a las dimensiones originales. Por otro lado, recortar filas o columnas enteras de una imagen es incluso una peor solución ya que directamente estamos eliminando información de la imagen que podría ser relevante.

El algoritmo de tallado de costuras o Seam-Carving (del artículo original *Seam Carving for Content-Aware Image Resizing* por Shai Avidan y Ariel Shamir) es útil para cambiar el tamaño de las imágenes mientras se conserva la perspectiva del contenido.

El algoritmo se basa en eliminar costuras de menor importancia (aquellas con menor diferencia de color entre los píxeles vecinos). El contenido más importante se conservará aún después de cambiar el tamaño. En este proyecto implementaremos el algoritmo Seam-Carving en Python para el ajuste de tamaño de imágenes (minimizar y aumentar) siguiendo los pasos e ideas establecidas en el artículo original. Además se mostrará su funcionamiento sobre diversas imágenes para ver cómo afecta a cada una de ellas.

## 2. Implementación Seam-Carving.

Formalmente definiremos una costura o hilo vertical de una imagen  $\mathbf{I}$  de dimensiones  $n \times m$  de la siguiente forma:

$$s^x = \{s_i^x\}_{i=1}^n = \{(x(i), i)\}_{i=1}^n, \forall i, |x(i) - x(i - 1)| \leq 1$$

donde  $x$  es un mapeado  $x : [1, \dots, n] \rightarrow [1, \dots, n]$

Intuitivamente una costura vertical es un camino 8-conectado de píxeles de la imagen desde la parte superior a la inferior de la misma, conteniendo un único píxel para cada fila de la imagen.

Definimos una costura o hilo horizontal como una costura vertical sobre una imagen  $\mathbf{I}$  que ha sido rotada  $90^\circ$  en sentido horario.

Dividiremos el proceso de implementación del algoritmo en las siguientes funcionalidades:

- Generar el mapa de energía asignando un valor de energía para cada píxel.
- Calcular las costuras de menor energía mediante programación dinámica.
- Eliminar todos los píxeles de la costura con menor energía.
- Repetir los pasos anteriores para el número deseado de filas y columnas.

Para cada sección, realizaremos pruebas del funcionamiento del algoritmo sobre el ejemplo básico de la imagen del castillo. En la última sección mostraremos el funcionamiento sobre imágenes diferentes.



Figura 1: Imagen original castillo.jpg

## 2.1. Generar el mapa de energía.

El primer paso es calcular el valor de energía para cada píxel para generar el mapa de energía de la imagen. Se pueden definir varias funciones para calcularla. Nosotros usaremos la siguiente fórmula:

$$e(\mathbf{I}) = \frac{\partial}{\partial x} \mathbf{I} + \frac{\partial}{\partial y} \mathbf{I},$$

donde  $\mathbf{I}$  es la imagen y para cada píxel en la imagen, en cada canal, calculamos las derivadas parciales de la imagen en los ejes  $\mathbf{X}$  e  $\mathbf{Y}$  y sumamos sus valores absolutos.

Notamos que en la implementación para calcular la energía usaremos una copia de la imagen en blanco y negro para así trabajar sobre un único canal.

Para calcular las derivadas parciales haremos uso de la función **ConvMasks1D** que hemos usado a lo largo de las prácticas y que calcula la convolución de una imagen usando máscaras de derivadas haciendo uso de la función **getDerivKernels** de *OpenCV*.

```
def mapa_energia(im, ksize=5):  
    im = im.astype(np.float32)  
    imGris = np.copy(im)  
  
    if len(im.shape)==3:  
        imGris=cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)  
  
    mapa_energia = abs(convMasks1D(imGris, 1, 0, ksize)) + abs(  
        convMasks1D(imGris, 0, 1, ksize))  
  
    return mapa_energia
```

La función **convMask1D** se ha implementado de la siguiente forma:

```
def convMasks1D(im, dx, dy, ksize, border=cv2.BORDER_DEFAULT):  
    im = im.astype(np.float32)  
    kerX, kerY = cv2.getDerivKernels(dx=dx, dy=dy, ksize=ksize, normalize=  
        True)  
    kerX = np.transpose(kerX)  
    kerX = np.flip(kerX)  
    kerY = np.flip(kerY)  
    imconv = cv2.filter2D(im, ddepth=-1, kernel=kerX,  
        borderType=border)  
    imconv = cv2.filter2D(imconv, ddepth=-1, kernel=kerY,  
        borderType=border)  
    return imconv
```

A continuación mostramos el mapa de energía de la imagen del castillo.



Figura 2: Mapa de energía castillo.jpg

## 2.2. Calcular las costuras de menor energía.

Nuestro próximo objetivo será encontrar un camino desde la parte superior de la imagen hasta la parte inferior de la imagen con la menor energía. Esta línea debe estar conectada: esto significa que cada píxel de la línea debe tocarse con el siguiente píxel de la línea, ya sea a través de un borde o una esquina.

Usaremos programación dinámica para buscar las costuras con menor energía de entre todas las posibles que comiencen con un píxel especificado.

Crearemos una matriz  $M$  para almacenar el valor mínimo de energía visto hasta ese píxel.  $M[i, j]$  contendrá el menor valor de la energía en el punto  $(i, j)$  de la imagen, considerando todas las uniones posibles hasta ese punto desde la parte superior de la imagen.

La energía acumulada (verticalmente) para el píxel en la posición  $(i, j)$  se calcula como:

$$M(i, j) = e(i, j) + \min(M(i - 1, j - 1), M(i - 1, j), M(i - 1, j + 1)). \quad (1)$$

donde  $e(i, j)$  es la energía del punto  $(i, j)$ .

La definición de  $M$  para costuras horizontales es análoga, intercambiando  $i$  por  $j$ .

Hemos implementado la función **seam\_vertical** en la cual calculamos el mapa de energía asociado a la imagen y en los bucles realizamos el calculo de la energía acumulada para cada píxel  $(i, j)$  según la fórmula (1).

```
def seam_vertical(im):
    fil , col = im.shape[0] , im.shape[1]

    M = mapa_energia(im)

    for i in range(1, fil):
        M[i,0] += np.min(M[i-1, 0:2])
        for j in range(1, col):
            M[i,j] += np.min(M[i-1, j-1:j+2])

    return M
```

A continuación se muestran las dos primeras costuras (vertical y horizontal) calculadas:



Figura 3: Primeras costuras vertical y horizontal.

## 2.3. Eliminar la costura de mínima energía.

En este paso borraremos la costura de menor energía y devolveremos la nueva imagen.

La energía mínima requerida para atravesar desde la parte superior de la imagen hasta la parte inferior estará presente en la última fila de la matriz  $M$ . La costura óptima se puede encontrar comenzando desde el píxel que contiene el mínimo acumulativo de energía y registrando los índices de píxeles que han contribuido a esa energía total. Necesitamos hacer *backtracking* para obtener la lista de píxeles presentes en esta costura.

Al mismo tiempo que obtenemos los índices de los píxeles que conforman la costura óptima (de menor energía), los vamos borrando, obteniendo finalmente la nueva imagen sin la costura.

Para eliminar una costura, creamos una copia de la imagen con los píxeles de la costura eliminados, disminuyendo así en uno el número de columnas de la imagen.

Hemos implementado esta funcionalidad en la función **cortar\_columna**, en la que eliminamos la costura de mínima energía de la imagen.

Calculamos la matriz que contiene las costuras de menor energía para la imagen y buscamos el valor mínimo de la última fila, el cual nos indicará la costura vertical de mínima energía, dicho valor será calculado utilizando la función **argmin**. Finalmente realizaremos backtracking sobre dicha costura para encontrar el camino de vuelta y guardaremos la nueva imagen con la costura eliminada en la copia inicial.

Para implementar el *backtracking*, buscaremos para cada elemento el mínimo situado en la fila superior en la misma columna y sus dos columnas adyacentes. Tendremos en cuenta que cuando estamos en un elemento de la costura situado en la columna 0, no podemos salirnos del rango de la imagen hacia la izquierda.

```
def cortar_columna(im):
    fil, col = im.shape[0], im.shape[1]
    nueva_im = np.empty((fil, col - 1, 3))

    M = seam_vertical(im)

    j = np.argmin(M[fil - 1, 0:col])
    nueva_im[fil - 1] = np.delete(im[fil - 1], j, axis=0)
    for i in range(2, fil + 1):
        if (j == 0):
            j += np.argmin(M[fil - i, 0:2])
```

```

else :
    j=j-1+np.argmax(M[ fil-i , j-1:j+2])
    nueva_im[ fil-i ] = np.delete(im[ fil-i ] , j , axis=0)

return nueva_im

```

## 2.4. Reducir imágenes.

La función principal del algoritmo Seam-Carving es reducir el tamaño de una imagen. Este proceso consiste en eliminar un número de columnas y filas determinadas aplicando reiteradamente los 3 pasos anteriores.

Para eliminar un número de columnas determinado  $n$  de la imagen, usaremos la función **podar\_columnas** que simplemente llamará  $n$  veces a la función **cortar\_columna**. Es decir, elimina las  $n$  costuras verticales de menor energía.

```

def podar_columnas(im, n):
    for i in range(n):
        im = cortar_columna(im)
    return im

```



Figura 4: Reducir columnas a la mitad.

Para eliminar las filas en lugar de columnas implementaremos una función **podar\_filas** realizando un giro de  $90^\circ$  en sentido horario sobre la imagen y usando la función que hemos usado para las columnas. Finalmente realizaremos otro giro de  $90^\circ$

en sentido contrario para obtener la imagen en la posición inicial.

```
def podar_filas(im, n):
    im = np.rot90(im, 1, (0, 1))
    im = podar_columnas(im, n)
    im = np.rot90(im, 3, (0, 1))
    return im
```



Figura 5: Reducir filas a la mitad.

En las siguientes secciones siempre hablaremos de implementaciones sobre columnas, ya que las funciones sobre las filas se realizarán como se ha descrito arriba, rotando la imagen.

### 3. Optimización del tiempo del algoritmo.

Uno de los principales problemas del algoritmo Seam-Carving es el tiempo de ejecución, que puede llegar a ser excesivo.

Como estamos implementando el código en Python, y este es un lenguaje con un tiempo de ejecución más lento comparados a otros como C++, vamos a utilizar Numba para reducir automáticamente el tiempo necesario para la ejecución.

Numba es un compilador JIT que traduce un subconjunto de código Python y Numpy a código máquina más rápido. Está diseñado para usarse con arrays de Numpy y traduce funciones a código máquina optimizado durante la ejecución utilizando la librería del compilador estándar LLVM,

Utilizaremos Numba para la función que calcula los hilos de la imagen, porque principalmente se realizan operaciones sobre arrays Numpy. Para su uso simplemente

importamos `jit` al principio del código y ponemos `@jit` antes de la definición de función:

```
@jit
def seam_vertical(im):
    ...
```

Ahora nos centraremos en modificar el algoritmo, para ello realizamos un análisis para encontrar cuales son las partes del algoritmo que requieren mayor tiempo de procesado.

Para cada eliminación de cada costura, el mapa de energía de la imagen se recalcula para tener en cuenta las energías cambiantes debido a la eliminación de la costura anterior. Esto en principio podría parecer muy costoso pero a la hora de generar el mapa de energía realmente se poco tiempo de ejecución.

Donde realmente se encuentra la mayor carga es que para cada costura que eliminamos tenemos que calcular todas las costuras posibles de la imagen que tengan la menor energía. Esto es muy costoso tanto computacionalmente como en tiempo, así que la optimización que hemos realizado se centrará fundamentalmente en reducir el número de costuras que debemos calcular para reducir cada columna/fila de la imagen.

Primero, realizaremos unos pequeños cambios en la función `seam_vertical` que reducirán ligeramente el tiempo.

Ofrecemos la posibilidad de pasarle como parámetro el mapa de energía. De esta forma, si se lo pasamos, lo copiará en la matriz  $M$  y si no se lo pasamos, calculará el mapa de energía y lo meterá en la misma matriz  $M$ .

Además creamos una nueva matriz *backtrack* donde almacenaremos los valores de *backtracking*. Al realizar el cálculo del backtracking en esta función aprovechamos las mejoras en tiempo que ofrece Numba.

```
@jit
def seam_vertical_O2(im, mapa_e = np.empty((0,0)):

    fil, col = im.shape[0], im.shape[1]
    backtrack = np.empty((fil, col), dtype=np.int)

    if len(mapa_e)==0:
        M = mapa_energia(im)
    else:
        M = np.copy(mapa_e)
```

```

for i in range(1, fil):
    indice = np.argmin(M[i-1, 0:2])
    backtrack[i, 0] = indice
    M[i, 0] += M[i-1, indice]
    for j in range(1, col):
        indice = np.argmin(M[i-1, j-1:j+2]) + j - 1
        backtrack[i, j] = indice
        M[i, j] += M[i-1, indice]

return M, backtrack

```

Para reducir significativamente el número de cálculos a realizar para la obtención de la costura de mínima energía nos basaremos en que si una costura tiene la menor energía, entonces es muy probable que las costuras cercanas a ella también tengan baja energía y la siguiente costura de menor energía (o al menos una costura con energía similar) se encuentre en un entorno de la costura mínima anterior.

Así, en lugar de calcular todas las costuras y el mapa de energía de toda la imagen cada vez que vayamos a eliminar otra costura de mínima energía, lo queharemos será trabajar solamente con una parte de la imagen. Dicha parte se corresponde con un entorno de la costura de mínima energía calculada anteriormente.

De esta manera, la primera vez calculamos todos las costuras para obtener la costura mínima y para las demás iteraciones operamos en un entorno definido de la siguiente forma:

Cuando la costura es vertical, el entorno comprende todas las filas de la imagen pero sólo un intervalo de las columnas [col\_min, col\_max], dicho intervalo depende de la costura eliminada anterior. Así:

- col\_min = columna más a la izquierda por la que ha pasado la costura - valor de intervalo,
- col\_max = columna más a la derecha por la que ha pasado la costura + valor de intervalo.

Donde valor de intervalo es un valor fijo que sirve para que el intervalo de las columnas en que calculamos las costuras sea mayor o menor. Cuanto menor sea el intervalo menor será el tiempo de ejecución, pero también será menor la probabilidad de obtener la costura de mínima energía.

Realizando varias pruebas hemos llegado a fijar el valor de intervalo en 50, que ha sido el valor que mejor relación entre calidad de imagen y tiempo ha proporcionado.

Claramente también hay que tener en cuenta que [col\_min, col\_max] sea un intervalo válido, por lo que col\_min no puede ser menor que 0 y col\_max no puede ser mayor que el número actual de columnas de la imagen a la que le hemos quitado la costura.

Adicionalmente, a la hora de implementar la eliminación de la costura vamos a utilizar la funcionalidad **reshape** en lugar de np.delete. Para ello, tenemos una matriz de *booleanos* con valores de **True** si la posición se mantiene y **False** si se debe eliminar.

A continuación mostraremos el código de las nuevas funciones **cortar\_columna** y **podar\_columnas** con las modificaciones que se han especificado anteriormente.

```
def cortar_columna_O2(im, col_min, col_max):
    intervalo = 50
    fil, col = im.shape[0], im.shape[1]
    matriz_bool = np.ones((fil, col, 3), dtype=np.bool)

    M, backtrack = seam_vertical_O2(im[0:fil, col_min:col_max])

    j = np.argmin(M[fil-1, 0:M.shape[1]])
    mini = j
    maxi = j
    for i in range(1, fil):
        matriz_bool[fil-i, col_min+j] = (False, False, False)
        j = backtrack[fil-i, j]
        mini = min(mini, j)
        maxi = max(maxi, j)
    matriz_bool[0, col_min+j] = (False, False, False)

    mini = max(col_min + mini - intervalo, 0)
    maxi = min(col_min + maxi + intervalo, col - 1)

    return im[matriz_bool].reshape((fil, col-1, 3)), mini, maxi
```

```
def podar_columnas_O2(im, n):
    col_min=0
    col_max=im.shape[1]
    for i in range(n):
        im, col_min, col_max = cortar_columna_O2(im, col_min, col_max)

    return im
```

## 4. Aumentar imágenes.

Si en vez de reducir la imagen queremos aumentarla, debemos agregar nuevas costuras. El principal problema de ello es que si en cada paso seleccionamos la costura óptima con menor energía y la replicamos, la próxima vez, la costura escogida será la misma, por lo que el resultado obtenido será una nueva imagen en la que hemos añadido múltiples veces la misma costura obteniendo un resultado visualmente incorrecto. Por tanto lo que haremos será escoger un número  $n$  de costuras diferentes con mínima energía y serán esas las que repliquemos.

La idea será trabajar siempre sobre el mismo mapa de energía que será calculado solo una vez y cada vez que obtengamos una costura mínima distinta daremos el valor infinito a las posiciones del mapa de energía de dicha costura. De esa forma conseguiremos que no se repitan las costuras de mínima energía.

Como en una imagen el número de costuras de mínima energía disjuntas que podemos obtener está limitado, aunque queramos no siempre será posible obtener un número  $n$  de costuras diferentes, a veces como mucho podremos obtener un número  $m$  de costuras distintas con  $m < n$ .

Por ello crearemos una variable *num\_seam* con el número de costuras mínimas diferentes que tenemos. Además tendremos una variable booleana llamada *terminar* que inicializaremos a **False** y que pondremos a **True** si no podemos obtener más costuras de manera que no se intersequen.

Igual que antes, trabajamos sobre una parte de la imagen  $[col\_min, col\_max]$  como se ha descrito antes, con la diferencia de que si en ese intervalo no podemos obtener otra costura mínima disjunta, entonces trabajaremos con toda la imagen, es decir, en el intervalo  $[col\_min = 0, col\_max = col]$  donde  $col$  es el número de columnas totales de la imagen.

Tendremos una matriz booleana llamada *mapa\_seam* donde marcaremos como **True** las posiciones por las que pasa alguna de las costuras mínimas que hemos obtenido.

```
def aniadir_columnas_O(im,n):
    fil , col = im.shape [0] , im . shape [1]
    num_seam=0
    mapa_e = mapa_energia(im)
    mapa_seam = np . zeros (( fil , col ) , dtype=np . bool )
    terminar=False
    intervalo = 100
    col_min = 0
```

```

col_max = col
while (num_seam < n and not(terminar)):
    M, backtrack = seam_vertical_O2(im[0:fil , col_min:col_max] , mapa_e[0:
        fil , col_min:col_max])
    j = np.argmin(M[ fil -1, 0:M.shape [1]])
    if(M[ fil -1, j ] == math.inf):
        if(col_min == 0 and col_max == col):
            terminar=True
        else:
            col_min = 0
            col_max = col
    else:
        mini = j
        maxi = j
        for i in range(1,fil):
            mapa_seam[ fil -i , col_min+j ] = True
            mapa_e[ fil -i , col_min + j]=math.inf
            j = backtrack[ fil -i , j ]
            mini = min(mini , j )
            maxi = max(maxi , j )

        mapa_seam[0 , col_min+j ] = True
        mapa_e[0 , col_min + j]=math.inf
        col_min = max( col_min+mini-intervalo , 0)
        col_max = min( col_min+maxi+intervalo , col)
        num_seam += 1
return mapa_seam , num_seam

```

Con la función anterior hemos obtenido el mapa por donde pasan las costuras de mínima energía y el número de dichas costuras.

A continuación, crearemos una nueva imagen donde añadiremos los valores de la imagen original y repetiremos los que pertenezcan a alguna costura el número de veces necesario hasta que la imagen tenga el número de columnas especificado.

Con el valor  $k$ , especificado como parámetro, dividido entre el número de columnas de la imagen obtenemos el máximo número de costuras diferentes que utilizaremos para aumentar la imagen. Así, sólo utilizaremos el  $\frac{1}{k} \cdot 100\%$  de costuras con la menor energía.

Sea  $n$  el número de columnas en que queremos aumentar la imagen y  $num\_seam$  el número de costuras diferentes obtenidas, cada valor de cada costura se deberá repetir  $\text{int}(n/num\_seam)$  veces y además el número de costuras que deberemos añadir una vez adicional para completar el número de columnas viene determinado por  $\text{int}(num\_seam * ((n/num\_seam) - \text{int}(n/num\_seam)))$ .

Por último hacemos un bucle en el que añadiremos cuidadosamente los valores de la nueva imagen.

```
def aumentar_columnas_O(im, n, k=3):
    if n == 0:
        return im

    fil, col = im.shape[0], im.shape[1]
    mapa_seam, num_seam = anadir_columnas_O(im, min(col//k, n))
    nueva_im = np.zeros((fil, col+n, 3))

    repetir = n//num_seam
    extra = int(num_seam * (n/num_seam - (n // num_seam) ) )

    for i in range(fil):
        cont_seam=0
        cont_extra=0
        for j in range(col):
            nueva_im[i,j+cont_seam]=im[i,j]
            if (mapa_seam[i,j]==True):
                for k in range(repetir):
                    cont_seam+=1
                    nueva_im[i,j+cont_seam]=im[i,j]
                if (cont_extra < extra):
                    cont_extra+=1
                    cont_seam+=1
                    nueva_im[i,j+cont_seam]=im[i,j]

    return nueva_im
```

Tanto este código como el resto, puede verse comentado línea a línea en el archivo *proyecto.py*. Para la memoria se han eliminado los comentarios y se ha hecho una explicación general de cada función para evitar saturarla.



Figura 6: Aumenta columnas al doble.

## 5. Redimensionar.

Una vez que podemos reducir y aumentar las dimensiones de la imagen utilizando seam-carving, entonces podemos agrupar ambas funciones en redimensionar la imagen.

Para el redimensionamiento especificaremos las dimensiones que queremos para la imagen y teniendo en cuenta si dichas dimensiones son menores o mayores comparadas con las originales, pasaremos a ejecutar el algoritmo de reducción o el de aumento respectivamente. La implementación del código realizada es:

```
def resize_seam(im, n_fil, n_col):
    fil, col = im.shape[0], im.shape[1]
    im_seam=np.copy(im)

    if (col > n_col):
        im_seam=podar_columnas_O2(im_seam, col-n_col)
    else:
        im_seam=aumentar_columnas_O(im_seam, n_col-col)
    if (fil > n_fil):
        im_seam=podar_filas_O2(im_seam, fil-n_fil)
    else:
        im_seam=aumentar_filas_O(im_seam, n_fil-fil)

    return im_seam
```

Teniendo en cuenta que el problema del redimensionamiento clásico de una imagen, sin emplear seam carving, es el cambio de la relación de aspecto. Si se realizará un redimensionamiento mediante **cv2.resize** en el que el número de filas y columnas son modificadas según la misma escala, entonces no se cambiaría la relación de aspecto y el redimensionamiento sería óptimo.

Siguiendo este argumento podemos realizar la siguiente modificación para mejorar el tiempo de ejecución y en general la calidad de la imagen.

Si a la hora de redimensionar tenemos que aumentar tanto el número de filas como de columnas simultáneamente o reducirlo, entonces obtendremos la escala en la que se va a modificar la imagen tanto para las columnas como para las filas. Tomaremos el mínimo de las dos escalas y aplicaremos **cv2.resize** manteniendo la relación de aspecto para esa escala. Finalmente, aplicaremos seam-carving para adaptar la escala al factor restante.

```

def resize_seam_O(im, n_fil, n_col, k=3):
    fil, col = im.shape[0], im.shape[1]
    im_seam=np.copy(im)

    if(n_fil>fil and n_col>col):
        escala_min = min(n_fil/fil, n_col/col)
        im_seam = cv2.resize(im_seam, (int(col*escala_min), int(fil*escala_min)))
        im_seam=aumentar_columnas_O(im_seam, n_col - int(col*escala_min), k)
        im_seam=aumentar_filas_O(im_seam, n_fil - int(fil*escala_min), k)
    else:
        if(n_fil<fil and n_col<col):
            escala_max = max(n_fil/fil, n_col/col)
            im_seam = cv2.resize(im_seam, (int(col*escala_max), int(fil*escala_max)))
            im_seam=podar_columnas_O2(im_seam, int(col*escala_max) - n_col)
            im_seam=podar_filas_O2(im_seam, int(fil*escala_max) - n_fil)
        else:
            im_seam=resize_seam(im, n_fil, n_col, k)

    return im_seam

```

Realizamos varias redimensiones con factos de escalado (filas, columnas), sobre la imagen de ejemplo:



Figura 7: Imagen original. / Escala (0.75,0.5).



Figura 8: Escala (2.0,1.5). / Escala (1.5,2.0).

## 6. Eliminar objetos.

Teniendo en cuenta el funcionamiento de seam-carving podemos utilizarlo para eliminar objetos en una imagen. Para ello, añadiremos una máscara que recubra al objeto que queremos eliminar y disminuiremos en ese espacio los valores de energía asignando un valor negativo muy grande. De esta forma, las costuras de energía mínimas que queremos eliminar, pasarán sobre el objeto y obtendremos una nueva imagen sin ese objeto manteniendo una calidad similar a la de la imagen original.

La diferencia respecto al código de reducción anterior aparece destacada entre `/* */`.

Para calcular las costuras teniendo en cuenta los elementos a eliminar simplemente calculamos las costuras como antes pero añadiendo en la posición de los píxeles a eliminar un valor negativo grande.

```
def seam_vertical_eliminar(im, mapa_eliminar):
    fil, col = im.shape[0], im.shape[1]
    backtrack = np.empty((fil, col), dtype=np.int)
    M = mapa_energia(im)

    /* **** */
    for i in range(fil):
        for j in range(col):
            if(mapa_eliminar[i, j]==1):
                M[i, j]=-10000
    /* **** */

    for i in range(1, fil):
        indice = np.argmin(M[i-1, 0:2])
```

```

    backtrack[i, 0] = indice
    M[i, 0] += M[i-1, indice]
    for j in range(1, col):
        indice = np.argmin(M[i-1, j-1:j+2]) + j - 1
        backtrack[i, j] = indice
        M[i, j] += M[i-1, indice]
    return M, backtrack

```

Una vez que hemos calculado las costuras, pasamos a eliminar la costura vertical de mínima energía. La diferencia con respecto de la función **cortar\_columna\_O2** es que le pasamos la máscara del objeto a eliminar y llevaremos la cuenta de cuantos píxeles de los que forman parte de la máscara a eliminar hemos borrado. Esto es necesario para establecer un criterio de parada para el algoritmo de eliminar objetos.

```

def cortar_columna_eliminar(im, col_min, col_max, mapa_eliminar):
    fil, col = im.shape[0], im.shape[1]
    intervalo = 50
    matriz_bool = np.ones((fil, col, 3), dtype=np.bool)

    M, backtrack = seam_vertical_eliminar(im[0:fil, col_min:col_max],
                                           mapa_eliminar[0:fil, col_min:col_max])

    j = np.argmin(M[fil-1, 0:M.shape[1]])
    mini = j
    maxi = j
    for i in range(1, fil):
        # *****
        if (mapa_eliminar[fil-i, col_min+j]==True):
            eliminados+=1
        # *****
        matriz_bool[fil-i, col_min+j] = (False, False, False)
        j = backtrack[fil-i, j]
        mini = min(mini, j)
        maxi = max(maxi, j)
        # *****
        if (mapa_eliminar[0, col_min+j]==True):
            eliminados+=1
        # *****
        matriz_bool[0, col_min+j] = (False, False, False)

        mini = max(col_min + mini - intervalo, 0)
        maxi = min(col_min + maxi + intervalo, col - 1)
        matriz_bool2 = matriz_bool[0:fil, 0:col, 0]

    return im[matriz_bool].reshape((fil, col-1,3)), mapa_eliminar[
        matriz_bool2].reshape((fil, col-1)), eliminados, mini, maxi

```

Para eliminar el objeto pasamos la imagen en la que se encuentra y un array con el conjunto de píxeles que ocupa dicho objeto. Transformamos dicho array en una matriz con las dimensiones de la imagen en la que se indica con **True** o **False** si ese elemento se debe o no eliminar, respectivamente. Quitaremos columnas mediante la función anterior hasta que no quede ninguno de los píxeles a eliminar.

Si al calcular una costura mínima, ésta no pasa por ninguno de los píxeles a eliminar, entonces el intervalo en el que se han calculado las costuras no es el correcto. Para solucionarlo, volvemos a calcular las costuras de toda la imagen.

```
def eliminar_objeto_vertical(im, pixels):

    fil, col = im.shape[0], im.shape[1]
    mapa_eliminar = np.zeros((fil, col), dtype=np.bool)

    for pix in pixels:
        mapa_eliminar[pix[0]][pix[1]] = True

    num_eliminar = len(pixels)
    col_min=0
    col_max=col

    while (num_eliminar > 0):
        im, mapa_eliminar, eliminados, col_min, col_max =
            cortar_columna_eliminar(im, col_min, col_max, mapa_eliminar)
        if(eliminados == 0):
            col_min=0
            col_max=im.shape[1]
            num_eliminar = num_eliminar - eliminados
    return im
```



Figura 9: Seleccionamos a la persona en la foto.



Figura 10: Eliminamos persona de la foto con seams verticales y horizontales.

Podemos observar en la imagen 10 como la eliminación con seams verticales (izquierda) es visualmente más satisfactoria que la de seams horizontales (derecha). Esto se debe a que la persona se puede considerar, en esta imagen, más un objeto vertical que horizontal.

## 7. Amplificar contenido.

En lugar de aumentar el tamaño de la imagen, podemos utilizar seam carving para ampliar los contenidos más relevantes de la imagen sin modificar sus dimensiones. Para ello, primero realizamos un escalado de la imagen, mediante la función **cv2.resize**, para aumentar su tamaño y después aplicamos seam carving para reducir la imagen a su tamaño original.

De esta forma logramos ampliar algunos de los contenidos con mayor energía de la imagen. Se ha implementado de la siguiente manera:

```
def amplificacion_seam(im, escala_vertical=2, escala_horizontal=2):
    im_seam=np.copy(im)
    fil , col = im.shape [0] , im .shape [1]
    n_fil = fil
    n_col = col
    if(escala_horizontal > 1):
        n_col = int(col*escala_horizontal)
    if( escala_vertical > 1):
        n_fil = int(fil*escala_vertical)
    im_seam = cv2.resize(im_seam, (n_col , n_fil))
    im_seam=podar_columnas_O2(im_seam, n_col-col)
    im_seam=podar_filas_O2(im_seam, n_fil-fil)

    return im_seam
```



Figura 11: Amplificar imagen.

## 8. Tiempo funciones con y sin optimización.

Además de las optimizaciones descritas, hemos realizado otra optimización de **podar\_columnas**, a la que hemos denominado *O1* similar a la explicada, pero con la diferencia de que cada cierto número de iteraciones vuelve a recalcular el mapa de energía de cero y todas las costuras asociadas. El código no ha sido descrito debido a que como podemos observar en la tabla, obtiene resultados peores a la versión *O2*. Para la función de **aumentar\_columnas** también hemos un código sin optimizar, en el que básicamente se calculan todas las costuras de la imagen, es decir, no nos limitamos a un entorno. Además añadimos algunos tiempo sin Numba, en los cuales obtenemos resultados idénticos de imagen pero mucho más lentos.

Ejemplo:	castillo.jpg	
Función	Optimización	Tiempo(seg.)
podar_columnas factor: 0.5	normal sin Numba	558.489
	normal	27.428
	O1	16.784
	O2	8.853
	O2 sin Numba	116.483
aumentar_columnas factor: 0.5	normal	30.337
	O	4.583
resize_seam factor: (1.5,2.0)	normal	11.117
	O	6.507

Tabla 1: Tabla comparativa optimizaciones funciones.

## 9. Experimentación con imágenes.

Para las diversas pruebas del algoritmo, hemos seleccionado una muestra de 35 imágenes de paisajes desérticos y marítimos que tienen como característica en común que contienen información relevante en alguna zona de la imagen y paisaje en el resto de la misma.

Las imágenes han sido escogidas para demostrar el funcionamiento de las diferentes aplicaciones que hemos abordado en el proyecto y que se proponían como usos del algoritmo Seam-Carving.

El ordenador donde hemos ejecutado el trabajo tiene las siguientes especificaciones:

**Procesador:** Intel Core i5-7200U CPU 2.50GHz 4

**Gráficos:** Intel HD Graphics 620 (Kaby Lake GT2)

**Sistema Operativo:** Ubuntu 18.04.1 LTS

### 9.1. Reducir el tamaño de la imagen.

Hemos aplicado reducción a la mitad de columnas utilizando la función `podar_columnas_O2` sobre las 35 imágenes escogidas y las guardamos en la carpeta resultados. A continuación mostraremos algunas muestras de los resultados obtenidos:



Figura 12: Reducir a la mitad en columnas imagen 20.



Figura 13: Reducir a la mitad en columnas imagen 24.



Figura 14: Reducir a la mitad en columnas imagen 6.

### 9.1.1. Limitaciones al reducir.

Podemos comprobar que aplicando Seam-Carving sobre imágenes con personas en primer plano, obtenemos unos resultados erróneos, debido a que partes de las personas tienen una energía baja pero su información sí es relevante, lo que dificulta la correcta aplicación del algoritmo.



Figura 15: Reducir a la mitad en columnas imagen 18.



Figura 16: Reducir a la mitad en columnas imagen 16.

Seam-carving también devuelve malos resultados en imágenes que deben conservar ciertas geometrías. En el ejemplo mostrado a continuación, vemos como el arco se deforma al aplicarle el algoritmo.

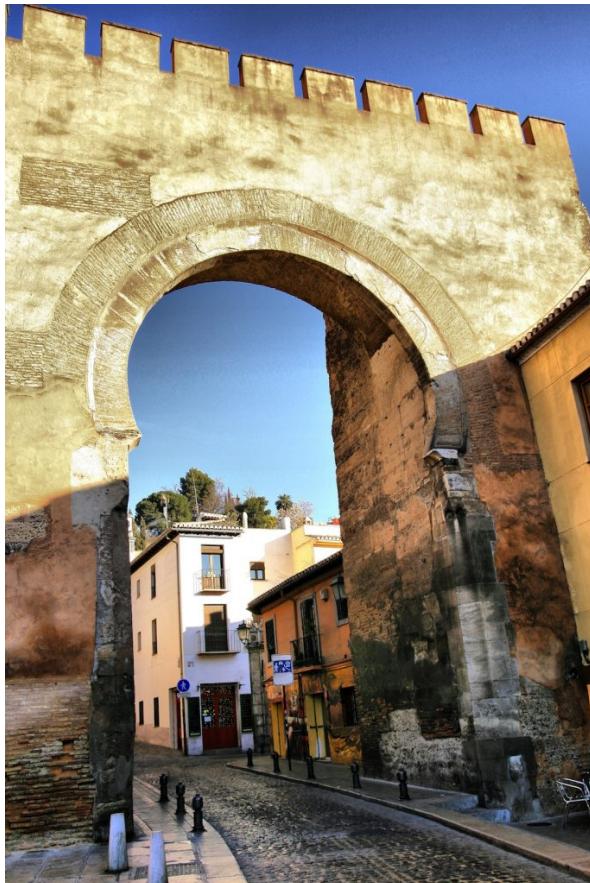


Figura 17: Reducir a la mitad en columnas imagen 36.

## 9.2. Aumentar el tamaño de la imagen.

A continuación mostraremos dos ejemplos de las imágenes de muestra a la que se le ha aplicado un factor de escala para aumentar el tamaño de la imagen en filas y columnas utilizando la función **resize\_seam\_O**. Podemos ver que obtiene buenos resultados para las imágenes seleccionadas.



Figura 18: Imagen 14 original.



Figura 19: Aumentar en factor de escala (1.5,2.0)



Figura 20: Imagen 24 original.



Figura 21: Aumentar en factor de escala (1.0,1.5)

### 9.3. Eliminar objetos en la imagen.

A continuación mostraremos dos ejemplos de eliminación de objetos en una imagen, en particular, la eliminación de dos personas. Utilizaremos la función `eliminar_objeto_vertical` pasándole como parámetro las posiciones de los píxeles que ocupan.



Figura 22: Seleccionar persona a eliminar.

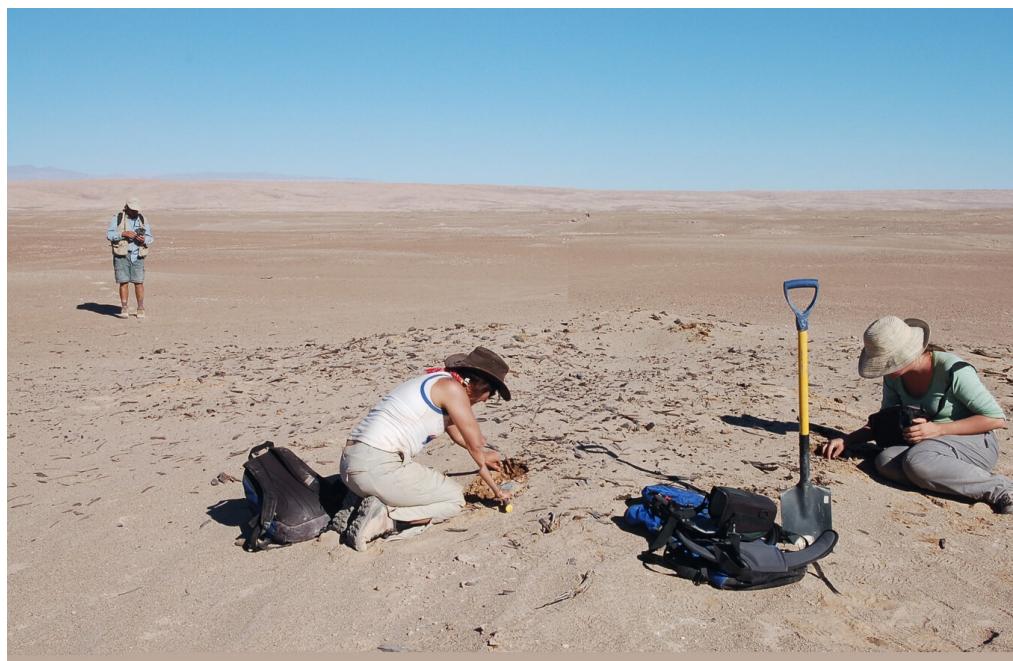


Figura 23: Eliminar persona de la imagen 1.



Figura 24: Seleccionar persona a eliminar.



Figura 25: Eliminar persona de la imagen 15.

### 9.3.1. Limitaciones al eliminar objetos.

Seam-carving da buenos resultados cuando eliminamos objetos relativamente pequeños en la imagen, sin embargo no es ideal cuando pretendemos eliminar una parte muy grande de la imagen.

En el ejemplo siguiente aparece una imagen con varios barcos, se pueden ver los resultados eliminando el barco principal que ocupa la mayor parte de la imagen comparado con uno de los barcos del fondo de la misma que es más pequeño.



Figura 26: Imagen original 23.



Figura 27: Seleccionar barcos a eliminar.



Figura 28: Resultados al eliminar barcos.

#### 9.4. Amplificar objetos en la imagen.

Finalmente mostraremos dos ejemplos para la ampliación de detalles en imágenes usando la función **amplificacion\_seam**. Podemos ver los resultados a continuación (notamos que las imágenes tienen las mismas dimensiones):



Figura 29: Ampliar contenido de la imagen 13. Factor (1.5,1.5).



Figura 30: Ampliar contenido de la imagen 26. Factor (1.7,1.7).

#### 9.4.1. Limitaciones al amplificar contenido.

Seam-carving en general no da buenos resultados cuando la inmensa mayoría de la imagen tiene una energía muy alta, no hay zonas de energía baja que se puedan eliminar sin distorsionar la imagen significativamente.

En el ejemplo siguiente, podemos ver como la amplificación del contenido de la catedral hace que las columnas se deformen.



Figura 31: Ampliar contenido de la imagen 37. Factor (2.0,2.0).

## 10. Conclusión.

Con Seam-Carving buscamos sobre todo realizar redimensionado de imágenes de forma que la calidad sea mejor que realizando un redimensionado clásico como el que realiza la función `cv2.resize` cuando no se mantiene la relación de aspecto original.

Además nos permite otras funcionalidades como la eliminación de ciertas partes de la imagen sin estropear la calidad de la misma o aumentar el tamaño de los detalles más importantes de la imagen manteniendo las mismas dimensiones.

Como hemos visto en los ejemplos, en imágenes que contengan zonas con alta energía y otra zona con poca energía que se puedan eliminar sin perder información relevante de la foto, este algoritmo funciona bastante bien especialmente a la hora de hacer pequeñas reducciones o aumentos del tamaño de la imagen y eliminar objetos pequeños.

Sin embargo, no podemos asegurar que el resultado al utilizar Seam-Carving vaya a ser idóneo ya que dependiendo del tipo de la imagen podemos encontrarnos con resultados distorsionados.

Por ejemplo, en imágenes que no contengan zonas con baja energía o en las que toda la información sea relevante, este algoritmo dará probablemente malos resultados. De igual forma tampoco se pueden eliminar objetos que ocupen una gran proporción en la imagen de una forma satisfactoria.

La mayor parte del tiempo de ejecución del algoritmo se dedica a calcular las costuras, por ello para optimizarlo podemos calcular las costuras en un intervalo de la imagen en vez de calcular las costuras de toda la imagen. Dependiendo del intervalo que utilicemos obtendremos mejores o peores resultados con un tiempo de ejecución menor o mayor.

## Referencias

- [1] S. AVIDAN Y A. SHAMIR. *Seam Carving for Content-Aware Image Resizing*.  
<https://www.win.tue.nl/~wstahw/edu/2IV00/seamcarving.pdf>
- [2] Documentación Numba JIT: <http://numba.pydata.org/numba-doc/latest/user/index.html>
- [3] Seam Carving GUI: <https://code.google.com/archive/p/seam-carving-gui/downloads>