UNIVERSIDAD DE ALMERÍA

TRATAMIENTO DIGITAL DE IMÁGENES

Redimensionado de Imágenes

Alberto Ángel Fuentes Funes
Grado en Ingeniería Informática
Curso 2019/2020
Mayo 2020

Resumen.

El procesamiento de imágenes, o también llamado Tratamiento de imágenes tiene como objetivo mejorar el aspecto de las imágenes y hacer más evidentes en ellas ciertos detalles, con la capacidad de analizar imágenes del mundo real adquiriendo información relevante de estas, para poder resolver las tareas de algún problema.

El tratamiento de las imágenes se puede en producir por medio de métodos ópticos, o bien por medio de métodos digitales, en una computadora.

En la rama de la informática, el tratamiento digital de imágenes es el uso de una computadora digital para procesar imágenes digitales a través de un algoritmo. Este proceso tiene muchas ventajas sobre el procesamiento de imágenes analógicas. Permite aplicar algoritmos a los datos de entrada.

La entrada de ese sistema es una imagen digital y el sistema procesa esa imagen usando algoritmos eficientes, y da una imagen como salida. Esta salida puede evitar problemas como la acumulación de ruido y distorsión durante el procesamiento. El ejemplo más común, en el desarrollo de un sistema informático capaz de realizar el procesamiento de una imagen, es el Adobe Photoshop. Es una de las aplicaciones más utilizadas para procesar imágenes digitales.

Indice.

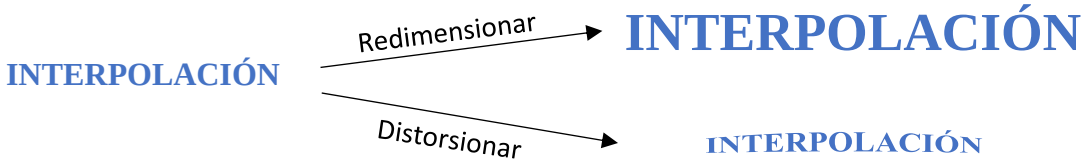
Resumen. **2**
Indice. **3**
Descripción del proyecto. **4**
Método de interpolación del vecino más cercano. **5**
Ejemplos de aplicación. 7
Conclusiones del algoritmo. 8
Método de interpolación bilineal. **9**
Ejemplos de aplicación. 10
Conclusiones del algoritmo. 12
Método de interpolación bicúbica. **13**
Ejemplos de aplicación. 15
Conclusiones del algoritmo. 16
Comparativas **17**
Otros métodos **18**
Anexo **19**
Bibliografía **24**

Descripción del proyecto.

El primer objetivo de este proyecto es adquirir un conocimiento más profundo sobre el tratamiento de digital de imágenes y sobre programación en C++ .

Este informe tiene como objetivo proporcionar una interpolación de imágenes mediante algoritmos de redimensionado. Redimensionar es el proceso de establecer de nuevo las dimensiones exactas o el valor preciso de algo.

La interpolación de imágenes aparece en todas las fotos digitales en algún momento. Ocurre cada vez que se cambia el tamaño o se reasigna (distorsiona) una imagen de una cuadrícula de píxeles a otra. El cambio de tamaño de la imagen es necesario cuando necesita aumentar o disminuir el número total de píxeles, mientras que la reasignación puede ocurrir en una variedad más amplia de escenarios: corregir la distorsión de la lente, cambiar la perspectiva y rotar una imagen.



Incluso si se realiza el mismo cambio de tamaño o reasignación de la imagen, los resultados pueden variar significativamente según el algoritmo de interpolación. Es solo una aproximación, por lo tanto, una imagen siempre perderá algo de calidad cada vez que se realice la interpolación.

En esta práctica solo trabajaremos con el proceso de interpolación de redimensionado. Probaremos y codificaremos los algoritmos más comunes para cambiar el tamaño de las imágenes y comentaremos detalladamente cual han sido los resultados. Cuando tratemos de ampliar la imagen deberemos establecer un redimensionado mayor al 100%. En caso de reducir el tamaño de la imagen, utilizaremos un redimensionado menor al 100%.

Trabajaremos con imágenes, que no han sido creadas por ordenador, en un formato que previamente se ha establecido en .bmp. BMP es un formato de imagen de mapa de bits que puede guardar imágenes de 24 bits o menor. Este formato puede proporcionar a los archivos una compresión sin pérdida de calidad.

En nuestro caso trabajaremos con imágenes a color, que posteriormente será transformadas a escala de grises y las podremos utilizar para aplicar los siguientes algoritmos:

- Método de interpolación del vecino más cercano.
- Método de interpolación bilineal.
- Método de interpolación bicúbica.

Método de interpolación del vecino más cercano.

Uno de los algoritmos más utilizados por su simplicidad es el de la **interpolación del vecino más cercano** o también conocido como “interpolación proximal o muestreo de puntos”.

La interpolación del vecino más cercano es el enfoque más simple para la interpolación. En lugar de calcular un valor promedio mediante algunos criterios de ponderación o generar un valor intermedio basado en reglas complicadas, este método simplemente determina el píxel vecino "más cercano" y asume el valor de intensidad del mismo. El algoritmo no da suficiente precisión para hacer un buen registro.

Para mostrar cómo funciona, mostraremos primero un ejemplo en un espacio unidimensional. *[Figura 1]*

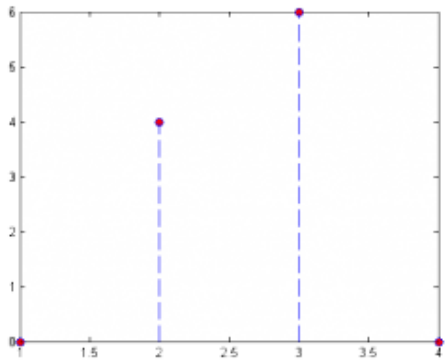


Fig. 1.Datos y=f(x) en espacio unidimensional

Digamos que queremos insertar más puntos de datos entre los puntos x1(=2) y x2(=3), que se aproximan a los valores entre ellos, que oscilan entre f(x)(=4) y f(x2)(=6). Usando la interpolación del vecino más cercano, nuestro resultado se vería así: *[Figura 2]*

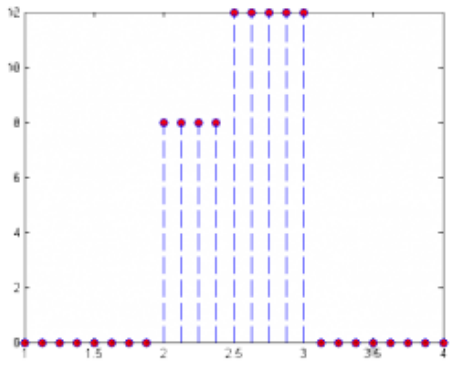


Fig. 2. Uso del vecino más cercano en espacio unidimensional

Podemos ver arriba que, para cada punto de datos, xi, entre nuestros puntos de datos originales, x1 y x2, les asignamos un valor f(xi) basado en cuál de los puntos de datos originales estaba más cerca del eje horizontal.

Ahora, extendiendo esto a 2D, supongamos que queremos cambiar el tamaño de una pequeña imagen de 2×2 píxeles, *[Figura 3]* para que "encaje" en la cuadrícula de imagen de 9×9 pixeles.

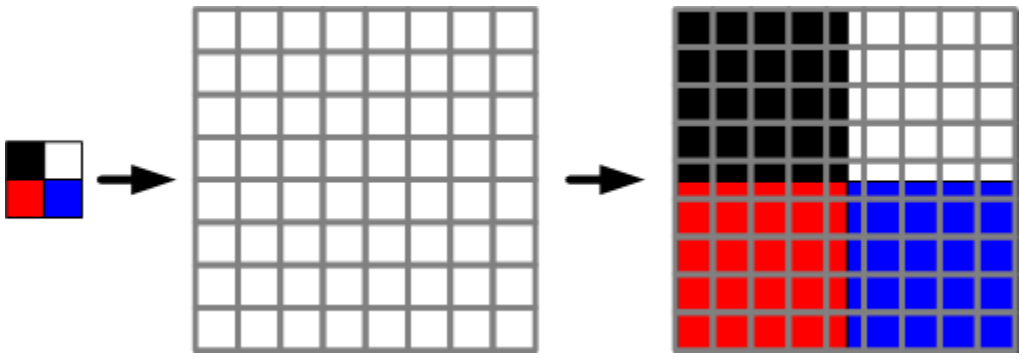


Fig. 3. Ejemplo de muestreo superior de una imagen.

Para visualizar la interpolación del vecino más cercano *[Figura 4]*. Los puntos de datos en el conjunto X representan píxeles de la imagen de original, mientras que los puntos de datos en el conjunto Y representan píxeles en nuestra imagen de salida.

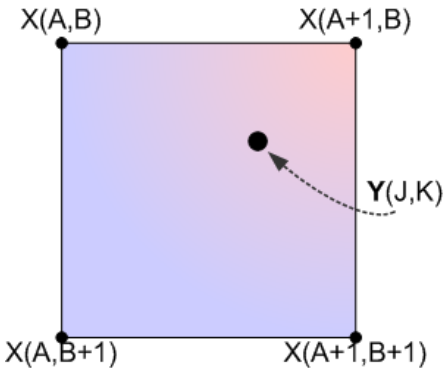


Fig. 4. Sistema Coordinado

Para la práctica, rellenamos la matriz de salida mediante la fórmula matemática *[Formula 1]*

$$i = (i' - 1) \cdot \frac{Size - 1}{Size' - 1} + 1$$

Formula 1

Siendo los siguientes datos:

- *i = posición actual de la matriz.*
- *i'= posición de la nueva matriz.*
- *Size = Tamaño de la imagen principal.*
- *Size' = Tamaño de la nueva imagen.*

Con estos datos podemos rellenar la matriz de salida "imagenSalida" a partir de los pixeles más cercanos, que iremos representando en cada ciclo de pixeles. *Código1* → *[Anexo I]*

```
    imagenSalida(i, j) = imagen(((i - 1)*mediaR) + 1, ((j - 1)*mediaC) + 1);
```

Ejemplos de aplicación.

En nuestro caso vamos a trabajar con 5 imágenes de prueba, cada una con una resolución diferente. *[Figura 5]*

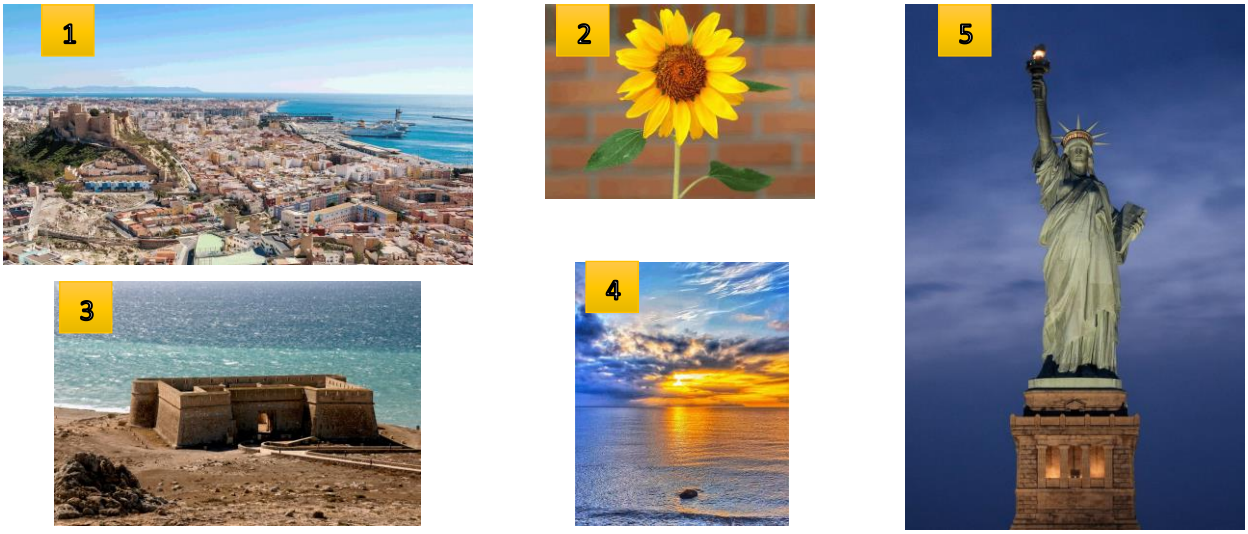
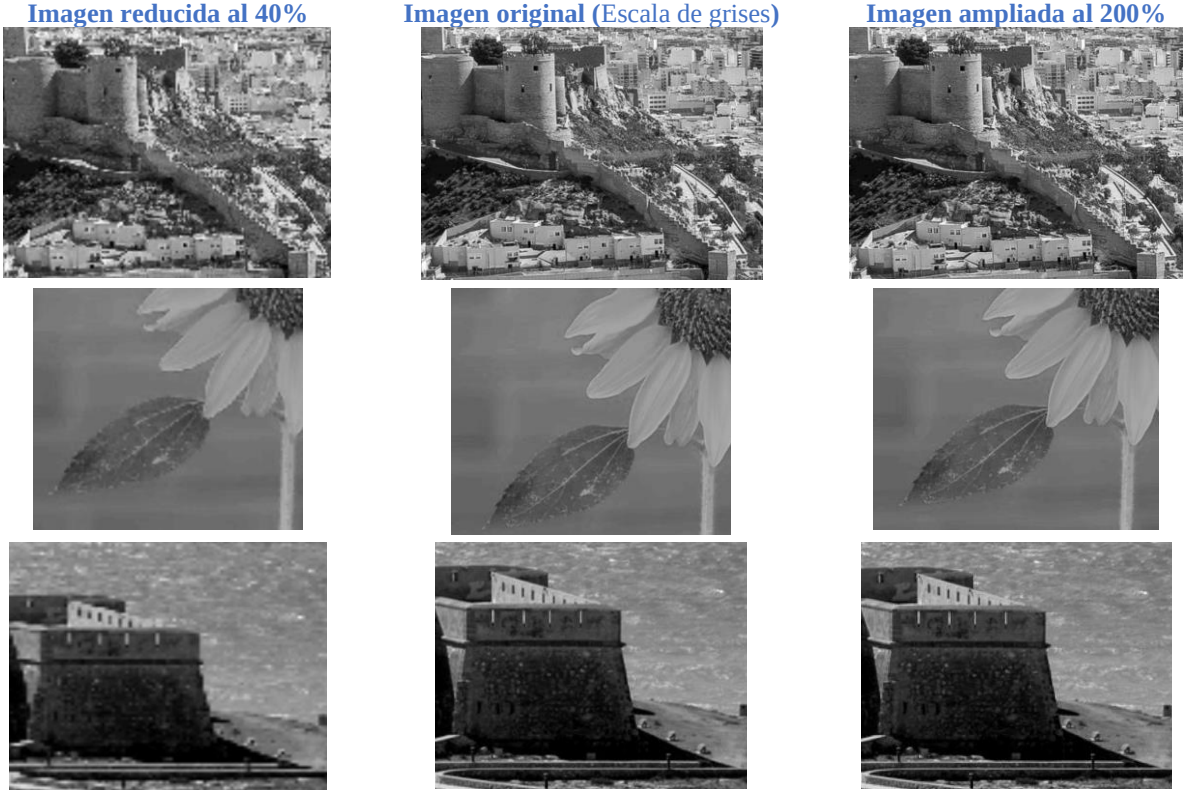


Fig. 5. Imágenes de prueba (1) Alcazaba de Almería y el puerto. (2) Girasol. (3) Castillo de Guardias Viejas (4) Playa en la universidad de Almería (5) Estatua de la libertad.

Para los ejemplos de ejecución, mostraremos solo una parte de la imagen, ya que las dimensionas son muy grandes y solo queremos observar el resultado del algoritmo.

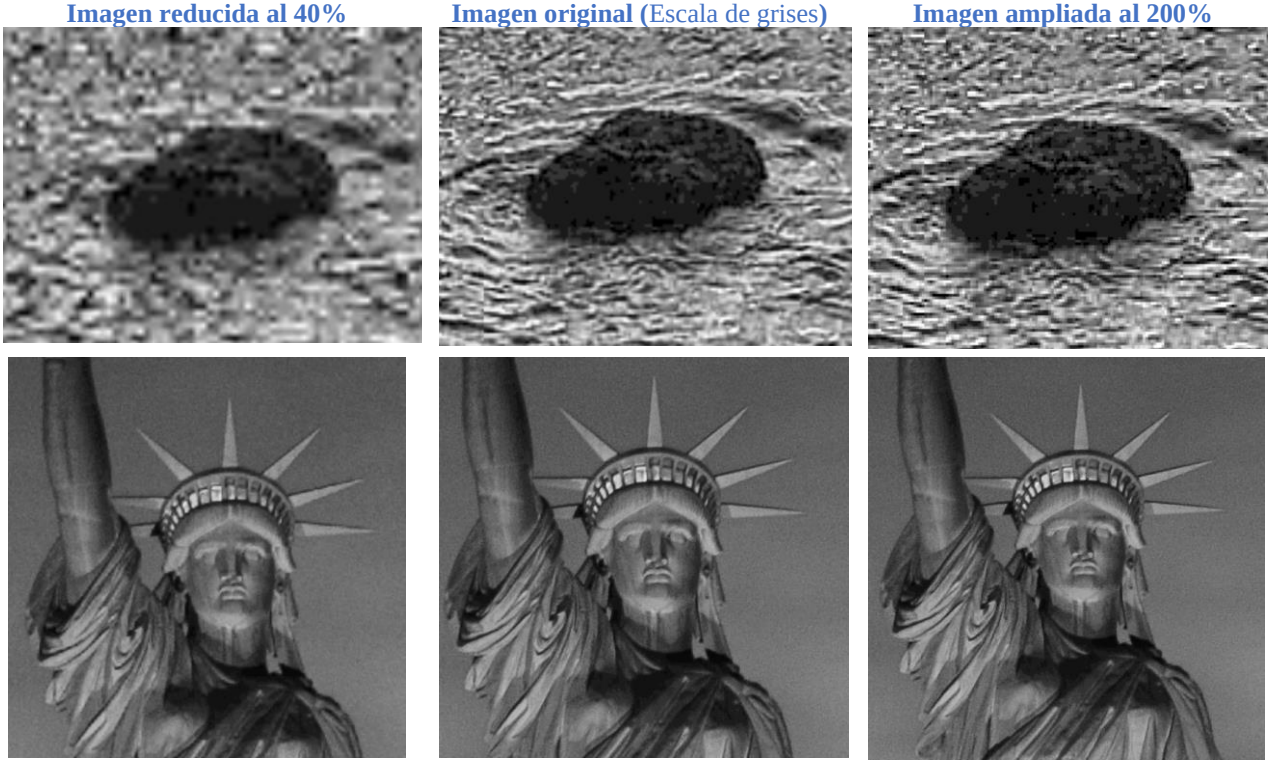
Imagen reducida al 40% | **Imagen original (**Escala de grises**)** | **Imagen ampliada al 200%**



Conclusiones del algoritmo.

El principal inconveniente de este algoritmo es que, a pesar de su velocidad y simplicidad, tiende a generar imágenes de baja calidad. Aunque las imágenes como las fotografías salen "en bloque", aunque con poca o ninguna pérdida notable de nitidez. El punto clave aquí es que este es un algoritmo simple y rápido. A continuación, se incluyen dos ejemplos más que demuestran los inconvenientes de este algoritmo.

Imagen reducida al 40% **Imagen original (**Escala de grises**)** **Imagen ampliada al 200%**



La imagen de la estatua de la libertad, por ejemplo, muestra una imagen ampliada sin ningún cambio en absoluto. Además, debido a la simplicidad de este algoritmo, la operación tarda muy poco tiempo en completarse. Hay que tener en cuenta que, aunque la imagen original y la ampliada se vean iguales, el tamaño entre ambas es el doble.

Como podemos ver en los resultados anteriores para imágenes con menor tamaño, los resultados son de baja calidad. Aunque se mantiene la nitidez de la imagen original, notamos cómo la imagen reducida tiene “irregularidades” (Ejemplo de la playa de la universidad), otras imágenes como (el castillo de guardias viejas) se pueden observar en sus muros bordes dentados, en resumen, reducir la imagen provoca que se veá pixelada y distorsionada si aplicamos este algoritmo.

Esto se conoce como aliasing, y hay varias formas de solucionarlo. Dos de las formas más sencillas son usar un mejor método de interpolación, como los que veremos a continuación.

Método de interpolación bilineal.

La interpolación bilineal también conocido como “filtrado bilineal o mapeo de textura bilineal” es una técnica para calcular los valores de un pixel basada en los pixeles cercanos. La diferencia clave es que utiliza los cuatro vecinos más cercanos.

Usando los cuatro pixeles vecinos más cercanas, la interpolación bilineal asigna el valor de la celda de salida tomando el promedio ponderado. La interpolación bilineal se realiza usando interpolación lineal primero en una dirección y luego nuevamente en la otra dirección. Aunque cada paso es lineal en los valores muestreados y en la posición, la interpolación en su conjunto no es lineal sino más bien cuadrática en la ubicación de la muestra.

Para mostrar cómo funciona, mostraremos primero un ejemplo en un espacio unidimensional. *[Figura 6]*

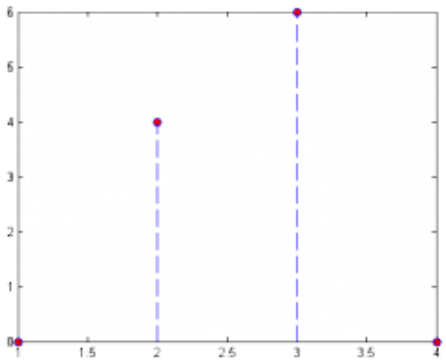


Fig. 6. Datos y=f(x) en espacio unidimensional

Digamos que queremos insertar más puntos de datos entre los puntos x1(=2) y x2(=3), que se aproximan a los valores entre ellos, que oscilan entre f(x)(=4) y f(x2)(=6). Ya hemos visto lo que sucede con la interpolación del vecino más cercano. Por tanto, en este caso haremos que los valores, $f(x_i)$ para cada punto de datos, x_i donde i = 0, 1, 2,…, n-1, entre x_1 y x_2, quepan en una línea, como se muestra a continuación:: *[Figura 7]*

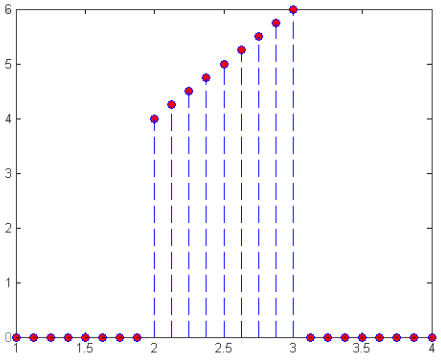


Fig. 7. Y=f(x) usando interpolación lineal.

Cada punto de datos se puede calcular usando la fórmula simple de intercepción de pendiente y=mx+b.

Ahora, extendiendo esto a 2D, supongamos que queremos cambiar el tamaño de una pequeña imagen de 2×2 píxeles, *[Figura 3]* para que "encaje" en la cuadrícula de imagen de 9×9 pixeles.

Para visualizar la interpolación bilineal *[Figura 8]*. Los puntos de datos en el conjunto X representan píxeles de la imagen de original, mientras que los puntos de datos en el conjunto Y representan píxeles en nuestra imagen de salida.

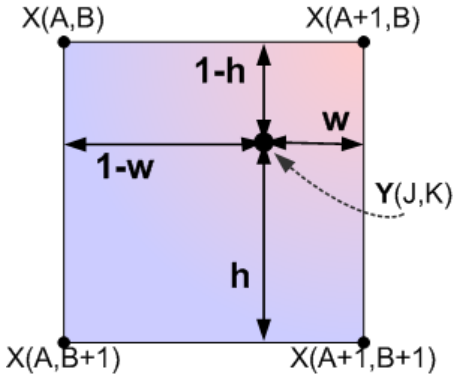


Fig. 8. Sistema coordinado.

De la imagen de arriba *[Figura 8]*, podemos ver ahora que cuando se realiza un muestreo ascendente utilizando la interpolación bilineal, simplemente estamos creando el nuevo píxel en la imagen objetivo a partir de un promedio ponderado de sus 4 píxeles vecinos más cercanos en la imagen de origen. Realizar este proceso es un poco más lento que el algoritmo del vecino más cercano, pero da un resultado mejor.

Con estos datos podemos rellenar la matriz de salida "imagenSalida" a partir de los pixeles más cercanos, que iremos representando en cada ciclo de pixeles. *Código2 → [Anexo I]*

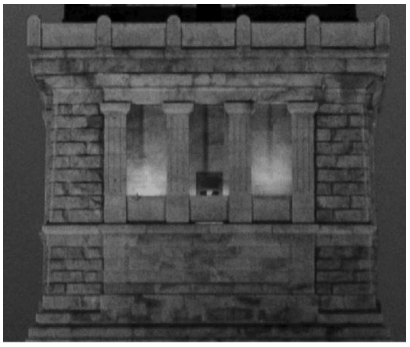
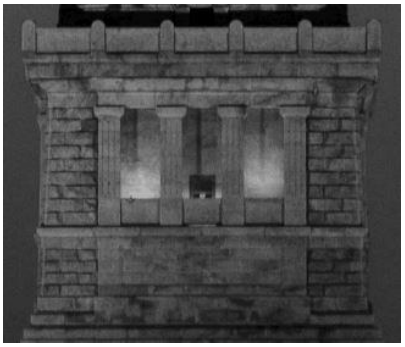
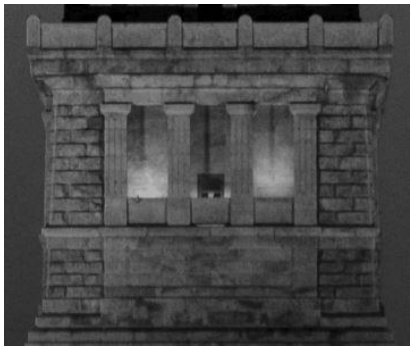
```
imagenSalida(i, j) = (
               imagen(((i - 1)*mediaR) + 1 ,     ((j - 1)*mediaC) + 1) +
               imagen(((i - 1)*mediaR) + 1 + a , ((j - 1)*mediaC)+ 1) +
               imagen(((i - 1)*mediaR) + 1 ,     ((j - 1)*mediaC) + 1 + b) +
               imagen(((i - 1)*mediaR) + 1 + a , ((j - 1)*mediaC) + 1 + b))/4;
```

Se puede observar que las fórmulas de obtención de pixeles son parecidas a las mostradas en el método del vecino más cercano *[Formula 1]*, pero en nuestro caso tenemos que hacer uso de las variables a y b para gestionar los otros tres nuevos pixeles y comprobando que no salimos de los bordes de la matriz origen.

Ejemplos de aplicación.

En nuestro caso vamos a trabajar con las 5 imágenes de prueba que hemos mostrado anteriormente, cada una con una resolución diferente. *[Figura 5]*

Para los ejemplos de ejecución, mostraremos solo una parte de la imagen, ya que las dimensionas son muy grandes y solo queremos observar el resultado del algoritmo.

Imagen reducida al 40%	**Imagen original (**Escala de grises**)**	**Imagen ampliada al 200%**
		
		
		
		
		

Conclusiones del algoritmo.

Para empezar, hay una transición suave entre los distintos valores de los píxeles. Cuando se sube el muestreo por un factor pequeño, es decir, cuando reducimos la imagen original podemos ver que la "pixelación" se reduce y es más suavizada que con el anterior algoritmo del vecino más cercano.

Se puede observar, por ejemplo, que la imagen reducida del girasol sigue estando pixelada respecto a la imagen original. Aunque el detalle del pixelado es menos pronunciado que con el anterior método.

Cuando se aumenta el muestreo por un factor muy grande, obtenemos un gradiente muy suave. Esto se produce a expensas de un algoritmo más complicado y una ligera pérdida de nitidez, como se muestra en las anteriores ejecuciones de imágenes.

Este ejemplo se puede observar claramente en la imagen de la playa de la universidad, la opción ampliada muestra el reflejo del sol en el agua, muchos más suavizado respecto a la imagen orignal.

Método de interpolación bicúbica.

Como hemos realizado con los métodos anteriores, antes de comenzar con la interpolación bicúbica, examinaremos primero como se representa en casos unidimensionales, para poder explicar con mayor detalle cómo funciona este algoritmo.

En el caso de la interpolación vecina más cercana o bilineal, obtenemos una función por partes que es continua durante pequeños intervalos. Para el vecino más cercano, obtenemos "picos" constantes y planos. Con la interpolación bilineal, obtenemos secciones que son lineales en un cierto rango, pero tienen discontinuidades agudas entre estas secciones. Necesitamos buscar una interpolación por partes que sea continua durante todo el intervalo de interpolación.

Primero generamos una sinusoide discreta *[Figura 9],* imaginemos que es la imagen original, en este caso reducimos los valores para que se más visible el ejemplo *[Figura 10].*

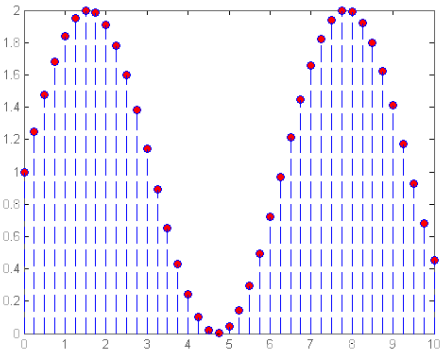


Fig. 9. Sinusoide discreta

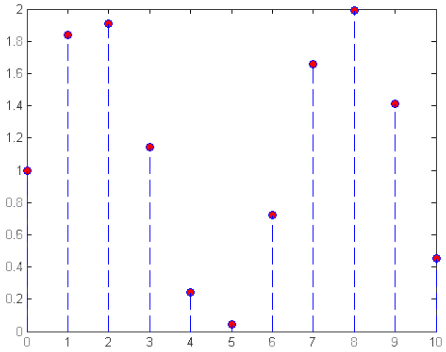


Fig. 10. Sinusoide reducida

Si aplicásemos el método del vecino más cercano en el espacio unidimensional, el resultado sería el mostrado en la *[Figura 11],* Por otra parte, si en este caso, fuese el método bilineal el algoritmo aplicado, el resultado mostrado sería el de la *[Figura 12]*

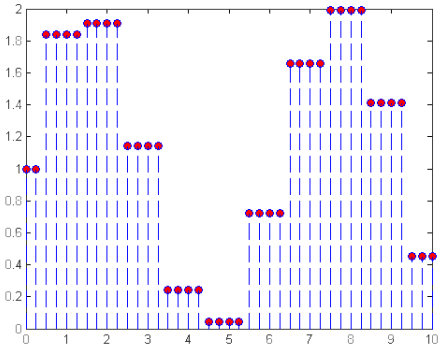


Fig. 11. Método vecino más cercano aplicado

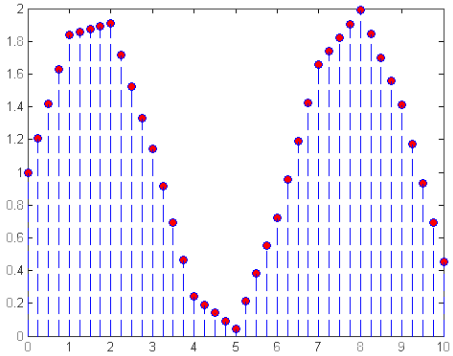


Fig. 12. Método bilineal aplicado

Se puede observar en el método del vecino más cercano *[Figura 11],* cómo los valores interpolados son constantes (planos) sobre esta función por partes. Hay discontinuidades agudas entre cada "pieza", y la trama anterior apenas se parece a una sinusoide.

En la *[Figura 12]* es mejor que cuando usamos la interpolación de vecino más cercano, ya que la gráfica anterior al menos parece una sinusoide en cierta medida. Además, cada "pieza" es lineal, en lugar de constante, por lo que hace un mejor trabajo de aproximación de la sinusoide original. Aunque ttodavía se puede observar que cada "pieza" tiene una discontinuidad aguda. Para solucionar este problema se utiliza el método de interpolación bicúbica. *[Figura 13]*

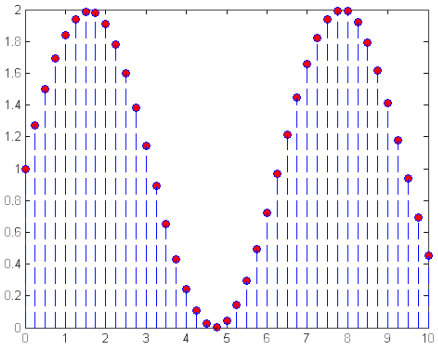


Fig. 13. Método de interpolación bicúbica

Ahora, en este caso *[Figura 13]* se puede observar cómo la gráfica anterior se parece bastante bien a la sinusoide original. *[Figura 9]* Además, no tiene discontinuidades agudas, como fue el caso con los dos métodos de interpolación anteriores. La interpolación cúbica es esencialmente otra forma de ajuste de curva, algo similar a la interpolación lineal.

La interpolación bicúbica se elige a menudo más de interpolación bilineal o al vecino más cercano en la imagen de remuestreo , cuando la velocidad no es un problema. En contraste con la interpolación bilineal, que sólo toma 4 pixeles (2 x 2) en cuenta, interpolación bicúbica considera 16 píxeles (4 x 4). Las imágenes de la nueva muestra con interpolación bicúbica son más suaves.

Con estos datos podemos rellenar la matriz de salida “imagenSalida” a partir de los pixeles más cercanos, que iremos representando en cada ciclo de pixeles. *Código3 → [Anexo I]*

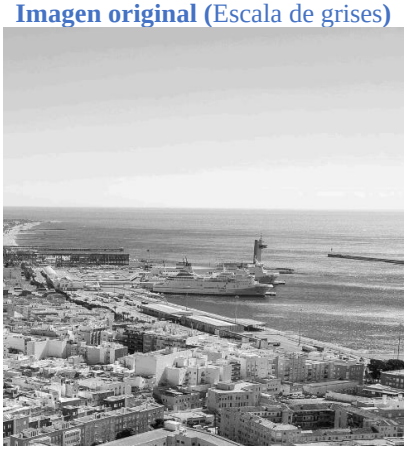
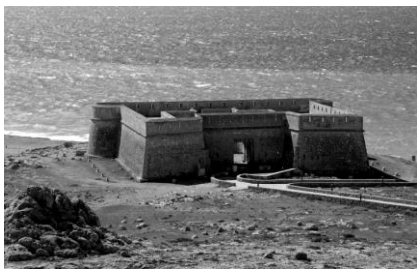
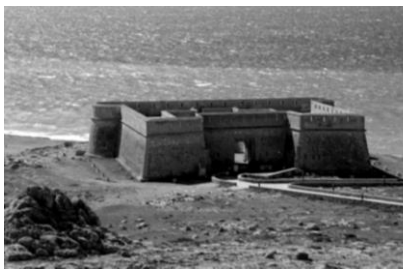
```
imagenSalida(i, j) = (
            imagen(((i - 1)*mediaR) + 1,       ((j - 1)*mediaC) + 1) +
            imagen(((i - 1)*mediaR) + 1 + ax,  ((j - 1)*mediaC) + 1) +
            imagen(((i - 1)*mediaR) + 1,       ((j - 1)*mediaC) + 1 + ay) +
            imagen(((i - 1)*mediaR) + 1 + ax,  ((j - 1)*mediaC) + 1 + ay) +
            imagen(((i - 1)*mediaR) + 1,       ((j - 1)*mediaC) + by) +
            imagen(((i - 1)*mediaR) + 1 + ax,  ((j - 1)*mediaC) + 1 + by) +
            imagen(((i - 1)*mediaR) + 1 + bx,  ((j - 1)*mediaC) + 1 + by) +
            imagen(((i - 1)*mediaR) + 1 + bx,  ((j - 1)*mediaC) + 1 + ay) +
            imagen(((i - 1)*mediaR) + 1 + bx,  ((j - 1)*mediaC) + 1) +
            imagen(((i - 1)*mediaR) + 1 + cx,  ((j - 1)*mediaC) + 1) +
            imagen(((i - 1)*mediaR) + 1 + cx,  ((j - 1)*mediaC) + 1 + ay) +
            imagen(((i - 1)*mediaR) + 1,       ((j - 1)*mediaC) + cy) +
            imagen(((i - 1)*mediaR) + 1 + cx,  ((j - 1)*mediaC) + 1 + by) +
            imagen(((i - 1)*mediaR) + 1 + ax,  ((j - 1)*mediaC) + 1 + cy) +
            imagen(((i - 1)*mediaR) + 1 + bx,  ((j - 1)*mediaC) + 1 + cy) +
            imagen(((i - 1)*mediaR) + 1 + cx,  ((j - 1)*mediaC) + 1 + cy))/16;
```

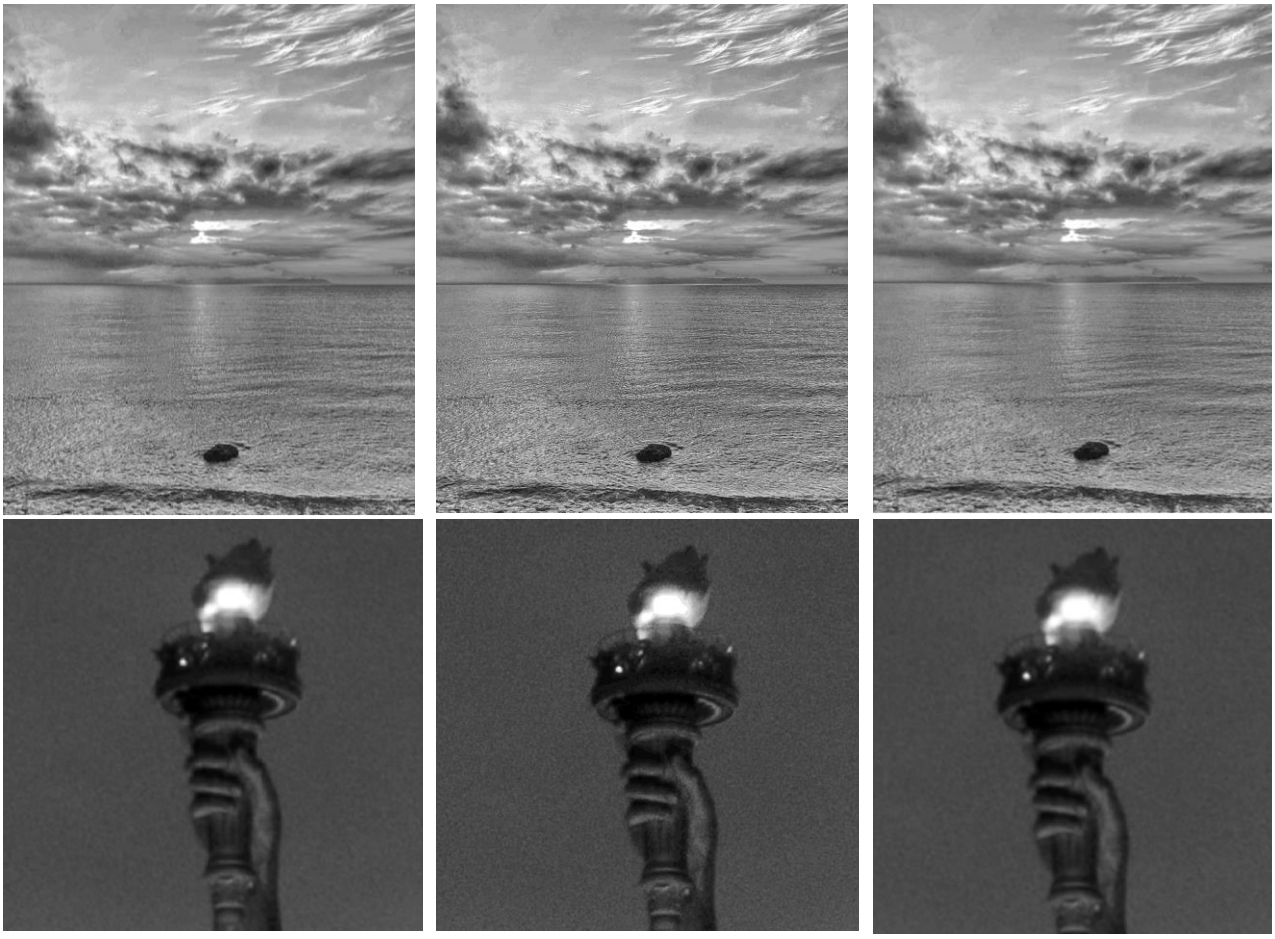
Se puede observar que las fórmulas de obtención de pixeles son parecidas a las mostradas en el método del vecino más cercano *[Formula 1]*, pero en nuestro caso tenemos que hacer uso de las variables a,b,c para gestionar los otros quince nuevos pixeles y comprobando que no salimos de los bordes de la matriz origen.

Ejemplos de aplicación.

En nuestro caso vamos a trabajar con las 5 imágenes de prueba que hemos mostrado anteriormente, cada una con una resolución diferente. *[Figura 5]*

Para los ejemplos de ejecución, mostraremos solo una parte de la imagen, ya que las dimensionas son muy grandes y solo queremos observar el resultado del algoritmo.

Imagen reducida al 40%	**Imagen original (**Escala de grises**)**	**Imagen ampliada al 200%**
		
		
		



Conclusiones del algoritmo.

Para empezar, hay una transición mucho más suave entre los distintos valores de los píxeles en comparación con los anteriores métodos. Cuando se sube el muestreo por un factor pequeño, es decir, cuando reducimos la imagen original podemos ver que la "pixelación" se reduce aun más y es más suavizada que con los anteriores algoritmos.

Se puede observar, por ejemplo, que la imagen reducida del girasol ya no se encuentra "pixelada" respecto a la imagen original.

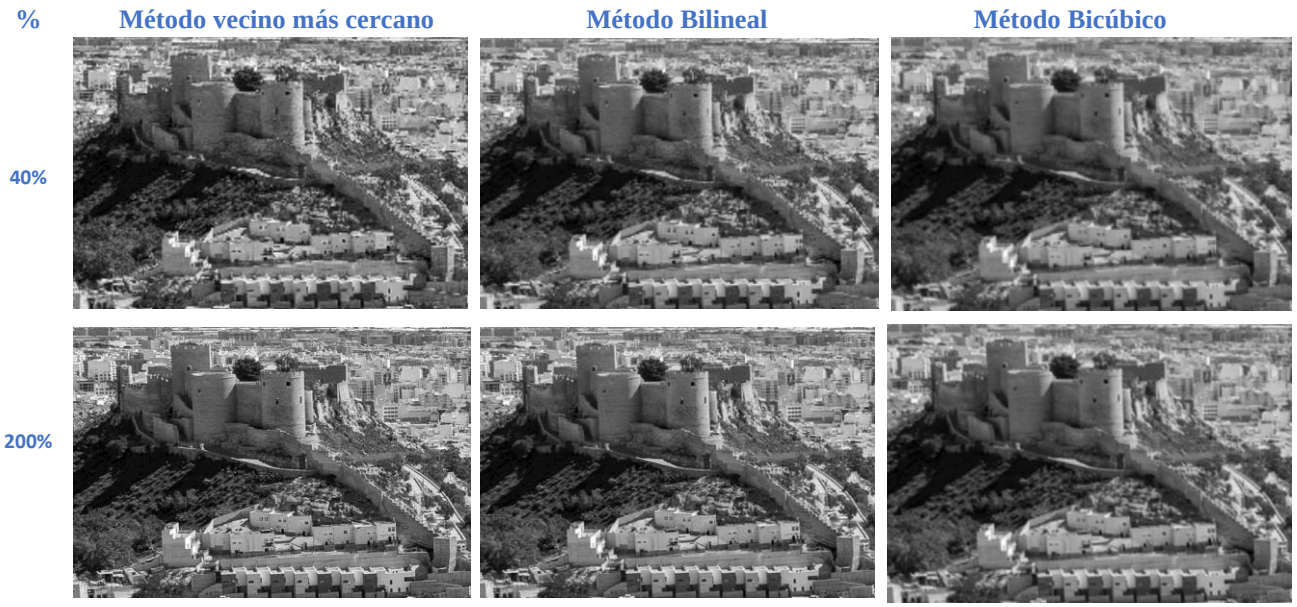
Cuando se aumenta el muestreo por un factor muy grande, obtenemos una nitidez más suave que la imagen original.

Este ejemplo se puede observar claramente en la imagen de la estatua de la liberta, la opción ampliada muestra el con mejor detalle los dedos que sujetan la antorcha, la nitidez se ha reducido respecto a la imagen original.

Comparativas

La interpolación bilineal es una técnica relativamente simple, no es mucho más complicada del término "vecino más cercano", donde la interpolación de estos pixeles que estan vacíos se llenan con sólo copiar los píxeles adyacentes. Por cada píxel "perdido" (los píxeles que tienen que ser creados para cubrir la imagen) el método bilineal toma los cuatro puntos que están más próximos a las esquinas en diagonal, y sus valores promedios para producir el píxel medio. La interpolación bicúbica, en cambio, tiene no sólo los cuatro píxeles diagonales más cercanos, sino sus puntos más cercanos también, para un total de 16 pixeles.

Veamos a continuación una comparativa más visual entre ambos algoritmos.



Se puede ver qué las imágenes de método con interpolación bicúbica se ven mejor que con los otros métodos. Aunque también podemos ver mayor la **halación** en el método bicúbico. Al igual que la interpolación vecina más cercana introduce **aliasing**, o la interpolación bilineal introduce **borrosidad**, este método también introduce un nuevo artefacto. Esto esencialmente causa "halos" alrededor de partes de una imagen, o un brillo excesivo de píxeles seleccionados cerca de los bordes de las formas. Por eso se puede observar que las sombras de las imágenes son menos intensas en este caso.

Debido a que cualquier método de interpolación se basa en la invención de nuevos datos, cualquier imagen redimensionada es igualmente fiel entre las técnicas de interpolación.

La diferencia principalmente reside en cómo la imagen es percibida por el humano, y porque la interpolación bicúbica hace uso de más datos, los resultados son generalmente más suaves. La interpolación bicúbica crea curvas más suaves que la interpolación bilineal y presenta un menor número de "artefactos", o píxeles que se destacan como visible deterioro de la calidad aparente de la imagen.

Cuando necesitas aumentar el tamaño de una imagen y el tiempo no es importante, la interpolación bicúbica proporciona los resultados suaves que son percibidos como de mayor calidad . Sin embargo, el hecho de que utilizan píxeles adicionales puede ser una desventaja cuando la imagen está siendo reducida en lugar de agrandada, porque también significa que más píxeles se descartan o se cambian. En estos casos, el número comparativamente menor de pixeles utilizados por el método bilineal puede producir resultados que son más agradables a la vista y con menos artefactos.

Otros métodos

Antes de concluir con el proyecto, debo señalar que hay muchos otros métodos de interpolación de imágenes como Lanczos, Stair-Interpolation, S-Spline, Genuine Fractals, etc.

Muchos de estos algoritmos se utilizan en productos comerciales como cámaras digitales de software de procesamiento de imágenes, incluso emuladores de consola. A continuación os mostramos una breve información sobre algunos otros métodos:

Lanczo se trata de una extensión al dominio 2D de esta misma técnica que se aplica de forma habitual para obtener más datos a partir del muestreo de una señal 1D (en una dimensión). En una dimensión, el valor de cada nuevo punto se calcula como una suma ponderada de los valores de los puntos originales (muestras) próximos a él. La influencia o peso que una muestra tiene en el cálculo de un punto vecino x viene dado por el núcleo de Lanczos.

Stair-Interpolation es un método general para la interpolación de los píxeles después de la ampliación de una imagen. La idea clave es interpolar varias veces en pequeños incrementos usando cualquier algoritmo de interpolación que es mejor que la interpolación del vecino más próximo , como la interpolación bilineal , y la interpolación bicúbica.

S-plines es una forma de interpolación donde el interpolante es un tipo especial de polinomio por partes llamado spline. La interpolación de splines a menudo se prefiere a la interpolación polinómica porque el error de interpolación puede hacerse pequeño incluso cuando se usan polinomios de bajo grado para la spline. La interpolación de spline evita el problema del fenómeno de Runge, en el que puede producirse oscilación entre puntos al interpolar utilizando polinomios de alto grado.

Genuine Fractals hace muy poco más que un simple remuestreo o cambio de tamaño en Photoshop para pequeñas cantidades de interpolación. Genuine Fractals renderiza las líneas de forma más nítida que un remuestreo de Photoshop, pero no mejora la resolución. No reemplaza las texturas faltantes, pero hace un trabajo increíble en líneas.

Esto significa que a medida que se amplía a cantidades increíbles, es posible que tenga una imagen que comienza a aparecer como caricatura, como si mirara la salida demasiado de cerca, ya que las líneas son nítidas pero la textura no está allí.

Anexo

Código implementado:

```
//IMPORTACION DE ENCABEZADOS
#include <iostream>
#include <C_Arguments.hpp>
#include <C_File.hpp>
#include <C_General.hpp>
#include <C_Image.hpp>
#include <C_Matrix.hpp>
#include <C_Memory.hpp>
#include <C_String.hpp>
#include <C_Trace.hpp>

//DEFICIONES PARA COLORES POR PANTALLA
#include <windows.h>
#define Color SetConsoleTextAttribute(hConsole, K);
#define Celeste K=11;
#define Rojo K=12;
#define Amarillo K=14;
#define Verde K=10;
#define Blanco K=7;

//GESTION DE COLORES POR PANTALLA
    HANDLE hConsole;
    int K;

//int Test(int argc, char **argv);

/*****************************************
*                                        *
* DECLARACION DE METODOS DE REDIMENSIONADO *
*                                        *
*****************************************/
//METODO VECINO MS CERCANO
void Redimensionado_Vecino_Cercano(string nombre_Imagen, C_Image imagen, int ImgAncho_Red, int
ImgAlto_Red, string red){
    hConsole = GetStdHandle(STD_OUTPUT_HANDLE);

    cout << "1 # Aplicando Metodo Vecino mas cercano..." << endl;

    //Imagen de salida
    C_Image imagenSalida(1, ImgAlto_Red, 1, ImgAncho_Red);

    //Calcula el alto y ancho de las imagenes
    double x1 = imagen.LastRow() - 1;
    double y1 = imagen.LastCol() - 1;
    double x2 = imagenSalida.LastRow() - 1;
    double y2 = imagenSalida.LastCol() - 1;

    //Calcular la media de las imagenes
    double mediaR = x1 / x2;
    double mediaC = y1 / y2;

    for (int i = 1; i <= ImgAlto_Red; i++){
        for (int j = 1; j <= ImgAncho_Red; j++){
            //Generar salida con el vecino cercano
            imagenSalida(i, j) = imagen(((i - 1)*mediaR) + 1, ((j - 1)*mediaC) + 1);
        }
    }
    //Guardado
    string nombreImagenSalida = "Red_VC_"+red+"_"+nombre_Imagen;
    const char * salida = nombreImagenSalida.c_str();
    imagenSalida.WriteBMP(salida);
    Verde;Color; cout << "Metodo realizado con exito" << endl;
    Blanco;Color;cout << "La imagen se guardo como: "<< nombreImagenSalida << endl;
    cout << " " << endl;
    void Pause();
}
```

```
//METODO  BILINEAL
void Redimensionado_Bilineal(string nombre_Imagen, C_Image imagen, int ImgAncho_Red, int
ImgAlto_Red, string red){
    hConsole = GetStdHandle(STD_OUTPUT_HANDLE);

    cout << "2 # Aplicando Metodo Bilineal..." << endl;

    //Imagen de salida
    C_Image imagenSalida(1, ImgAlto_Red, 1, ImgAncho_Red);

    //Calcula el alto y ancho de las imagenes
    double x1 = imagen.LastRow() - 1;
    double y1 = imagen.LastCol() - 1;
    double x2 = imagenSalida.LastRow() - 1;
    double y2 = imagenSalida.LastCol() - 1;

    //Calcular la media de las imagenes
    double mediaR = x1 / x2;
    double mediaC = y1 / y2;

    int a, b;

    for (int i = 1; i <= ImgAlto_Red; i++){
        for (int j = 1; j <= ImgAncho_Red; j++){
            a = 1;
            b = 1;

            //Comprobar limites
            if (i == ImgAlto_Red){ a = 0; }
            if (j == ImgAncho_Red){ b = 0;}

            //Generar salida con los 4 vecinos cercanos
            imagenSalida(i, j) = (
                            imagen(((i - 1)*mediaR) + 1 ,     ((j - 1)*mediaC) + 1) +
                            imagen(((i - 1)*mediaR) + 1 + a , ((j - 1)*mediaC)+ 1) +
                            imagen(((i - 1)*mediaR) + 1 ,     ((j - 1)*mediaC) + 1 + b) +
                            imagen(((i - 1)*mediaR) + 1 + a , ((j - 1)*mediaC) + 1 + b))/4;
        }
    }

    //Guardado
    string nombreImagenSalida = "Red_BL_"+red+"_"+nombre_Imagen;
    const char * salida = nombreImagenSalida.c_str();
    imagenSalida.WriteBMP(salida);
    Verde;Color; cout << "Metodo realizado con exito" << endl;
    Blanco;Color;cout << "La imagen se guardo como: "<< nombreImagenSalida << endl;
    cout << " " << endl;
    void Pause();
};

//METODO  BICUBICO
void Redimensionado_Bicubico(string nombre_Imagen, C_Image imagen, int ImgAncho_Red, int
ImgAlto_Red, string red){
    hConsole = GetStdHandle(STD_OUTPUT_HANDLE);

    cout << "3 # Aplicando Metodo Bicubico..." << endl;

    //Imagen de salida
    C_Image imagenSalida(1, ImgAlto_Red, 1, ImgAncho_Red);

    //Calcula el alto y ancho de las imagenes
    double x1 = imagen.LastRow() - 1;
    double x2 = imagenSalida.LastRow() - 1;
    double y1 = imagen.LastCol() - 1;
    double y2 = imagenSalida.LastCol() - 1;

    //Calcular la media de las imagenes
    double mediaR = x1 / x2;
    double mediaC = y1 / y2;

    int ax, ay, bx, by, cx, cy, limiteR, limiteC;
```

```
    for (int i = 1; i <= ImgAlto_Red; i++){
        for (int j = 1; j <= ImgAncho_Red; j++){
            ax = 1;
            ay = 1;
            bx = 2;
            by = 2;
            cx = 3;
            cy = 3;
            limiteR = ((i - 1)*mediaR)+1;
            limiteC = ((j - 1)*mediaC)+1;

            //Comprobar limites
            if (i == ImgAlto_Red || limiteR == x1){ax = 0; bx = 0; cx = 0;}
            if (j == ImgAncho_Red || limiteC == y1){ay = 0; by = 0; cy = 0;}
            if (i == ImgAlto_Red-1 || limiteR == x1-1){bx = 0; cx = 0;}
            if (j == ImgAncho_Red-1 || limiteC == y1-1){by = 0; cy = 0;}
            if (i == ImgAlto_Red-2 || limiteR == x1-2){cx = 0;}
            if (j == ImgAncho_Red-2 || limiteC == y1-2){cy = 0;}

            //Generar salida con los 16 vecinos cercanos
            imagenSalida(i, j) = (
                                    imagen(((i - 1)*mediaR) + 1,      ((j - 1)*mediaC) + 1) +
                                    imagen(((i - 1)*mediaR) + 1 + ax, ((j - 1)*mediaC) + 1) +
                                    imagen(((i - 1)*mediaR) + 1,      ((j - 1)*mediaC) + 1 + ay) +
                                    imagen(((i - 1)*mediaR) + 1 + ax, ((j - 1)*mediaC) + 1 + ay) +
                                    imagen(((i - 1)*mediaR) + 1,      ((j - 1)*mediaC) + by) +
                                    imagen(((i - 1)*mediaR) + 1 + ax, ((j - 1)*mediaC) + 1 + by) +
                                    imagen(((i - 1)*mediaR) + 1 + bx, ((j - 1)*mediaC) + 1 + by) +
                                    imagen(((i - 1)*mediaR) + 1 + bx, ((j - 1)*mediaC) + 1 + ay) +
                                    imagen(((i - 1)*mediaR) + 1 + bx, ((j - 1)*mediaC) + 1) +
                                    imagen(((i - 1)*mediaR) + 1 + cx, ((j - 1)*mediaC) + 1) +
                                    imagen(((i - 1)*mediaR) + 1 + cx, ((j - 1)*mediaC) + 1 + ay) +
                                    imagen(((i - 1)*mediaR) + 1,      ((j - 1)*mediaC) + cy) +
                                    imagen(((i - 1)*mediaR) + 1 + cx, ((j - 1)*mediaC) + 1 + by) +
                                    imagen(((i - 1)*mediaR) + 1 + ax, ((j - 1)*mediaC) + 1 + cy) +
                                    imagen(((i - 1)*mediaR) + 1 + bx, ((j - 1)*mediaC) + 1 + cy) +
                                    imagen(((i - 1)*mediaR) + 1 + cx, ((j - 1)*mediaC) + 1 + cy)
                                  )/16;
        }
    }
    //Guardado
    string nombreImagenSalida = "Red_BC_"+red+"_"+nombre_Imagen;
    const char * salida = nombreImagenSalida.c_str();
    imagenSalida.WriteBMP(salida);
    Verde;Color; cout << "Metodo realizado con exito" << endl;
    Blanco;Color;cout << "La imagen se guardo como: "<< nombreImagenSalida << endl;
    cout << " " << endl;
    void Pause();
}

//METODOS POR INVESTIGAR Y REALIZAR
//void Redimensionado_Escalera(string nombre_Imagen, C_Image imagen, int ImgAncho, int ImgAlto);
//void Redimensionado_SPline(string nombre_Imagen, C_Image imagen, int ImgAncho, int ImgAlto);
//void Redimensionado_Lanczos(string nombre_Imagen, C_Image imagen, int ImgAncho, int ImgAlto);
//void Redimensionado_Genuine_Fractals(string nombre_Imagen, C_Image imagen, int ImgAncho, int);

/*****************************************
*                                        *
*       PROCESOS EXTRAS DE EJECUCIм       *
*                                        *
*****************************************/

//METODO CONVERTIR A ESCALA DE GRISES
void Guardar_Gris(string nombre_Imagen, C_Image imagen){

    cout <<"Convirtiendo imagen a escala de grises..." <<endl;
    string nombreImagenSalida = "Gris "+nombre Imagen;
    const char * salida = nombreImagenSalida.c_str();
    imagen.WriteBMP(salida);
}
```

```
/*****************************************
*                                        *
*           EJECUCIм PRINCIPAL           *
*                                        *
*****************************************/
//Main
int main(){

    hConsole = GetStdHandle(STD_OUTPUT_HANDLE);

    //DECLARACION DE VARIABLES
    int proceso;                //Proceso de redimensionado a realizar
    int anchoImg;               //Ancho de la imagen a procesar
    int altoImg;                //Alto de la imagen a procesar
    int ImgAncho_Red;           //Ancho del resultado de la imagen redimensionada
    int ImgAlto_Red;            //Alto del resultado de la imagen redimensionada

    C_Image imagen;             //Imagen a modificar

    double redimensionado;      //Tama□e redimensionado

    string nombre_Imagen;       //Nombre de la imagen a cargar
    const char * cadena;        //Nombre de la imagen convertido a cadena

    //PROCESO DE CARGA DE IMAGEN A REDIMENSIONAR
    Celeste;Color;
    cout <<"_____________________________________________________"<< endl;
    cout <<"|                                                   |"<< endl;
    cout <<"|      PRACTICA CON REDIMENSIONADO DE IMAGENES      |"<< endl;
    cout <<"|                                                   |"<< endl;
    cout <<"|                     Alberto Angel Fuentes Funes   |"<< endl;
    cout <<"|___________________________________________________|"<< endl;
    do{
        do{
            Blanco;Color;
            cout << " " << endl;
            cout << "Introduce el nombre de la imagen o (0 para salir): ";
            Amarillo;Color;
            cin >> nombre_Imagen;
            Blanco;Color;
            if(nombre_Imagen == "0") return 0;
            cadena = nombre_Imagen.c_str();

            if (!C_FileExists(cadena)){
            //Error 1 --> Si la imagen no ha sido encontrada, vuelve a buscar nueva imagen
                Rojo;Color;
                cout <<"[ERROR][1] La imagen: '"<< cadena <<"' no se ha podido encontrar."<<endl;
                Blanco;Color;
            }

            } while (!C_FileExists(cadena));

        //Lectura de la imagen BMP
        imagen.ReadBMP(cadena);

        //Convertimos la imagen a escala de grises y guardarla, si es necesario.
        imagen.Grey();

        //Comrpobar si ya tenemos la imagen en gris
        string nombreImagenSalida = "Gris_"+nombre_Imagen;
        const char * salida = nombreImagenSalida.c_str();
        if (C_FileExists(salida))
              cout <<"Cargando imagen..." <<endl;
        else {
            Guardar_Gris(nombre_Imagen, imagen);
        }

        //Obtener ancho y alto de la imagen
        anchoImg = imagen.LastCol();
        altoImg = imagen.LastRow();
```

```
        //Obtener el redimensionado
        cout << " " << endl;
        cout << "Introduce el porcentaje de redimensionado: ";
        Amarillo;Color;
        cin >> redimensionado;
        Blanco;Color;
        std::string red = std::to_string(static_cast<long long>(redimensionado));

        redimensionado = redimensionado/100; //Calculo del redimensionado
        ImgAncho_Red=anchoImg*redimensionado;//Calculo del nuevoAncho de la imagen redimensionar
        ImgAlto_Red=altoImg*redimensionado;  //Calculo del nuevoAlto de la imagen redimensionar

        Celeste;Color;
        cout << " " << endl;
        cout << "---------DATOS INTRODUCCIDOS---------"  << endl;Blanco;Color;
        cout << "    Su imagen: "<< nombre_Imagen        << endl;  //Salida por pantalla de los
        cout << "    Redimensionado: x"<< redimensionado << endl;  //datos cargados
        cout << "    Ancho: "<< anchoImg<<" pixeles"     << endl;
        cout << "    Alto: "<< altoImg<<" pixeles"       << endl;Celeste;Color;
        cout << "-------------------------------------"  << endl;

        do{
            cout << " " << endl;
            cout << "-----Introduce un metodo-----"<< endl; Blanco;Color;
            cout << "1) Metodo Vecino mas cercano" << endl; //Menu de ejecuciones de metodos
            cout << "2) Metodo Bilineal"           << endl;
            cout << "3) Metodo Bicubico"           << endl;
            cout << "4) Todos los metodos"         << endl;Celeste;Color;
            cout << "----------------------------"<< endl;Blanco;Color;
            cout << "5) Salir"                     << endl;Celeste;Color;
            cout << "----------------------------"<< endl;
            Amarillo;Color;
            cin >> proceso;
            Blanco;Color;
        } while (proceso < 0 && proceso >= 10);

        switch (proceso){
            case 1:
                cout << " " << endl;
                Redimensionado_Vecino_Cercano(nombre_Imagen,imagen,ImgAncho_Red,ImgAlto_Red,red);
            break;
            case 2:
                cout << " " << endl;
                Redimensionado_Bilineal(nombre_Imagen, imagen, ImgAncho_Red, ImgAlto_Red, red);
            break;
            case 3:
                cout << " " << endl;
                Redimensionado_Bicubico(nombre_Imagen, imagen, ImgAncho_Red, ImgAlto_Red, red);
            break;
            case 4:
                cout << " " << endl;
                Celeste;Color;cout << "APLICANDO TODOS LOS METODOS.." << endl;Blanco;Color;
                Redimensionado_Vecino_Cercano(nombre_Imagen,imagen,ImgAncho_Red,ImgAlto_Red,red);
                Redimensionado_Bilineal(nombre_Imagen,imagen,ImgAncho_Red,ImgAlto_Red,red);
                Redimensionado_Bicubico(nombre_Imagen,imagen,ImgAncho_Red,ImgAlto_Red,red);
            break;
            case 5:
                return 0;
            break;
        };

    } while (true);
    return 0;
}
```

Este código se encuentra subido en https://github.com/21albertoff/TDI por si desea probar el programa.

Bibliografía

[1]"interpolación spline - Spline interpolation - qwe.wiki", *Es.qwe.wiki*, 2020. [Online]. Disponible en: https://es.qwe.wiki/wiki/Spline_interpolation. [Accedido: 19- May- 2020].

[2]U. imagen., "Un método rápido para escalar imágenes en color. Lección: escala de imagen.", *Beasthackerz.ru*, 2020. [Online]. Disponible en: https://beasthackerz.ru/es/wi-fi-lokalnaya-set/bystryi-metod-masshtabirovaniya-cvetnyh-izobrazhenii-urok.html. [Accedido: 19- May- 2020].

[3]"Interpolación (fotografía)", *Es.wikipedia.org*, 2020. [Online]. Disponible en: https://es.wikipedia.org/wiki/Interpolaci%C3%B3n_(fotograf%C3%ADa). [Accedido: 19- May- 2020].

[4]"Stair Interpolation (SI)", *Fredmiranda.com*, 2020. [Online]. Disponible en: http://www.fredmiranda.com/SI/. [Accedido: 19- May- 2020].

[5]"Algoritmo de Lanczos", *Es.wikipedia.org*, 2020. [Online]. Disponible en: https://es.wikipedia.org/wiki/Algoritmo_de_Lanczos. [Accedido: 19- May- 2020].

[6]"Understanding Digital Image Interpolation", *Cambridgeincolour.com*, 2020. [Online]. Disponible en: https://www.cambridgeincolour.com/tutorials/image-interpolation.htm. [Accedido: 19- May- 2020].

[7]"Redimensionar imágenes con c#.", *Netveloper.com*, 2020. [Online]. Disponible en: https://www.netveloper.com/redimensionar-imagenes-con-csharp. [Accedido: 19- May- 2020].

[8]"codigo C interpolaciones y ejemplos", *Forum.bennugd.org*, 2020. [Online]. Disponible en: http://forum.bennugd.org/index.php?topic=3457.0. [Accedido: 19- May- 2020].

[9]"Bilinear interpolation - Rosetta Code", *Rosettacode.org*, 2020. [Online]. Disponible en: https://rosettacode.org/wiki/Bilinear_interpolation#C. [Accedido: 19- May- 2020].

[10]"Interpolation (Bilinear Filtering)", *Scratchapixel.com*, 2020. [Online]. Disponible en: https://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/interpolation/bilinear-filtering. [Accedido: 19- May- 2020].

[11]"C/C++ Function to Compute the Bilinear Interpolation", *Computing & Technology, The Ultimate Computer Technology Blog and The Knowledgebase of Computing*, 2020. [Online]. Disponible en: https://helloacm.com/cc-function-to-compute-the-bilinear-interpolation/. [Accedido: 19- May- 2020].

[12]*Oa.upm.es*, 2020. [Online]. Disponible en: http://oa.upm.es/1282/1/JAVIER_VIDAL_VALENZUELA.pdf. [Accedido: 19- May- 2020].

[13]"article_lr.pdf", *Docs.google.com*, 2020. [Online]. Disponible en: https://docs.google.com/viewerng/viewer?url=http://www.ipol.im/pub/art/2011/g_lmii//article_lr.pdf. [Accedido: 19- May- 2020].

[14]S. G_, "Zoom An Image With Different Interpolation Types", *Codeproject.com*, 2020. [Online]. Disponible en: https://www.codeproject.com/Articles/236394/Bi-Cubic-and-Bi-Linear-Interpolation-with-GLSL. [Accedido: 19- May- 2020].

[15]G. Estévez, "/*Prog*/ Delphi-Neftalí /*finProg*/ » Redimensionar una imagen (Antialiasing)", *Neftali.clubdelphi.com*, 2020. [Online]. Disponible en: https://neftali.clubdelphi.com/redimensionar-una-imagen-antialiasing/. [Accedido: 19- May- 2020].