



**DEPARTAMENTO
DE COMPUTACION**

Facultad de Ciencias Exactas y Naturales - UBA

Demosaicing

12 de diciembre de 2014

Métodos Numéricos
Trabajo Práctico Nro. 3

Integrante	LU	Correo electrónico
Martin Carreiro	45/10	martin301290@gmail.com
Kevin Kujawski	459/10	kevinkuja@gmail.com
Juan Manuel Ortiz de Zárate	403/10	jmanuoz@gmail.com



Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

<http://www.fcen.uba.ar>

Índice

1. Resumen	3
2. Introducción teórica	4
2.1. PSNR	5
2.2. Artifacts	5
3. Desarrollo	7
3.0.1. Bordes	7
3.1. Vecinos	7
3.2. Bilineal	8
3.3. Direccional	9
3.3.1. Splines Cubicos	9
3.3.2. Decisión de parámetros	10
3.4. High Quality	10
3.5. Herramientas desarrolladas	12
3.5.1. Image.py	12
3.5.2. green_psnr.py	12
3.5.3. image_random.py	12
4. Experimentación Y Resultados	13
4.1. Experimentación individual	13
4.1.1. Colores	13
4.1.2. Tablero de colores	16
4.1.3. Imagen 9 - Avión	18
4.2. Experimentación comparativa	21
4.2.1. Comparación de tiempos	21
4.2.2. Comparación de calidad	23
4.2.3. Calidad subjetiva	23
4.2.3.1. Moiré / False color	24

4.2.3.2. Zippering	24
4.2.3.3. Blur	25
4.2.3.4. Ringing	25
5. Discusión	26
5.1. Colores contra cuadrados de colores	26
6. Conclusiones	27
6.1. Algoritmos	27
6.2. Bordes	27
6.3. Otro uso	27

1. Resumen

En el siguiente trabajo investigaremos el comportamiento de distintos algoritmos pensados para resolver el problema de Demosaicing. Describiremos sus distintos funcionamientos, realizaremos pruebas transversales a todos ellos y pruebas particulares a cada uno. Con el objetivo de, en primera instancia, hacer comparaciones a grandes rasgos de los resultados obtenidos para luego con las pruebas individuales evaluar el resultado en los casos potencialmente conflictivos de cada uno y así tener un panorama más completo a la hora de concluir cual es el mejor o si sus eficiencias varian según los contextos de uso.

2. Introducción teórica

El problema de Demosaicing consta de poder construir una imagen con información de los 3 canales en cada uno de sus pixel en base a una que sólo tiene definidos los valores de un sólo canal para cada celda. Este desafío es muy común en las camáras de fotos digitales ya que en realidad cada fotosensor detecta un sólo color, por lo cual a la imagen capturada hay que aplicarle algún procedimiento lógico para que el usuario final pueda verla efectivamente con todos los colores.

Particularmente estaremos analizando 4 algoritmos distintos que intentan solucionar esto. Ellos son: Vecinos, Bilineal, Direccional y High Quality. Todos estos deben aplicarse sobre imágenes en formato Bayer array, es decir, imágenes que en cada pixel tienen información sólo de un canal (Verde, Rojo o Azul) y estos además tienen una distribución especial (en la que por ejemplo el verde predomina en cantidad ya que es el color que mejor recepción tiene en el ojo humano).

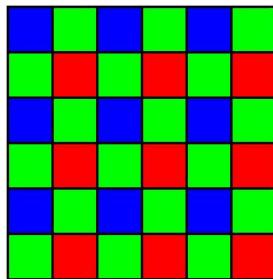


Figura 1: Patrón de píxeles en una imagen bayerizada

Para nuestras pruebas lo que haremos es tomar imágenes con todos sus canales completos en todos los pixeles y pasarlas al formato Bayer array. Como en lo que queremos hacer foco en este trabajo es la calidad de estos procedimientos y sus resultados, no nos interesa que la imagen Bayer haya sido tomada realmente por una cámara, es más, nos sirve más tener una imagen full color y transformala ya que así podemos comparar nuestros resultados contra la original.

Por esto es que desarrollamos una función muy simple que realiza esta transformación. Básicamente lo que hace es:

- Celdas en columnas y filas pares, deja sólo el canal azul
 - Celdas en columnas y filas impares, deja sólo el canal rojo
 - En todas las otras dejamos sólo el canal verde



Imagen original y bayerizada

2.1. PSNR

Es un método para definir la relación entre una señal y el ruido de la regeneración de la misma expresado en decibeles. En este caso, así como es comúnmente utilizado, sirve para resolver si la imagen final es "parecida.^a a la original. Es importante notar que a medida que mayor sea el resultado, mejor es la calidad de la imagen.

2.2. Artifacts

Una forma que utilizaremos para medir la calidad subjetiva de los distintos algoritmos de demosaicing para las imágenes con las que experimentaremos es mediante la detección de **artifacts**. Los artifacts son defectos comunes que ocurren en el procesamiento de imágenes y que han sido categorizados para un mejor análisis, entre los cuales podemos encontrar y nos enfocaremos: Moiré, False Color, Zippering, Blur y Ringing/Edge. Por tal motivo vamos a definir y mostrar un ejemplo de cada uno:

Moiré: El artifact Moiré ocurre en zonas de la imagen en donde hay líneas muy cercanas y que producto de los algoritmos de demosaicing se producen superposiciones de las mismas, dando como resultado en la mayoría de los casos un nuevo patrón sobre ellas



Ejemplo de imagen original vs artifact Moiré

False Color: El artifact False Color ocurre cuando el algoritmo de demosaicing arroja como resultado una zona en la imagen de distinto color al original. Suele venir acompañado cuando ocurre Moiré

Zippering: El artifact Zippering (cremallera) es un artifact que suele ocurrir en zonas de la imagen donde hay elementos borde, donde en la misma se genera un cambio abrupto en los valores, y consiste en que se produce un efecto de lineas que tienen mucha similitud a una cremallera en estos bordes



Figura 2: Ejemplo de imagen original vs artifact Zippering

Blur: Como su nombre lo indica, el artifact Blur (borroneo) se observa cuando la imagen se ve más borrosa, con menos nitidez y definición que la original



Figura 3: Ejemplo de imagen original vs artifact Blur

Ringing: El artifact Ringing aparece con frecuencia en zonas de borde produciendo que los mismos pierdan suavidad (los bordes sufren un pixelamiento) y aparezca un *borde fantasma*, que consiste en un borde alrededor como si fuese una sombra del mismo borde.

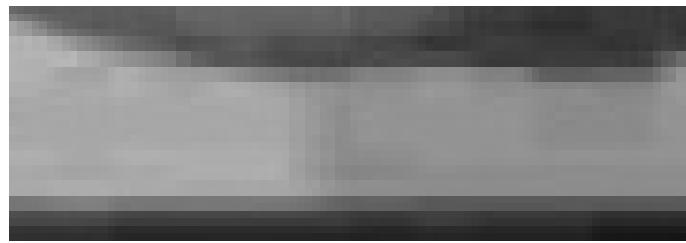


Figura 4: Ejemplo de artifact Ringing en borde recto y redondeado

3. Desarrollo

Como explicamos en la introducción estaremos analizando 4 algoritmos distintos, estos son: Vecinos, Bilineal, Direccional y High Quality. En líneas generales anticiparemos que los primeros dos se utilizan para rearmar toda la imagen desde la bayerización. El resto serán utilizados para aproximar mejor el valor del verde original en los píxeles que en la bayerización solo habían atrapado rojo y azul. Esto es porque el verde es el color que más ve el ser humano y mientras mejor esté la imagen en ese color, subjetivamente quedará mejor.

La idea será comparar sus resultados subjetivamente, es decir a simple vista como vemos la imagen resultante, objetivamente y temporalmente, cuanto tiempo demora en ejecutarse el procedimiento, para poder concluir finalmente las ventajas y desventajas de cada uno. A continuación explicaremos como implementamos cada uno.

3.0.1. Bordes

Los 4 algoritmos que presentaremos a continuación aproximan los colores de los píxeles en base a otros píxeles que se encuentran alrededor del mismo, por esta razón muchas veces no es posible aproximar estos valores en los bordes de la imagen ya que no se tiene información al respecto más allá de los mismos. Por lo cual, salvo el algoritmo de Vecinos que lo permitía, decidimos no aproximar los valores de los píxeles en el borde de la imagen al momento del demosaicing.

3.1. Vecinos

Este es el más simple de todos. La idea es establecer el valor de los colores faltantes de cada pixel en base al vecino que tenga dicho valor. Por ejemplo si estamos en un pixel azul le preguntamos a algún vecino rojo y otro verde su valor y los seteamos en los correspondientes colores de nuestro pixel.

Se diferencia entre fila par e impar cuando la celda es verde ya que es distinta la disposición de sus vecinos en cada caso, esto en cambio se mantiene inmutable en los casos de celda roja o azul. La selección del vecino izquierdo y superior izquierdo cuando es roja es porque siempre existe ese vecino, a diferencia del derecho por ejemplo ya que el rojo puede ser una celda borde y no tener dicho adyacente. La misma idea aplicamos al caso de la celda azul y de las verdes, apuntamos a elegir el vecino que siempre existe.

Algorithm 1 vecinos(*imagenBayerizada*)

```

1: for cada celda en imagenBayerizada do
2:   if celda es roja then                                ▷ Fila y columna impar
3:     celda.verde = vecinoIzq.verde;
4:     celda.azul = vecinoSuperiorIzq.azul;
5:   if celda es azul then                                ▷ Fila y columna par
6:     celda.rojo = vecinoInferiorDerecho.rojo;
7:     celda.verde = vecinoDerecho.verde;
8:   if celda es verde & filaPar then
9:     celda.rojo = vecinoDerecho.rojo;
10:    celda.azul = vecinoSuperior.azul;
11:   if celda es verde & filaImpar then
12:     celda.rojo = vecinoInferior.rojo;
13:     celda.azul = vecinoIzquierdo.azul;

```

3.2. Bilineal

La interpolación es una forma de aproximar valores desconocidos en algún punto a partir de valores que sí conocemos, sabiendo de antemano que van a tener un cierto grado de error. Esto se puede trasladar a nuestro problema, ya que tenemos que aproximar valores para un color de pixel a partir de valores cercanos conocidos, con la única diferencia que en vez de ser en una dimensión como si fuese una función, esta será en dos dimensiones, de ahí el nombre de '*bilineal*'. Esta interpolación en 2 dimensiones se realiza a partir de los 4 puntos más cercanos del color del pixel que queremos aproximar. Para calcular el valor de dicho punto primero se saca el promedio de los valores en dirección horizontal, luego en dirección vertical y por último el promedio de dichos valores. Cabe aclarar que dependiendo el color de pixel actual y el color que se quiera aproximar, podemos llegar a tener información de solo 2 puntos cercanos, pero alineados en la misma dirección, con lo que solo se calculará la aproximación en esa dirección.

Algorithm 2 bilineal(*imagenBayerizada*)

```

1: for cada celda en imagenBayerizada do
2:   if celda es roja then                                ▷ Fila y columna impar
3:     celda.azul = funcionLineal(puntosOblicuosALaCelda, imagenBayerizada, AZUL);
4:     celda.verde = funcionLineal(puntosAdyacentesALaCelda, imagenBayerizada, VERDE);
5:   if celda es azul then                                ▷ Fila y columna par
6:     celda.rojo = funcionLineal(puntosOblicuosALaCelda, imagenBayerizada, ROJO);
7:     celda.verde = funcionLineal(puntosAdyacentesALaCelda, imagenBayerizada, VERDE);
8:   if celda es verde & filaPar then
9:     celda.rojo = funcionLineal(puntosSuperiorEInferior, imagenBayerizada, ROJO);
10:    celda.azul = funcionLineal(puntosDerechaEIzquierda, imagenBayerizada, AZUL);
11:   if celda es verde & filaImpar then
12:     celda.azul = funcionLineal(puntosSuperiorEInferior, imagenBayerizada, AZUL);
13:     celda.rojo = funcionLineal(puntosDerechaEIzquierda, imagenBayerizada, ROJO);

```

Algorithm 3 *funcionLineal(puntos, imagenBayerizada, color)*

```

1: resultado = 0;
2: cantPuntos = 0;
3: for cada punto en puntos do
4:   if punto enRango then
5:     resultado += punto.color
6:   cantPuntos += 1
return resultado/cantPuntos

```

3.3. Direccional

Para este algoritmo nos basamos en lo explicado por Burden y Faires[1] para el cálculo de los splines y en lo desarrollado por Ron Kimmel[2] para el algoritmo en sí. El método lo aplicamos sólo para el color verde mientras que para los otros utilizamos bilineal.

Algorithm 4 *direccional(imagenBayerizada)*

```

1: hacerBilineal(imagenBayerizada)
2: coeficientes = new Matrix([0,0]) ▷ De la tupla el primero representa el coeficiente horizontal, el otro el vertical
3: for cada fila en imagenBayerizada.menosPrimerayUltima do
4:   resolverPorSpline(horizontal, fila) ▷ Llenará la matriz de coeficientes
5: for cada columna en imagenBayerizada.menosPrimerayUltima do
6:   resolverPorSpline(vertical, columna) ▷ Llenará la matriz de coeficientes
7: for cada celda en imagenBayerizada.menosElBorde do
8:   if celda es verde then
9:     derivadaX, derivadaY = calcularDerivadasDireccionales(celda)
10:    if derivadaX > derivadaY then
11:      celda.verde = 0,3 * coeficientes.celda.horizontal + 0,7 * coeficientes.celda.vertical;
12:    else
13:      celda.verde = 0,7 * coeficientes.celda.horizontal + 0,3 * coeficientes.celda.vertical;

```

Dado que las interpolaciones mediante splines no son nada triviales lo explicaremos mas adelante en detalle. Las derivadas en x la aproximamos haciendo $|G(x - 1, y) - G(x + 1, y)|$ donde G es el valor del color verde en ese punto, la derivada en y es análoga. Dado que un mayor valor en la derivada puede estar indicándonos un potencial borde le damos mayor peso a la derivada cuyo valor es más chico multiplicando a este por 0.7 y a la otra por 0.3. Finalmente las sumamos para obtener el verde correspondiente en nuestra celda.

3.3.1. Splines Cubicos

Los Splines son funciones partidas que dadas n particiones disjuntas del dominio generan n funciones para cada subconjunto interpolando todos los puntos en el medio. Para nuestro caso, cada subconjunto estará dado por cada pixel que en la bayerización contenía solo verde adjunto a un pixel rojo y azul para cada dirección, vertical y horizontal. Es decir, en la dirección horizontal, el verde de la izquierda, y el de la derecha ; para vertical, el de arriba y el de abajo. La función *resolverPorSpline* mantendrá en cada posición representando cada celda, los resultados de la siguiente función para cada dirección. Una vez que tenemos cada una de esas funciones que tendrán la forma

$$f_j(x) = d_j(x - x_j)^3 + c_j(x - x_j)^2 + b_j(x - x_j) + a_j \quad \forall x_j \leq x < x_{j+k}, \text{ en nuestro caso } k = 2$$

Solo queda calcular el valor del verde en el pixel rojo o azul. Es fácil notar que $(x - x_j) = 1$, por lo que para cada función el resultado va a ser $d_j + c_j + b_j + a_j$, siendo a_j = el verde original en ese pixel.

Solo queda triangular y obtener cada una de las b_j , c_j y d_j .

La implementación está extraída del algoritmo presentado en el libro Burden y Faires [1].

3.3.2. Decisión de parámetros

La elección de color está dada en relación a los valores calculados por el spline, y como ya hemos explicado cada uno aportará en distinta proporción a partir del valor de la derivada. La pregunta es ahora, en cuánta medida para cada una. Hicimos pruebas y llegamos a la conclusión de que lo mejor es el par 03,07 a partir de la comparación objetiva de PSNR. Recordemos que a mayor valor, mejor es el resultado y más parecido a la imagen original.

	<i>PSNR</i>
03, 07	37,7168767571
07, 03	37,3389911482
01, 09	36,9913700856
09, 01	36,0896037821
05, 00	16,0625448484
00, 05	16,0544599348

Todas las pruebas recién mostradas son de la imagen img1.bmp, incluída en este trabajo, pero la proporción se mantuvo en valores similares para otras pruebas.

3.4. High Quality

Todos los algoritmos anteriores aproximaban el valor mediante un color, pero esto no era suficiente para darle la suficiente definición ya que muchas veces los colores tienen un brillo o luminicencia que impacta en los tres pero que no se distingue cuando para el calculo del mismo se usan valores cercanos de ese , por lo tanto el algoritmo de demosaicing que denominamos "High Quality" y se basa en el paper de Malvar, He y Cutler [3], propone realizar cada color para darle una mejor definición utilizando los colores restantes. Por lo tanto se podría decir que este algoritmo no es un algoritmo para aproximar colores faltantes, sino que se utiliza a partir de una aproximación ya obtenida anteriormente para darle una mejor calidad a la misma.

Como la imagen Bayerizada se compone del doble de pixeles verdes que el resto, este resulta ser el color más importante para encontrar la interpolación correcta. Por lo tanto, y como figura en el enunciado del TP, decidimos inclinarlos por solo analizar dicho color, y en consecuencia solo realizando el cálculo apropiado para el mismo.

Antes de comenzar con el algoritmo, queremos hacer una observación de implementación Cuando comenzamos las pruebas del algoritmo de Quality notamos que en zonas oscuras aparecían pixeles con verde mucho mayor a 255 y nuestro primer intento fue fijarlo en 255. Obviamente esto generó pixeles verde claro en zonas indebidas (mayormente oscuras), e intentamos fijarlo en cero en comparación con el nivel de azul y rojo de ese mismo pixel. Pero produjo que en otras zonas se oscurecieran cuando no debían.

Este caso no estaba contemplado en el paper, y pudimos observar cual era el problema. Quality aproxima el valor del verde dado el promedio de sus vecinos rojos o azules para dar luminicencia, el problema es que si el valor actual del verde era muy chico comparado con estos rojos y azules mencionados, al hacer la cuenta,

el valor quedaba negativo, y en el producto quedaba mucho más grande que 255, y al hacer la acotación dicha en el primer párrafo era que nos quedaban los verdes saturados.

La solución a dicho problema fue de antemano chequear que la resta, no me convierta un número negativo, en caso de ser así, planchar en cero. De esta manera no arruinamos ni zonas oscuras, ni zonas claras.

Algorithm 5 *highQuality(imagenBayerizada)*

```

1: hacerBilineal(imagenBayerizada)
2: for cada celda en imagenBayerizada.menosElBorde do
3:   if celda no es verde then                                ▷ colorActual es rojo o verde
4:     Sumo los valores del color actual que se encuentran a 2 pixeles de distancia
5:     gradiente = imagenBayerizada.dosArriba().colorActual
6:     gradiente += imagenBayerizada.dosAbajo().colorActual
7:     gradiente += imagenBayerizada.dosDerecha().colorActual
8:     gradiente += imagenBayerizada.dosIzquierda().colorActual
9:     correccion = (celda.colorActual - gradiente/4)      ▷ Obtengo el valor del color actual en esta
   posicion y le resto la suma anterior dividido 4
10:    celda.verde = 0,5 * correccion      ▷ A esta ultima suma la multiplico por un alfa = 0.5 y el
    resultado se lo sumo al valor del color verde que tengo en esta posicion

```

El alfa en 0.5 es un valor que aporta el paper para refinar la calidad y mejorar la aproximación.

3.5. Herramientas desarrolladas

Para este trabajo desarrollamos herramientas en python. Una para calcular el error objetivo cometido y otra para correr los scripts en C++. A continuación explicaremos brevemente cada una.

3.5.1. Image.py

Este script se encarga de crear la imagen bayerizada, crear los txt para pasarle a los programas en c++, compilar dichos programas y crear las imágenes resultantes en base a los txt generados por los programas c++.

Recibe 2 parámetros:

1. nombre imagen: debe ser el nombre de la imagen sin su extensión y la misma debe ubicarse en la carpeta 'images' que debe estar a la misma altura que 'src'. La imagen debe estar en formato bmp.
2. algoritmo: debe ser el nombre del algoritmo a ejecutar. Las opciones son: 'vecino', 'quality', 'bilineal' o 'directional'.

Las imágenes resultantes son guardadas en la misma carpeta donde esta este script.

3.5.2. green_psnr.py

Este archivo realiza el cálculo de PSNR entre los colores verdes de todos los píxeles (excepto los bordes) de dos imágenes distintas, las cuales deben pasarse por parámetro.

Los parámetros que se le deben pasar son:

1. imagen 1: ruta de la imagen con el nombre de la imagen y su extensión.
2. imagen 2: ruta de la imagen con el nombre de la imagen y su extensión.

Imprime por pantalla el PSNR resultante.

3.5.3. image_random.py

Este script genera una imagen aleatoria de ancho y alto deseado.

Los parámetros que se le deben pasar son:

1. ancho: ancho en pixeles.
2. alto: alto en pixeles.
3. output: ruta de salida de la imagen random.

4. Experimentación Y Resultados

A continuación expondremos los resultados obtenidos por cada algoritmo para distintas imágenes. El objetivo será posteriormente hacer análisis de calidad subjetiva (es decir que vemos a simple vista), objetiva y tiempo de computos. Para, como dijimos en un principio, determinar ventajas y desventajas de cada uno de ellos. Para los análisis objetivos desarrollamos un programa en python que compara pixel a pixel basado en PSNR (Peak signal-to-noise ratio). Elegimos 3 casos particulares pero como se verá, los resultados son parecidos y se ha comportado de la misma manera en los otros ejemplos dados por la cátedra y lo mismo en otros ejemplos nuestros que no valían la pena ponerlos en el informe.

4.1. Experimentación individual

4.1.1. Colores

Se nos ocurrió que podría ser interesante chequear los comportamientos de estos procedimientos en una imagen con muchos bordes ya que estos, en algoritmos como el directional, son factores importantes y potencialmente conflictivos. Además hicimos que la imagen sea grande (5000 x 3000 pixeles) para poder analizar tiempos de computo y para influir en la calidad subjetiva, ya que si ponemos imágenes con mucha definición pequeños errores podrían pasar desapercibidos para el ojo humano.

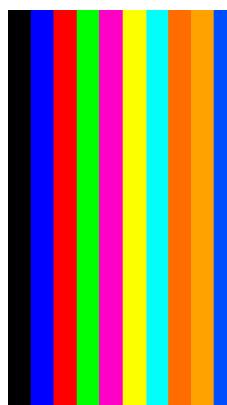


Figura 5: Original

Los resultados obtenidos a primera vista parecían todos iguales. Pero al hacerles zoom, pudimos ver que no era así. En la carpeta *imagenes_extra* que adjuntamos con el código se encuentran tanto la imagen original como sus respectivos resultados.

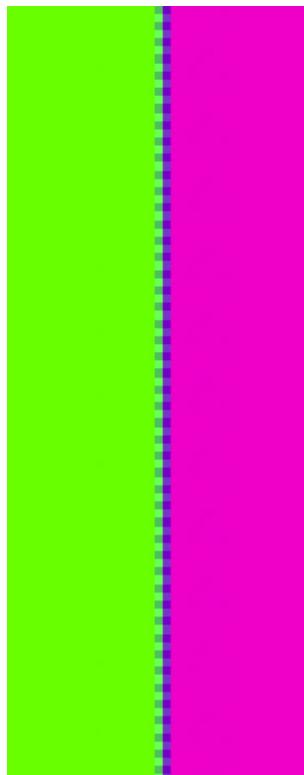


Figura 6: Bilineal Zoom

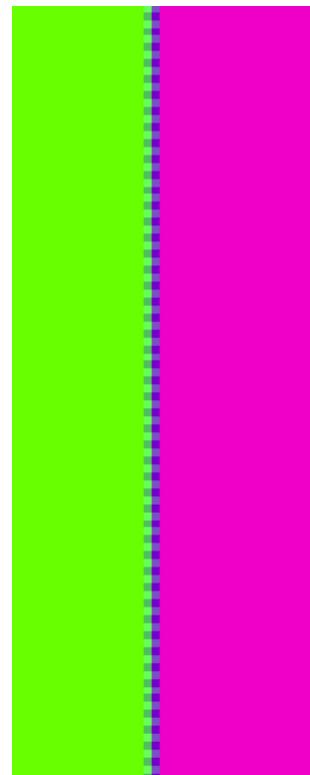


Figura 7: High Quality Zoom

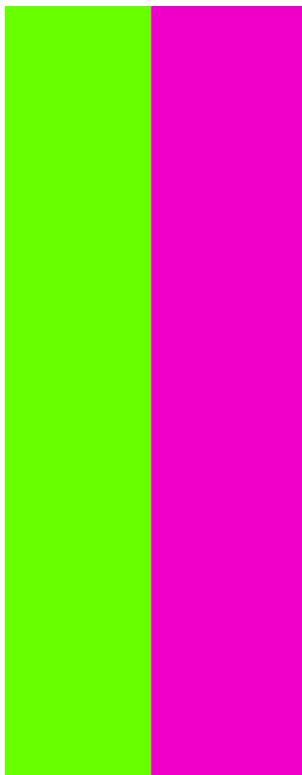


Figura 8: Directional Zoom

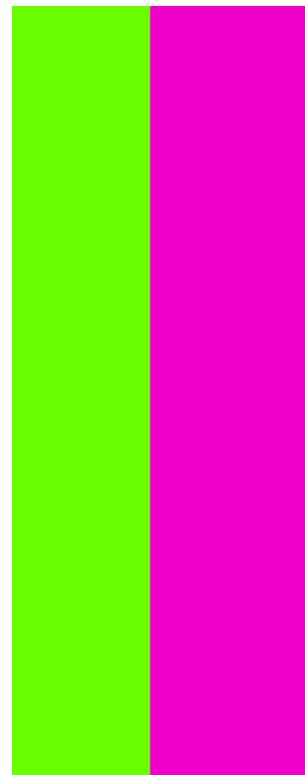


Figura 9: Vecinos Zoom

El algoritmo bilineal y el high quality tuvieron pequeños errores en los bordes. Podemos observar que en esos casos entre el verde y el violeta parece haber como una ‘cosedura’ (*zipper artifact*), la misma se repite en todos los bordes de la imagen. Estas diferencias imperceptibles por el tamaño de la imagen a primera vista pueden ser efectivamente comprobadas mediante un análisis de calidad objetivo.

A continuación podemos ver los PSNR obtenidos al comparar los distintos resultados con la imagen original. Las imágenes que al hacerles zoom tenían esas ”coseduras” efectivamente fueron las que mas bajo PSNR dieron, es decir las que mas difirieron de la real.

	PSNR
<i>highquality</i>	39,67
<i>bilineal</i>	41,14
<i>directional</i>	48,13
<i>vecinos</i>	48,13

A continuación veremos también cual fué el tiempo de computo de cada algoritmo.

	Tiempo(segundos)
<i>directional</i>	67,82
<i>highquality</i>	64,88
<i>bilineal</i>	63,00
<i>vecinos</i>	10,31

Como era de esperarse, las proporciones de tiempo tienen sentido. Vecinos es el mas rápido ya que es el procedimiento mas simple y quality tarda mas que bilineal ya que antes aplica bilineal para luego mejorarlo (aunque en este caso no lo mejora). Para este caso tanto por calidad objetiva, subjetiva como en tiempo de cómputo vecinos parecería ser el mejor.

4.1.2. Tablero de colores

Dado que en la prueba anterior tuvimos un resultado muy fuerte (que el PSNR de vecinos dió mejor que el de HighQuality). Decidimos hacer otra prueba más con este mismo formato para poder investigar un poco más que pasó. La imagen que tomamos fué la siguiente, que es parecida a la anterior en el sentido de que tiene muchos colores con límites bien drásticos,es decir que no hay cambios graduales de color sino que cambian de un pixel a otro.

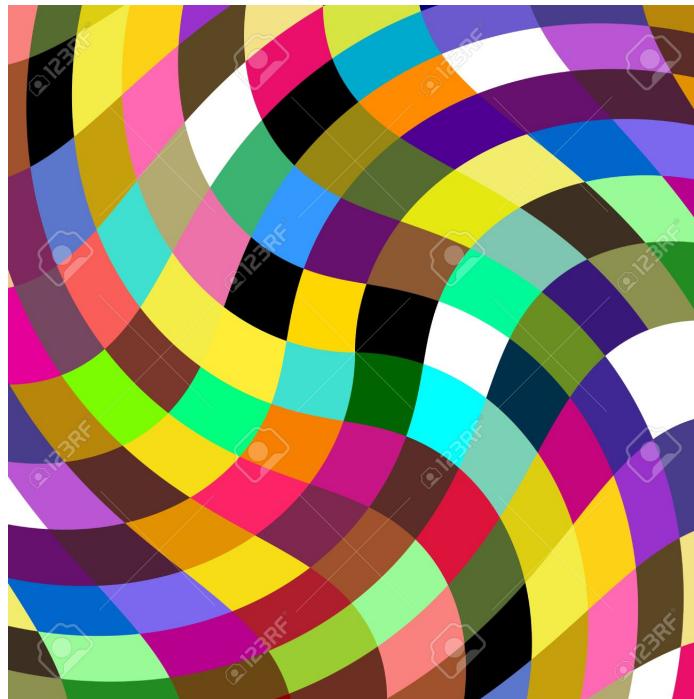


Figura 10: Tablero de colores

Podemos observar que además la imagen tiene una marca de agua. Decidimos utilizarla igual porque nos va a terminar siendo útil también en nuestras pruebas. Esta imagen también es mas chica ya que ahora no nos interesa tanto el tiempo de cómputo

Los resultados obtenidos tuvieron visibles diferencias a la imagen original. Las marcas de agua cambiaron de color (*false color artifact*) en todos los casos y,a diferencia de la prueba anterior, el resultado de vecinos tuvo claros errores en los bordes mientras que los otros a simple vista parecería que no. No exponemos estos resultados para no sobre cargar de imágenes el informe pero dejamos adjunta la imagen original por si se desea corroborar esto haciendo uno mismo los tests.

Calidad cuantificable Yendo a la parte que más nos interesaba, la calidad cuantificable. Podemos observar que ahora si tuvo un comportamiento mas intuitivo.

	PSNR
<i>highquality</i>	38,58
<i>bilineal</i>	36,08
<i>directional</i>	34,90
<i>vecinos</i>	32,00

Efectivamente la mejor esta vez fué el *highquality* y el peor vecinos. También tiene sentido que *spline(direccional)* haya sido peor en calidad ya que depende de toda la fila y columna mientras otros dependen de sus valores más próximos.

Tiempos A continuación veremos cual fué el tiempo que tomó cada algoritmo.

	<i>Tiempo(segundos)</i>
<i>directional</i>	4,46
<i>highquality</i>	4,21
<i>bilineal</i>	4,14
<i>vecinos</i>	0,15

4.1.3. Imagen 9 - Avión

Nos pareció interesante hacer un análisis particular de la Imagen 9 ya que posee varias características en donde se pueden observar diferentes condiciones para ver como actúan los cuatro algoritmos, estos son: tener gran cantidad de sectores bordes, gran variedad de colores que contrastan altamente y principalmente tener texto. Destacamos esto último ya que una de las cualidades más importantes que deben poseer estos algoritmos es la de asegurar nitidez y suavidad, detalles que se ven al procesar imágenes que poseen texto. A continuación mostraremos el resultado de los algoritmos para esta imagen:



Figura 11: Imagen original



Algoritmos de Vecinos e Interpolación Bilineal



Algoritmos de Interpolación Direccional y HighQuality

Como se puede observar, a simple vista pareciera que los 4 algoritmos dan un resultado bastante aceptable, pero mirando detalladamente y comparando entre sí se puede observar una gran diferencia entre ellos, principalmente en zonas borde y en la zona del texto del avión. Subjetivamente pareciera ser que la imagen más parecida a la original es la del HighQuality, seguida por la Interpolación Direccional, pero objetivamente en base a los resultados obtenidos al método PSNR se observa que el algoritmo de Interpolación Bilineal supera al Direccional, algo que llama bastante la atención y que indica que lo subjetivo se diferencia bastante de los objetivo.

<i>algoritmo</i>	<i>PSNR</i>
<i>highquality</i>	37,71
<i>bilineal</i>	35,34
<i>directional</i>	33,35
<i>vecinos</i>	30,49

Yendo más a fondo aún en la cuestión y para mostrar el nivel de detalle que obtiene cada algoritmo, decidimos hacer zoom en la zona donde la imagen posee texto:

Acá si se puede observar la nitidez y suavidad que presenta cada algoritmo, viéndose como mejora la calidad del texto para cada imagen y como va disminuyendo el ruido aumentando la refinación alrededor del mismo.



Figura 12: Vecinos

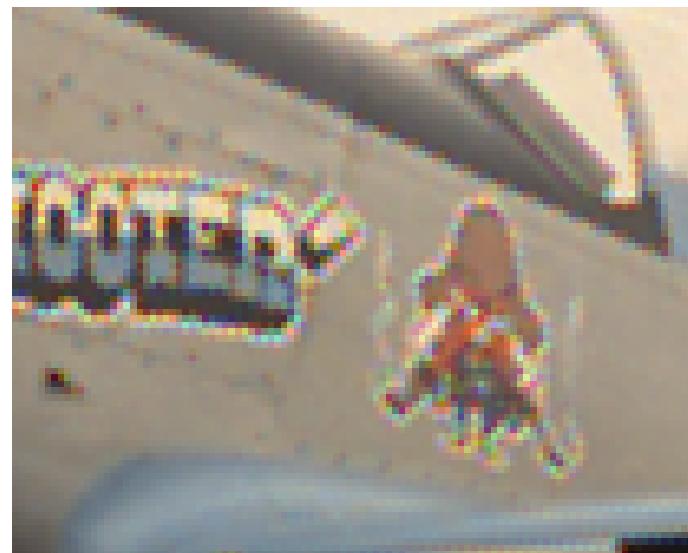


Figura 13: Bilineal



Figura 14: Direccional



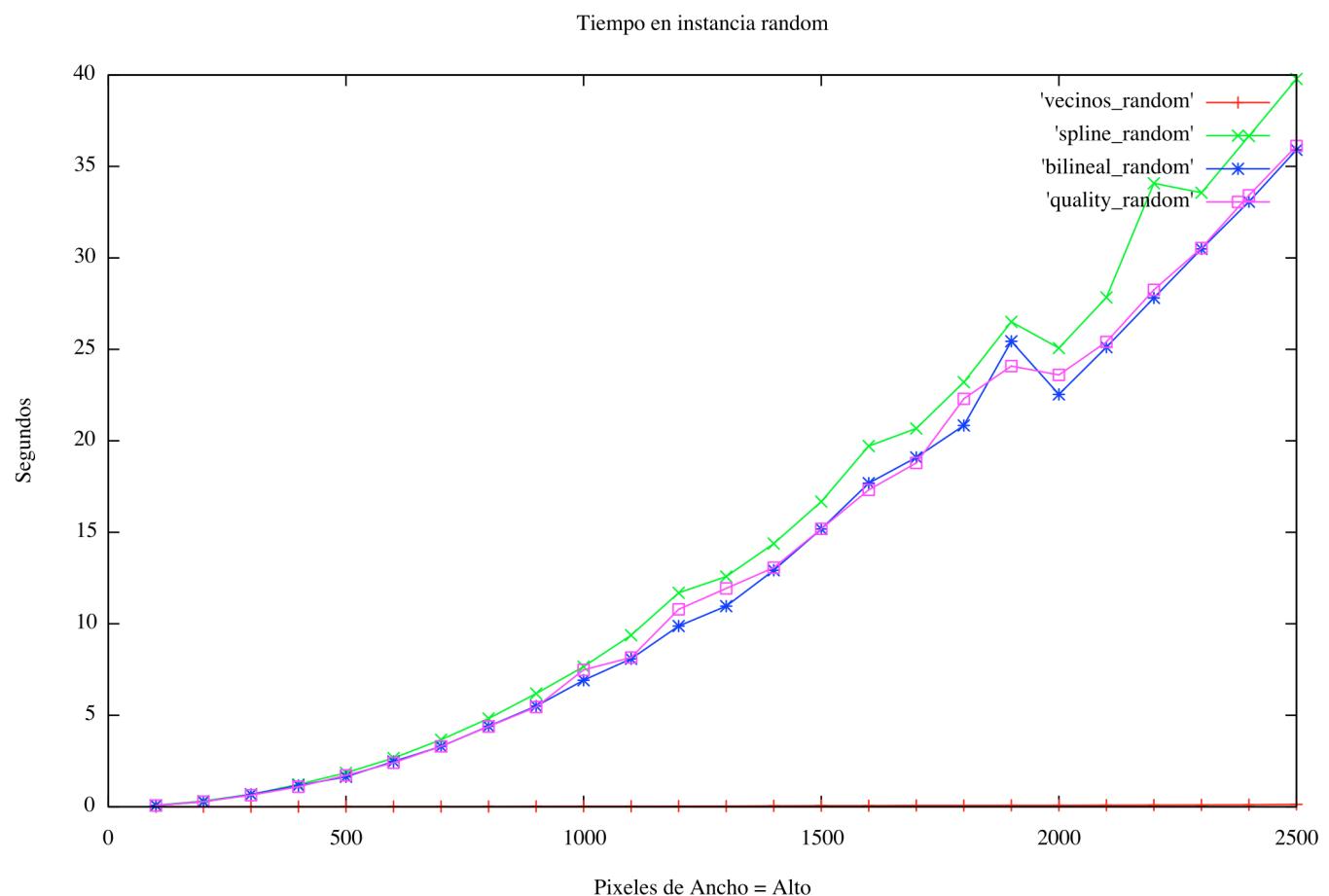
Figura 15: Quality

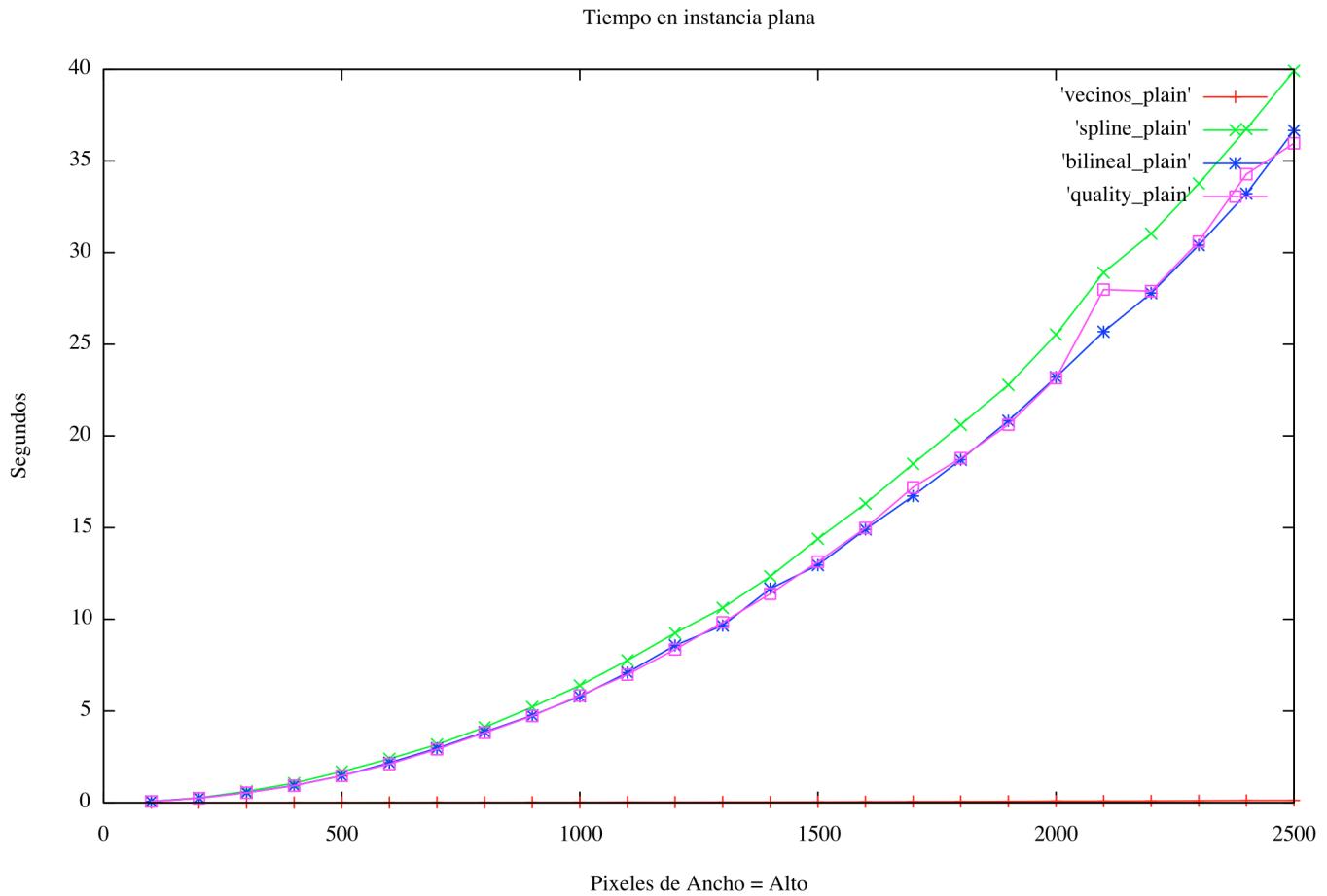
4.2. Experimentación comparativa

4.2.1. Comparación de tiempos

En esta sección presentaremos una comparativa entre los tiempos que tarda cada algoritmo a medida que aumenta el tamaño de la imagen. Para tal fin testamos cada algoritmo con 2 tipos de imágenes cuadradas generadas que van aumentando su tamaño, una con pixeles de colores aleatorios y la otra con todos los pixeles del mismo color, es decir, plana. Creemos que es una forma correcta de comparar los tiempos ya que usamos la misma instancia para cada prueba y además utilizamos estas dos instancias para comprobar si la complejidad de los algoritmos se ve influenciada por la información que contiene la imagen además del tamaño.

A continuación el gráfico con la comparación de tiempos entre los 4 algoritmos para una imagen generada aleatoriamente:





Vecinos al ser el más simple, tiene sentido que haya tardado menos. Lo único que hace es recorrer la imagen y llenar por cada color faltante de cada pixel, de, valga la redundancia, sus vecinos que captaron ese color en la bayerización. Bilineal en segundo puesto, ya que realiza un promedio y necesita ver más puntos.

Tanto Direccional y Quality, utilizan bilineal ya que no son algoritmos de aproximación de cero, si no de mejora. Por lo que ya tienen una cota inferior. Igualmente es notable que Quality siendo el mejor resultado, como venimos viendo en puntos anteriores y un detalle mejor en la próxima sección, también sea el de menor tiempo. Esto se debe a que tiene una cuenta parecida al bilineal pero en pocos píxeles, a diferencia de Direccional que resuelve un sistema de ecuaciones por cada fila y columna.

Igualmente no es poco notar que son bastante lentos los algoritmos, y las pruebas si bien fueron con imágenes relativamente grandes, no alcanzan a las cámaras promedio que claramente no tardan 10 segundos en resolver la foto final. O existen optimizaciones que no llegamos a hacer o hay un mejor aprovechamiento del hardware, paralelismo, etc. Como el algoritmo de HighQuality solamente procesa los verdes de la imagen luego de primero utilizar el algoritmo Bilineal, el tiempo que tardará es el de este último más el tiempo del agregado para que las imágenes tengan una mayor calidad para el color verde. Por último se puede observar que como creímos, los tiempos que tardan los algoritmos comparando las dos instancias para cada tamaño es casi idéntico, por lo que podemos concluir que el contenido de la imagen a procesar no es tan influyente como si lo es el tamaño de la misma.

Para los experimentos incluimos dos archivos random.sh y plain.sh para simular los mismos

4.2.2. Comparación de calidad

En esta sección compararemos solo los verdes de las imágenes y veremos los resultados utilizando PSNR. Recordemos que este término, calcula nivel de ruido en una señal.

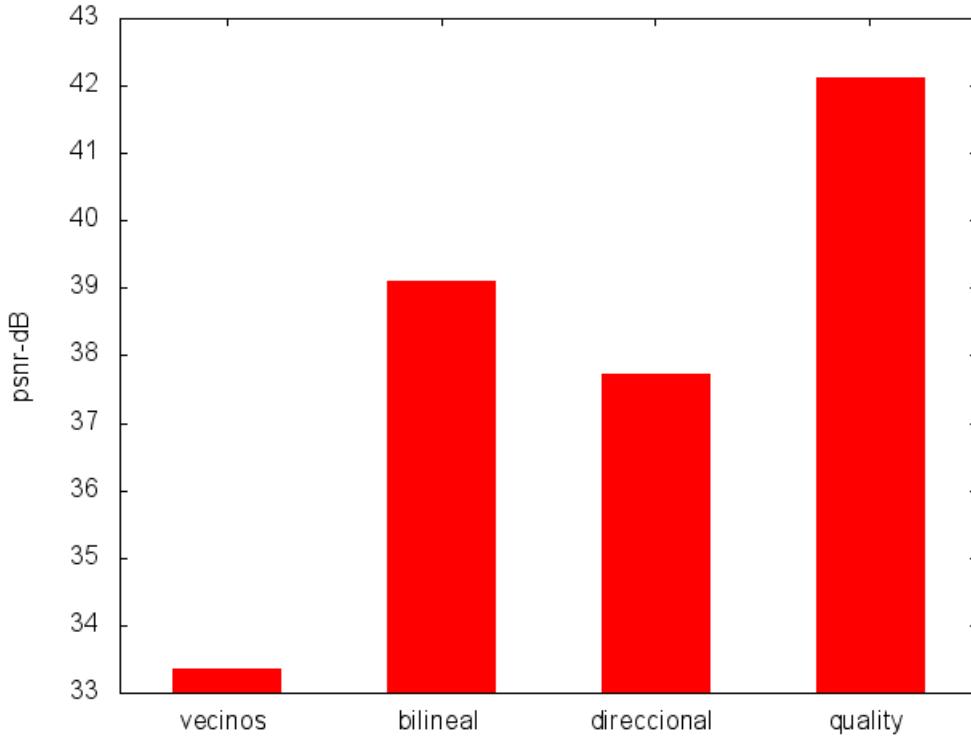


Figura 16: Img1

Como venimos viendo en otros ejemplos, img9 y este, como era esperado, el vecinos es el que peor hizo, y quality fue el mejor. Sin embargo notemos que Spline si bien tendría que tener un valor mayor que bilineal ya que subjetivamente notamos mejora entre ellos, no lo tiene, es decir, el nivel de ruido es mayor en el direccional. Esto tiene sentido en la medida que el algoritmo direccional utiliza toda la fila y toda la columna para calcular el valor de las coordenadas b_j , c_j y d_j por lo que está sujeto a valores distintos al valor final, a diferencia de Bilineal que está sujeto a sus vecinos directos que es mucho más probable que tengan un valor más próximo.

4.2.3. Calidad subjetiva

En las secciones anteriores analizamos la calidad de las imágenes basándonos en los valores de los píxeles que resultaban después de aplicar los distintos algoritmos de demosaicing y la comparamos con la original pero no teníamos en cuenta las diferencias entre estas a la vista del ojo humano. Por esto último en esta sección mediremos la calidad subjetiva de las imágenes en base a los **artifacts** que se pueden encontrar en cada una de ellas.

A continuación los resultados a la experimentación con las distintas imágenes de prueba, para cada artifact mostraremos el resultado que nos pareció mas productivo para mostrar, pero dejando en claro que la búsqueda y análisis fue en base a todos los algoritmos:

4.2.3.1 Moiré / False color



Figura 17: Original



Figura 18: Demosaicing (HighQuality)

4.2.3.2 Zippering



Figura 19: Original

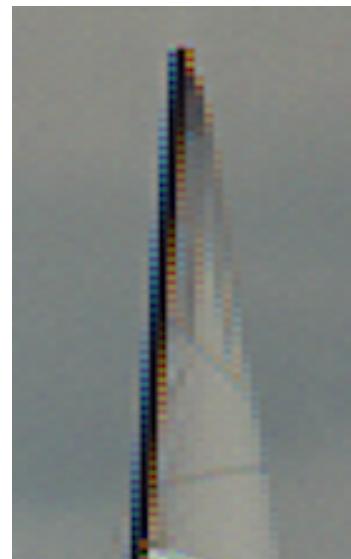


Figura 20: Demosaicing (Vecinos)

4.2.3.3 Blur



Figura 21: Original

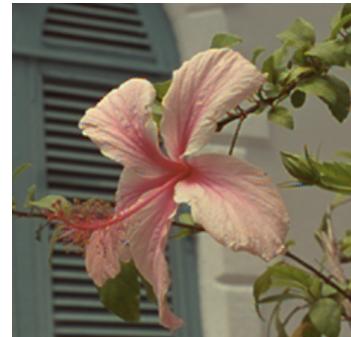


Figura 22: Demosaicing (Bilineal)

4.2.3.4 Ringing

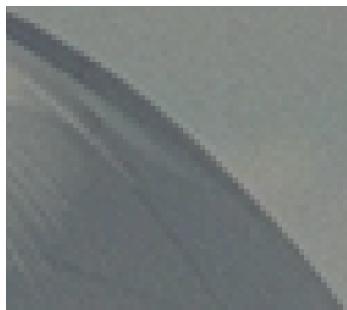


Figura 23: Original

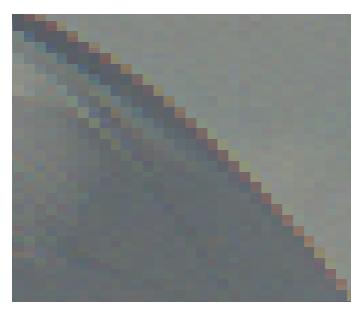


Figura 24: Demosaicing (Vecinos)

Lo llamativo es que en mayor o menor intensidad los artifacts aparecen en los 4 algoritmos, aunque no quiere decir que esto siempre sea así ya que vimos en la sección de experimentación individual un caso en que no ocurría. Por lo tanto, como esta comparación es subjetiva, basandonos en nuestro “*ojo humano*”, el algoritmo en donde los artifacts se aprecian con menor intensidad es en HighQuality, seguido por Spline, Bilineal y Vecinos. Esto era de esperarse ya que los algoritmos de HighQuality y Spline primero aproximan la imagen con el algoritmo Bilineal y luego la mejoran. Y también el resultado de Vecinos fue el esperable ya que es un algoritmo bastante básico.

5. Discusión

5.1. Colores contra cuadrados de colores

Cuando probamos con la primer imagen de colores, vecinos resultó ser el que mejor resultados tuvo en todos los sentidos. Esto nos llamó mucho la atención ya que es un algoritmo muy simple para superar a otros desarrollados en distintos papers.

Por eso decidimos hacer una segunda prueba con una imagen que tenga la misma idea pero más compleja (con límites tambien horizontales, curvos y con la marca de agua) y ahí si obtuvimos resultados mas acordes. Debido a nuestra implementación los límites que son tan rectos y verticales tienen una baja influencia sobre el error. Esto es porque pocos píxeles son los que necesitan pedirle el color a vecinos de distintos lados de la frontera. Por eso fué que funcionó tan bien en el primer caso. En cambio en el segundo fué mucho más ineficiente por las complejidades antes mencionadas.

6. Conclusiones

6.1. Algoritmos

En base a los algoritmos presentados, concluimos que a partir del tiempo de cómputo y resultados que ofrecen los mismos, siempre conviene optar por el algoritmo de HighQuality, ya que como se observa en las secciones anteriores, la calidad que ofrece es extremadamente buena y el tiempo que tarda es aceptable. Otra cosa que nos llama la atención fueron los resultados obtenidos por las dos interpolaciones bilineales y direccionales, ya que si bien en teoría la direccional debería obtener mejores resultados, estos solo se aprecian al mirarlos, es decir subjetivamente, ya que objetivamente el bilineal aproxima mejor los colores originales y obtiene un menor ruido en la señal del color verde.

6.2. Bordes

En caso de una imagen con cambios grandes en zonas pequeñas, todos los algoritmos tienden a fallar en esas zonas. Esto se debe a que todos los algoritmos dependen fuertemente de los pixeles próximos cercanos, los cuales poseen valores muy distintos al valor a calcular. Esto es denominado borde.

Otro punto interesante es que si bien el algoritmo de High Quality funcionó mejor objetiva y subjetivamente para las imágenes que son fotografías esto no fué así en el caso de la imagen "colores". Pensamos que esto se debe a que este procedimiento fué especialmente pensado para fotografías, es decir imágenes con bordes mas suaves o sin tanta saturación de color de cada lado del borde. Y que es por esto que en este caso los resultados dieron casi lo contrario de todo el resto (vecinos fué el mejor en todo sentido).

6.3. Otro uso

Si bien el TP está apuntado a los algoritmos que corren en las cámaras de fotos, un uso útil puede ser el de compresión de imágenes, ya que bayerizando la imagen original reduce en un tercio su tamaño y en poco tiempo (menor aún en la práctica cuando lo usan las cámaras digitales) se puede obtener la original, o al menos una aproximación bastante acertada de la misma.

Referencias

- [1] R. Burden y J.D.Faires, Análisis numerico, International Thomson Editors, 9th edition, 2011, page 149
- [2] Ron Kimmel. Demosaicing: Image reconstruction from color ccd samples. Image processing, IEEE transactions on, 1999.
- [3] Henrique S. Malvar, Li-wei He y Ross Cutler, High-quality linear interpolation for demosaicing of bayer-patterned color images, Microsoft Research One Microsoft Way, Redmond, WA 98052