



**DEPARTAMENTO
DE COMPUTACION**

Facultad de Ciencias Exactas y Naturales - UBA

Demosaicing

13 de noviembre 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	2
2. Introducción teórica	3
3. Desarrollo	4
3.1. Vecinos	4
3.2. Bilineal	4
3.3. Direccional	5
3.3.1. Splines Cubicos	6
3.4. High Quality	6
4. Experimentación Y Resultados	8
4.1. PNSR	8
4.2. Colores	8
4.3. Imágen 9	13
4.4. Comparación de tiempos	15
4.5. Comparación de calidad	16
5. Conclusiones	17

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 cámaras 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).

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

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, osea 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.

3.1. 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.

Pseudocódigo:

```

1 Para cada celda:
2     si es roja:
3         celda.verde = vecinoIzq.verde
4         celda.azul = vecinoSuperiorIzq.azul
5     si es azul:
6         celda.rojo = vecinoInferiorDerecho.rojo
7         celda.verde = vecinoDerecho.verde
8     si es verde y fila par:
9         celda.rojo = vecinoDerecho.rojo
10        celda.azul = vecinoSuperior.azul
11    si es verde y fila impar:
12        celda.rojo = vecinoInferior.rojo
13        celda.azul = vecinoIzquierdo.azul

```

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.

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 mas 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.

Para todos los pixeles de la imagen:

Si estoy parado en un pixel azul:

Calculo el valor de rojo que tendrá el pixel
haciendo el promedio de los valores en rojo de los
4 puntos oblicuos a este (que estén en rango).

Calculo el valor de verde que tendrá el pixel
haciendo el promedio de los valores en verde de los

4 puntos adyacentes a este (que estén en rango).

Si estoy parado en un pixel rojo:

Calculo el valor de azul que tendrá el pixel
haciendo el promedio de los valores en azul de los
4 puntos oblicuos a este (que estén en rango).

Calculo el valor de verde que tendrá el pixel
haciendo el promedio de los valores en verde de los
4 puntos adyacentes a este (que estén en rango).

Si estoy parado en un pixel verde:

Si estoy parado en una fila de azules:

Calculo el valor de rojo que tendrá el pixel
haciendo el promedio de los valores en rojo de los
2 puntos superiores e inferiores (que estén en rango).

Calculo el valor de azul que tendrá el pixel
haciendo el promedio de los valores en azul de los
2 puntos a derecha e izquierda (que estén en rango).

Si estoy parado en una fila de rojos:

Calculo el valor de azul que tendrá el pixel
haciendo el promedio de los valores en azul de los
2 puntos superiores e inferiores (que estén en rango).

Calculo el valor de rojo que tendrá el pixel
haciendo el promedio de los valores en rojo de los
2 puntos a derecha e izquierda (que estén en rango).

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.

Lo que hace es:

```
[frame=single]
Para cada celda:
Interpolo mediante spline su fila y columna
Calculo sus derivadas aproximadas en dirección horizontal y vertical
Si la derivada horizontal es mayor:
celda.verde = interpolacion horizontal * 0.3 + interpolacion vertical * 0.7
sino
celda.verde = interpolacion horizontal * 0.3 + interpolacion vertical * 0.7
```

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. 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 .

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" se basa en el paper de Malvar, He y Cutler, propone realizar cada color para darle una mejor definición utilizando los colores restantes. Por lo tanto se podria 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 calculado 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 parche plancharlo en 255. Obviamente esto generó pixeles verde claro

en zonas indebidas (mayormente oscuras), e intentamos plancharlo 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.

```
1 Primero hacemos bilineal sobre todos los colores de la imagen
2 Por cada pixel de la imagen (exceptuando los bordes):
3   Si la imagen cae en verde la ignoro
4   Si cae en rojo o azul:
5     Al color verde de ese pixel le sumo el valor del color actual calculado en la
       bilineal
6     Sumo los valores del color actual que se encuantran a 2 pixeles de distancia
7     Obtengo el valor del color actual en esta posicion y le resto la suma
       anterior dividido 4
8     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.

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).

4.1. PNSR

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.

4.2. 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.

Tambien expondremos la imagen bayerizada para que quede claro que la conversión que estamos haciendo es correcta.

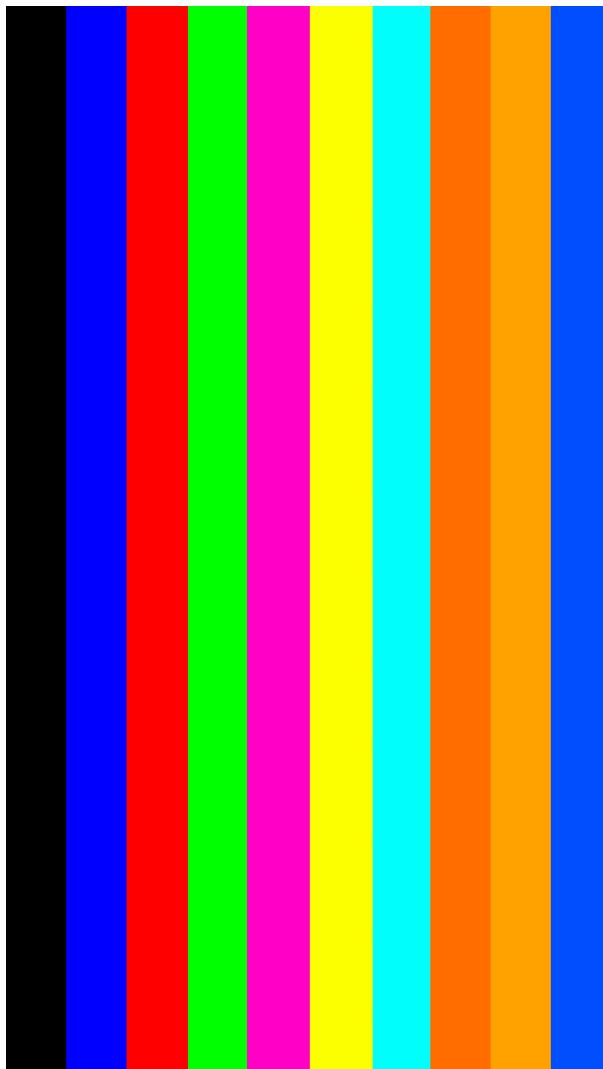


Figura 1: Original

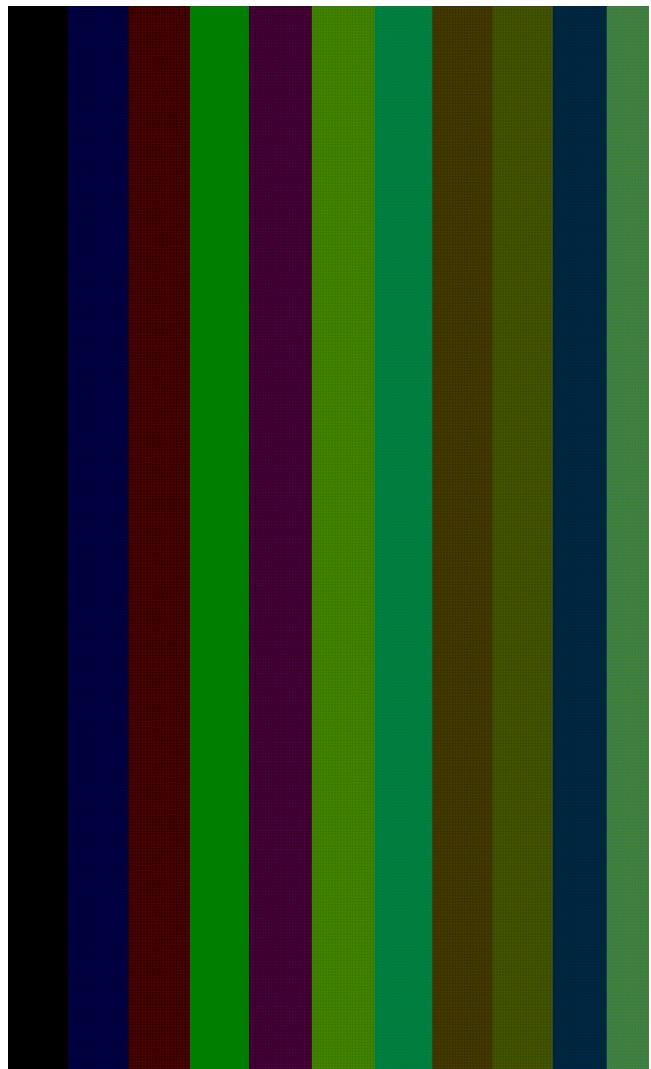


Figura 2: Bayerizada

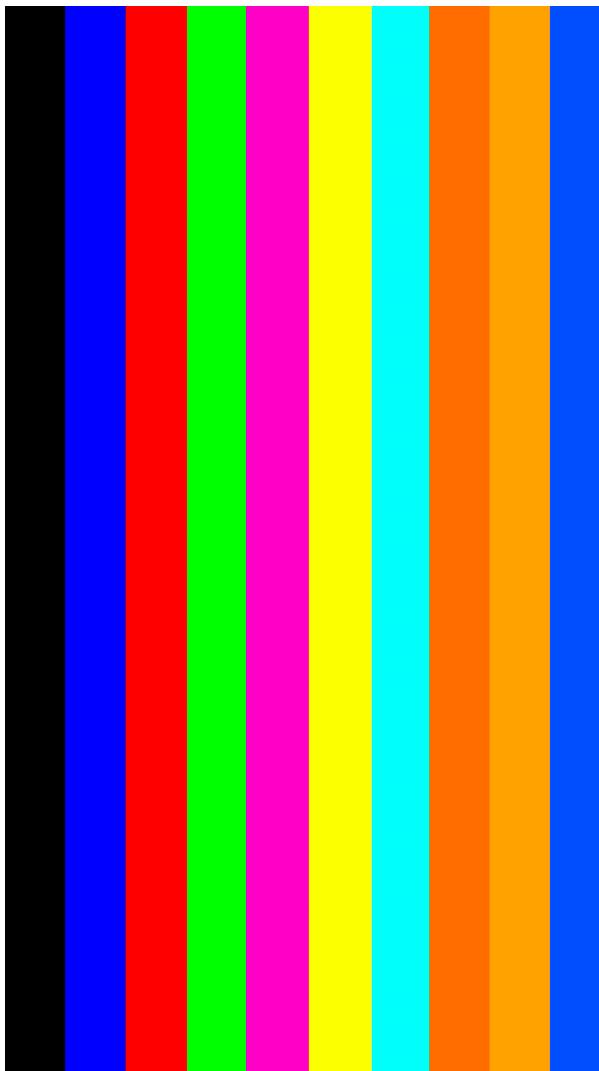


Figura 3: Bilineal

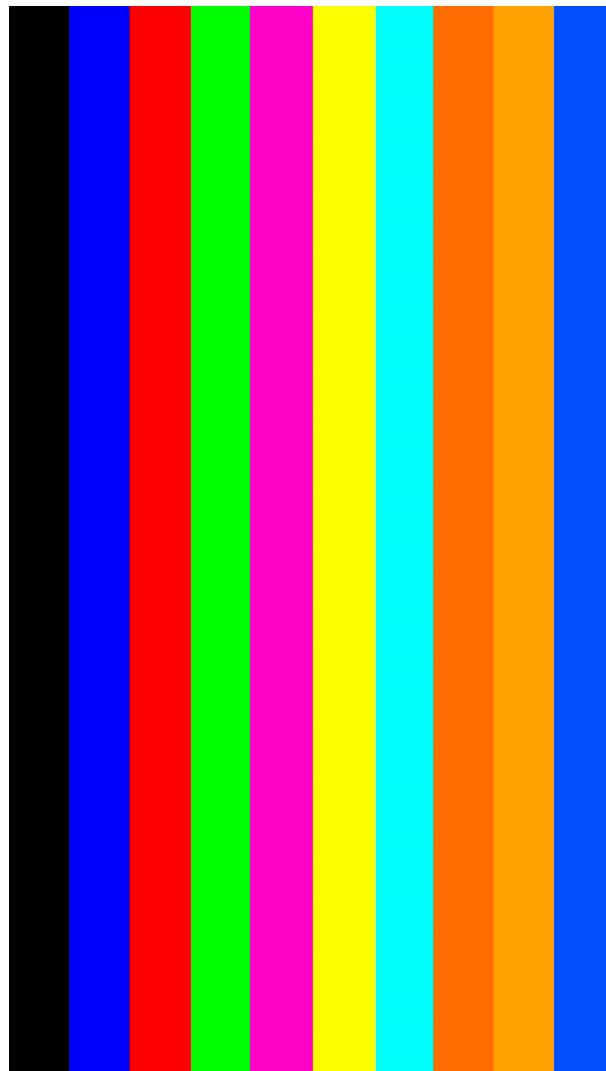


Figura 4: High Quality

Efectivamente a primera vista parecería que todas dieran lo mismo. Pero veamos que si le hacemos zoom, no es así.

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’, 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.

	<i>PSNR</i>
<i>quality</i>	39,67
<i>bilineal</i>	41,14
<i>directional</i>	48,13
<i>vecinos</i>	048,13

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

	<i>Tiempo(segundos)</i>
<i>quality</i>	64,88
<i>bilineal</i>	63,00
<i>directional</i>	57,82
<i>vecinos</i>	10,31

Como era de esperarse, las proporciones 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.

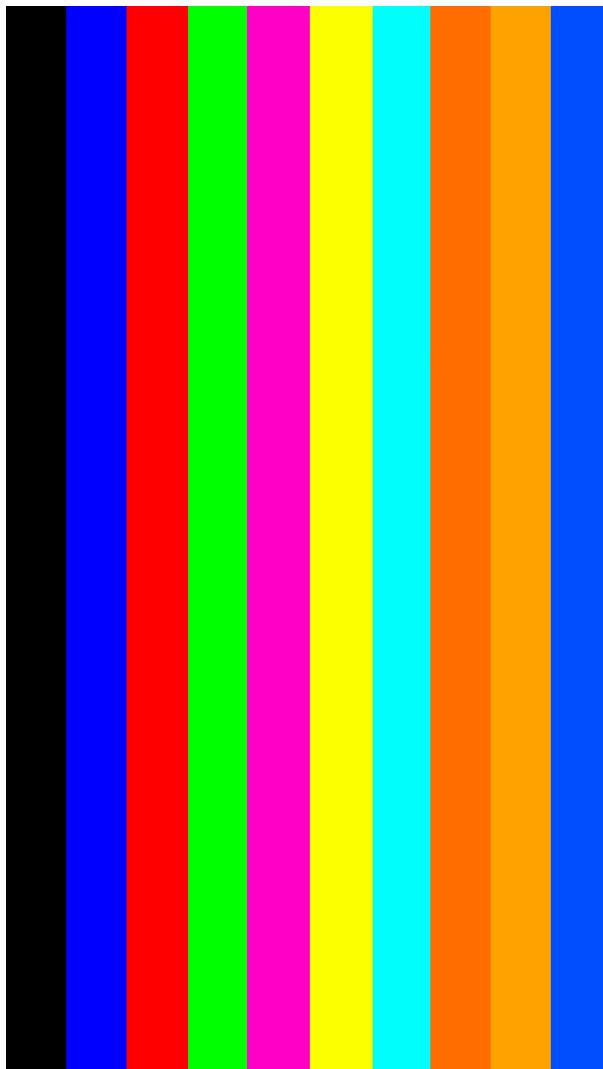


Figura 5: Directional

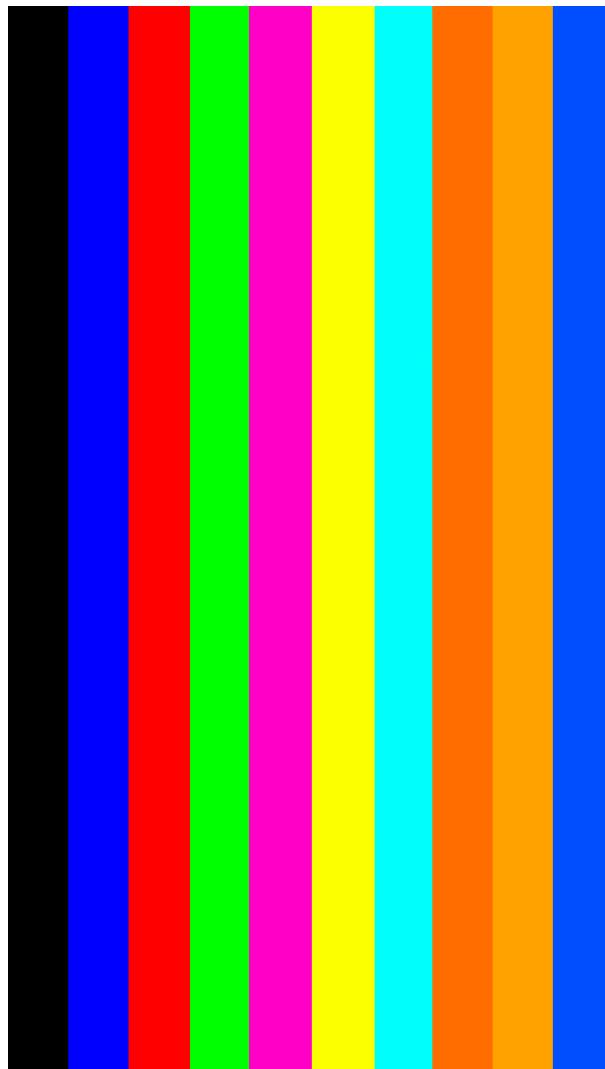


Figura 6: Vecinos

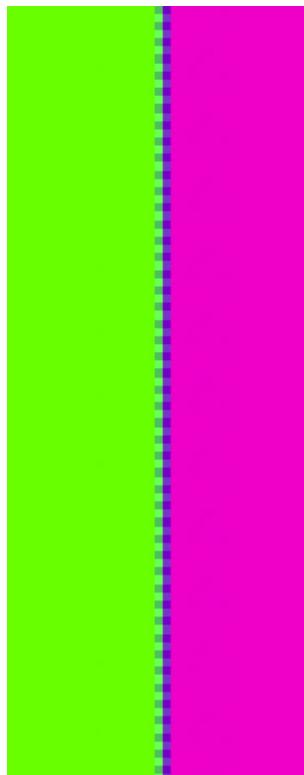


Figura 7: Bilineal Zoom

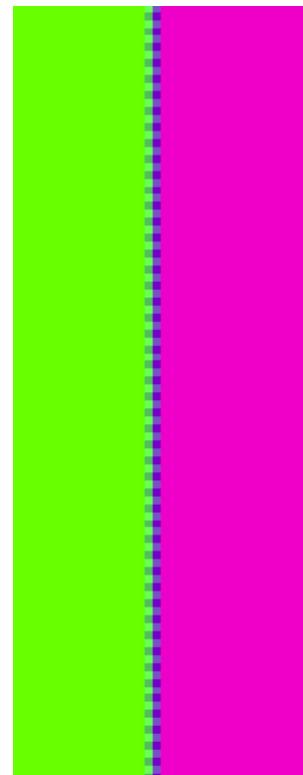


Figura 8: Bilineal Zoom

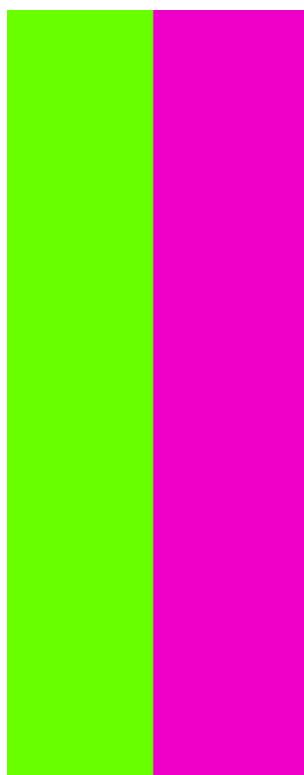


Figura 9: Directional Zoom

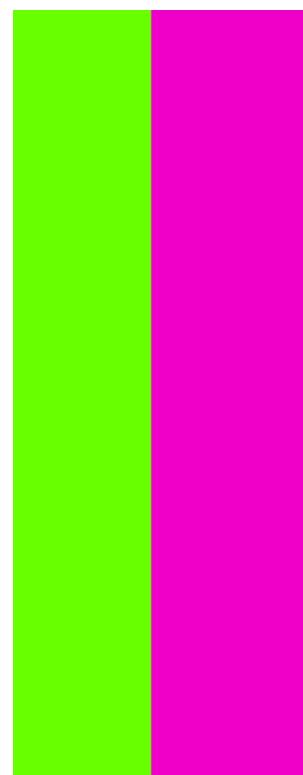
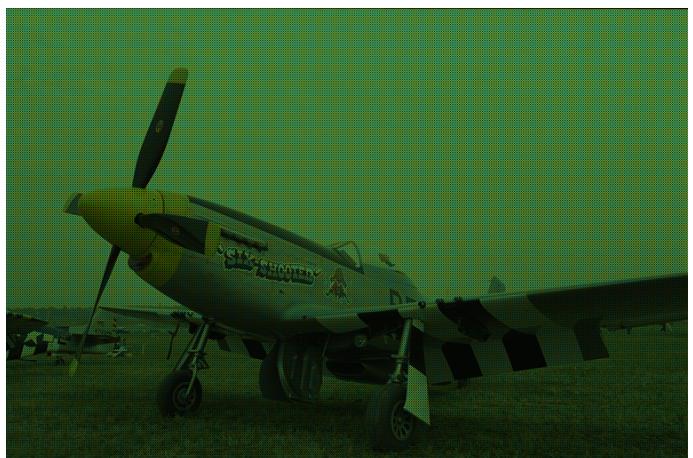


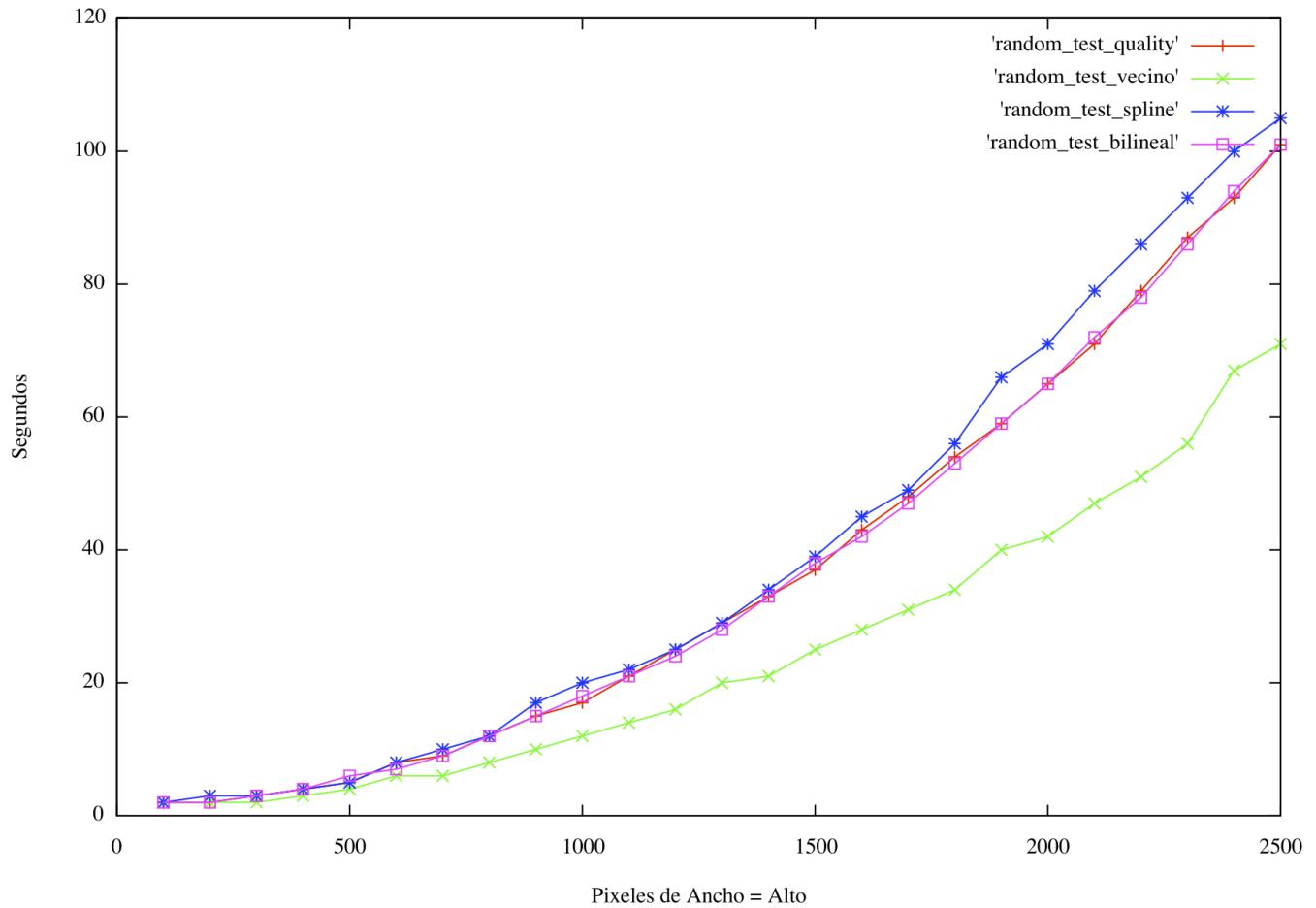
Figura 10: Vecinos Zoom

4.3. Imágen 9



4.4. 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 imágenes cuadradas generadas aleatoriamente que van aumentando su tamaño, y creemos que es una forma correcta de comparar los tiempos ya que la complejidad de cada algoritmo es independiente de la información que tenga la imagen, pero dependiente del tamaño.



4.5. 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.

Notemos que Spline si bien tendría que tener un valor mayor que bilineal 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.

5. Conclusiones

Referencias

- [1] R.Burdeny J.D.Faires, *Analisis numerico*, International Thomson Editors, 1998
- [2] Ron Kimmel. Demosaicing : image reconstruction from color ccd samples. *IMAGE PROCESSING, IEEE TRANSACTIONS ON*, 1999.