DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 3

Marche un telebeam Don Niembraaaaaa...”

Métodos Numéricos
Primer Cuatrimestre de 2015

Integrante	LU	Correo electrónico
Gastón Zanitti	058/10	gzanitti@gmail.com
Ricardo Colombo	156/08	ricardogcolombo@gmail.com
Dan Zajdband	144/10	Dan.zajdband@gmail.com
Franco Negri	893/13	franconegri200@gmail.com
Alejandro Albertini	924/12	ale.dc@hotmail.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. Introducción

En el presente trabajo práctico nos encargaremos de analizar el problema de la interpolación de polinomios mediante distintos métodos. Para ello se nos ofrece como marco la necesidad de realizar zoom a distintas imágenes, con el fin de crear un prototipo que permita decidir en tiempo real si una pelota entró o no dentro de un arco de fútbol.

Nuestro objetivo es entonces, dada una imagen de nxm pixeles de tamaño original y un número natural k que denota la cantidad de filas y columnas que se quieren agregar entre cada pixel, encontrar la forma más óptima de rellenar estos valores.

Presentaremos entonces tres técnicas a detallar con sus respectivas ventajas y desventajas:

1. Vecinos
2. Interpolación bilineal
3. Interpolación por splines

Además, con el fin de poder realizar un análisis cuantitavo sobre cada método, consideraremos dos medidas que comparan las imágenes originales contra sus transformadas, ofreciendo una noción de error o ruido:

1. Error cuadrático medio (ECM)
2. Peak to signal noise ratio (PSNR)

Por último, nuestra visión preliminar del problema nos indica que si bien es posible que los tres algoritmos retornen buenos resultados para valores de k relativamente bajos ($k = 1$ y quizas hasta 2) y la versión de vecinos sea mucho más eficiente temporalmente, es probable que esta deje de ser útil muy rápidamente a medida que crece k. Por el contrario, tanto la interpolación bilineal como la interpolación por splines deberian funcionar mejor para valores de k grandes, en el caso de esta última, siendo la que menos ruido introduzca (por la necesidad de que el polinomio sea derivable en cada intersección, suavizando el cambio de uno a otro).

2. Desarrollo

A continuación presentaremos el desarrollo de los experimentos que exploran las técnicas mencionadas en la intriducción con su respectivo análisis:

2.1. Vecinos

El primer método que analizaremos será el de vecinos. Este consiste en rellenar los valores de cada una de las columnas nuevas de la imagen replicando su valor más próximo. La principal ventaja de este método es la simpleza de su implementación, que consta únicamente de dos bucles para iterar la matriz original. Dicha característica también le provee de una eficiencia del orden de O(n^2) siendo n el alto y ancho de la imagen destino.

TP3 1 void vecinos(Matriz *image, Matriz *imageRes , int k)

```
1: for 0 to imageRes→rows - 1 do
2:     for 0 to imageRes→cols - 1 do
3:         imageRes→at(i, j) = image→at(round(i/(k+1)), round(j/(k+1)))
4:     end for
5: end for
```

Como se mencionó anteriormente, a pesar su alta eficiencia temporal en comparación a los demás métodos, nuestra intuición nos dice que este va ser el que presente una mayor cantidad de ruido, debido a que simplemente se están replicando los píxeles de las imágenes. Como resultado, teniendo en cuenta que lo único que se logra es .ªñadir grosor.ª los píxeles, deberían conseguirse imágenes de mayor tamaño pero con una ganancia igual de rápida en los valores de ruido a medida que los valores de k aumentan.

2.2. Interpolación bilineal

El segundo método que analizaremos será el de interpolación bilineal. En este caso, la idea consiste en generar un polinomio entre dos puntos consecutivos de la imagen para, por medio de este, calcular los valores necesarios para la extensión.
Primero realizaremos el cálculo por filas y una vez calculados estos valores, repetiremos el mismo procedimiento por columnas. Sean entonces Q_{11}, Q_{12}, Q_{21}, Q_{22} los cuatro puntos de la imagen original sobre los que queremos interpolar, el objetivo es conseguir un polinomio P que valga lo mismo en cada uno de estos puntos y aproxime los nuevos valores intermedios. Usaremos entonces para esto el polinomio interpolador de Lagrange.

Interpolando entonces en el eje X obtenemos la siguiente fórmula:

$$f(x, y_1) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21})$$
$$f(x, y_2) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22})$$

Ahora, realizando el mismo procedimiento pero en el eje Y, obtenemos lo siguiente:

$$f(x, y) \approx \frac{y_2 - y}{y_2 - y_1} f(x, y_1) + \frac{y - y_1}{y_2 - y_1} f(x, y_2)$$

Si notamos, los puntos que acompañan a las bases polinómicas de Lagrange son los mismos que calculamos sobre el eje X, por lo que podemos realizar el remplazo para llegar a una fórmula cerrada:

$$f(x,y) \approx \frac{y_2-y}{y_2-y_1}\left(\frac{x_2-x}{x_2-x_1}f(Q_{11})+\frac{x-x_1}{x_2-x_1}f(Q_{21})\right)+\frac{y-y_1}{y_2-y_1}\left(\frac{x_2-x}{x_2-x_1}f(Q_{12})+\frac{x-x_1}{x_2-x_1}f(Q_{22})\right)$$

Distribuyendo los valores dentro de los paréntesis, obtenemos la ecuación final

$$\begin{aligned} f(x,y) = & \frac{1}{(x_2-x_1)(y_2-y_1)}\left(f(Q_{11})(x_2-x)(y_2-y)+f(Q_{21})(x-x_1)(y_2-y)\right. \\ & \left.+f(Q_{12})(x_2-x)(y-y_1)+f(Q_{22})(x-x_1)(y-y_1)\right) \end{aligned}$$

Notar que se obtiene el mismo polinomio interpolador tanto si se hace primero sobre el eje X y luego sobre el Y, como a la inversa. Ahora, gracias a este polinomio interpolador, podemos conseguir los valores de las posiciones (x, y) que agregamos a nuestra imagen para realizar el zoom.
Para facilitar la lectura y escritura del ejercicio vamos a definir una estructura que se llama punto. Dentro de la misma vamos a tener 3 valores, el primero es el valor en x, el segundo en y y el tercero un valor que sera de la imagen original. En el siguiente codigo las variables $q11$, $q12$, $q21$, $q22$ son del tipo descripto anteriormente y se utilizan para definir los 4 puntos en los cuales se va a realizar el polinomio interpolador, la matriz A representa la imagen original y la matriz Res representa la imagen extendida.

TP3 2 void bilineal(matriz A, vector Res,int k)

1: Para i= $0...CantFilas-1$
2: Para j= $0...CantColumnas-1$
3: q11 = $< 0, 0, A_{i,j} >$
4: q12 = $< 0, k+1, A_{i,j+1} >$
5: q21 = $< k+1, 0, A_{i+1,j} >$
6: q22 = $< k+1, k+1, A_{i+1,j+1} >$
7: Para x=$0...k+1$
8: Para y=$0...k+1$
9: valorRes = poliniomioInterpolador(q11,q12,q21,q22,x,y)
10: $Res_{i*(k+1)+x,j*(k+1)+y}$ = valorRes

Donde *polinomiointerpolador* es la función que se encarga de generar el polinomio interpolador en el punto, de la siguiente manera:

TP3 3 void polinomioInterpolador(punto q11,punto q12, punto q21, punto q22, int x, int y)

```
1: denominador = 1/ ((q22.x-q11.x)* (q22.y-q11.y))
2: numerador1= q11.valor* (q22.x-res.x)*(q22.y-res.y) + q21.val * (res.x-q11.x)*(q22.y-res.y)
3: numerador2= q12.valor* (q22.x-res.x)*(res.y-q11.y) + q22.val * (res.x-q11.x)*(res.y-q11.y)
4: retorno ((numerador1+numerador2)*denominador)
```

Para el caso del polinomio interpolador se agregaron 2 lineas al final de la rutina, las cuales aplican saturación en caso de ser necesario cuando luego de realizar todas las cuentas el valor que nos queda es mayor a 255, fijandolo en 255, y cuando el valor es menor a 0, fijandolo en 0.

En este método, comparado con el anterior que solo replicaba píxeles vecinos, se está calculando un polinomio para tratar de introducir cierto nivel de suavidad entre los puntos de la imagen original a medida que se recorren los píxeles. Un dato importante a tener en cuenta es que dado que el grado del polinomio aumenta a medida que la cantidad de puntos a interpolar es mayor, decidimos que esta se realice entre solo dos puntos de la imagen original, para ofrecer un mejor desempeño entre puntos, haciendo que el polinomio calculado sea mas operativo y evitando que este oscile demasiado (situación conocida como Fenómeno de Runge[1]). Como contrapartida de esta optimización, es necesario recalcular el polinomio interpolador para todo par de puntos entre la imagen pero teniendo en cuenta que no se trabaja con imágenes de una definición desmesurada, es un costo que se puede pagar en detrimento de los beneficios obtenidos.

2.3. Interpolación por Splines

Por último nos centraremos en la interpolación por splines. Este método, similar al anterior, requiere el cálculo de Splines (al igual que antes, por filas y luego por columnas) para obtener los valores de los casilleros a extender. Además, en este caso, dado que la interpolación por medio de splines trata de generar polinomios para un segmento específico de la imagen, realizaremos un análisis sobre el algoritmo de interpolación por splines para tratar de obtener el tamaño de ventana más óptimo. Dado que este algoritmo solo incluye los valores de un recuadro de tamaño específico, creemos que agregar más valores puntos a considerar por el spline, no presentará beneficio alguno porque se estarían dejando de lado los valores de la imagen externos al punto a calcular (que estarían influyendo sobre un polinomio que intenta interpolar valores posiblemente lejos de los suyos).

Sea $S_j(x) = a_j + b_j(x - x_j) + c_j(x - x_j)^2 + d_j(x - x_j)^3$ nuestro polinomio interpolador para cada j intervalo entre 0 y $n - 1$, necesitamos entonces resolver los coeficientes. Utilizamos la construcción de Splines despejando estos últimos en función de c_j para formar el siguiente sistema de ecuaciones $Ax = b$:

[1] `https://es.wikipedia.org/wiki/Fenomeno_de_Runge`

$$A = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ h_0 & 2(h_0+h_1) & h_1 & & \\ 0 & h_1 & 2(h_1+h_2) & h_2 & \\ \vdots & \ddots & \ddots & \ddots & 0 \\ & & h_{n-2} & 2(h_{n-2}+h_{n-1}) & h_{n-1} \\ 0 & \cdots & 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{x} = \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix}$$

$$\mathbf{b} = \begin{bmatrix} 0 \\ \frac{3}{h_1}(a_2 - a_1) - \frac{3}{h_0}(a_1 - a_0) \\ \vdots \\ \frac{3}{h_{n-1}}(a_n - a_{n-1}) - \frac{3}{h_{n-2}}(a_{n-1} - a_{n-2}) \\ 0 \end{bmatrix}$$

donde $h = x_{j+1} - x_j$. Para nuestra aplicación $h = k$ y se mantiene fijo, ya que la distancia entre los puntos de la nueva imagen es igual a k. Una vez resuelto este sistema y con los valores de $c_0, ..., c_n$ ya calculados, podemos despejar los coeficientes que necesitabamos en base a las siguientes ecuaciones (que se derivan de las condiciones del mismo spline):

a_j es el valor del pixel en la posición j de la imagen original en la fila o columna iterada por el spline

$b_j = \frac{1}{k}(a_{j+1} - a_j) - \frac{k}{3}(2c_j + c_{j+1})$

$d_j = \frac{c_{j+1} - c_j}{3k}$

La implementación del algoritmo bicúbico consta de dos partes muy similares, ya que primero se recorre la matriz por columnas para realizar el método de splines y luego se realiza el metodo por filas para completar la matriz. Definimos para él una clase llamada spline donde almacenaremos los arreglos *as*, *bs*, *cs* y *ds*, estos contienen los coeficientes del polinomio para la posición i de la fila o columna en donde estemos calculando los splines.

La función *saturar* hace que si el valor es mayor a 255 o menor a 0 los fija en esos dos valores respectivamente.
Como mencionamos anteriormente se puede ver que de las lineas 1-8 se realiza el splines por columnas, luego en las lineas 9-16 se realiza el splines por filas, por lo tanto solo analizaremos el primer bloque (lineas 1- 8) para el siguiente es análogo.
En la linea 3 se llama al algoritmo de splines pasandole los n puntos de la imagen con el cual vamos a calcular coeficientes para cada polinomio, una vez obtenido esto se recorren los k puntos entre cada par de pixeles de la imagen resultante y se evalua el polinomio en ese punto, logrando asi el valor de cada punto en la imagen resultante.

Este algoritmo fue tomado del Burden 9na edicion, en el cual estamos calculando los coeficientes del poliniomio dado un arreglo de elementos que van a ser nuestros puntos.

TP3 4 void bicubico(matriz A, vector Res,int k)

1: Para i= $0..CantFilas - 1$
2: Para j= $0..CantColumnas - 1$
3: $Splinespline = calcularSpline(CantColumnas, columna_j, k)$
4: Para j= $0..CantColumnas - 1$
5: Para l= $0..k + 1$
6: valor= $spline.a[i] + spline.b[i] * l + spline.c[i] * j^2 + spline.d[i] * j^3$
7: saturar(valor)
8: $Res_{i*(k+1),j*(k+1)+l} = valor$
9: Para i= $0..CantColumnas - 1$
10: Para j= $0..CantFilas - 1$
11: spline= $calcularSpline(CantFilas, fila(j), k)$
12: Para j= $0..CantFilas - 1$
13: Para l= $0..k + 1$
14: valor= $spline.a[i] + spline.b[i] * l + spline.c[i] * j^2 + spline.d[i] * j^3$
15: saturar(valor)
16: $Res_{j*(k+1)+l,i)} = valor$

TP3 5 spline calcularSpline(int cant,arreglo(int) pixelesOriginales,int k)

1: arreglo alfa[cantColumnas]
2: Para j= $0..Cant - 1$
3: $alfa_j = (3/k) * (pixelesOriginales_{j+1} - pixelesOriginales_j) - (3/k) * (pixelesOriginales_j - pixelesOriginales_{j-1})$
4: arreglo(float) ln ,cn , zn
5: l[0]=1,c[0]=0,z[0]=0
6: Para i= $0..Cant - 1$
7: $l_i = 2 * (2 * k) - k * c_{i-1}$
8: $c_i = k/l_i$
9: $z_i = alfa_i - (k * z_{i-1})/l_i$
10: arreglo(int) as,bs,cs,ds
11: Para i= $0..Cant - 1$
12: $cs_i = z_i - c_i * cs_{i+1}$
13: $bs_i = (as_{i+1} - asi)/k - k * (cs_{i+1} + 2 * cs_i)/3$
14: $(cs_i + 1] - cs[i])/(3 * k)$
15: devolver spline(as,bs,cs,ds)

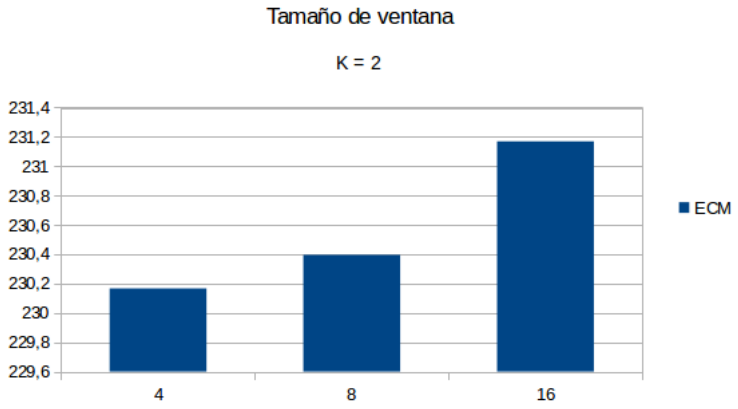
Este método, que podría considerarse un refinamiento del anterior, introduce la particularidad de que se le pide al polinomio interpolador que las intersecciones de las funciones que interpolan al punto $n-1$ y n y al n y al $n+1$, sean derivables. Como resultado, se agrega mucha más suavidad entre puntos que con la técnica anterior que solo respetaba que las funciones empiecen y terminen en el mismo punto (dando lugar a posibles picos, como en el caso de que dos puntos se interpolen con una recta ascendente y los siguientes con una descendente). Además, esta técnica evita de forma natural la oscilación del polinomio mencionada en el método anterior dado que siempre se toman polinomios por partes. Gracias a esto, cuando se intenta reducir el error de interpolación se puede incrementar el número de partes del polinomio que se usa para construir el spline, en lugar de incrementar su grado.

3. Análisis

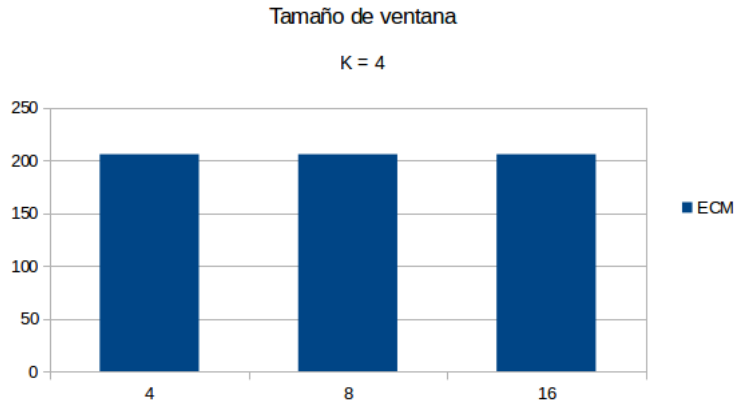
A continuación se presenta un análisis comparativo de los tres métodos implementados. Cabe mencionar que, dado que la experimentación requería que ambas imágenes (original y modificada) tengan el mismo tamaño para poder realizar un análisis cuantitativo de los algoritmos mediante las medidas de comparación que se mencionaron en la introducción, decidimos achicar la imagen original mediante un programa de edición de imágenes para luego agrandarla mediante nuestros métodos y poder comparar los resultados obtenidos con la imagen original.

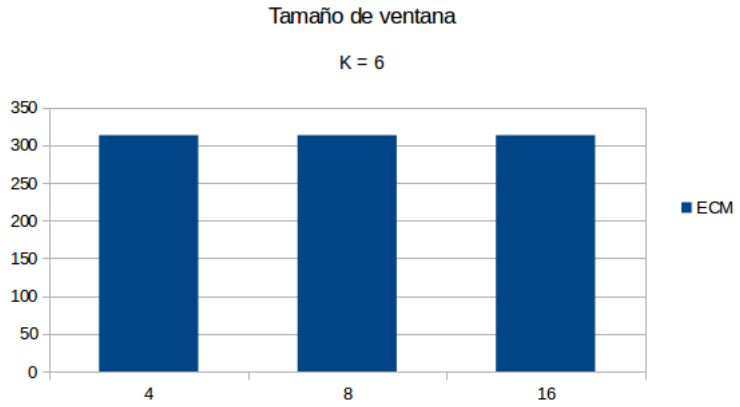
3.1. Ventana óptima para método de Splines

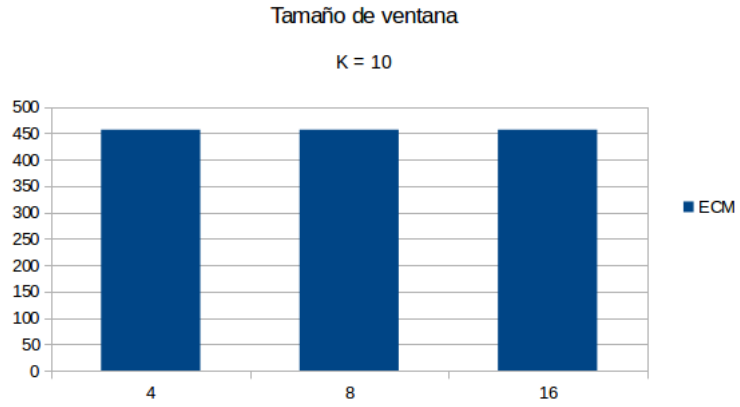
Nuestro primer análisis se encargará de encontrar un valor óptimo para la cantidad de las ventanas utilizadas en el método de splines. Para dicho fin, elegimos correr varias instancias del método con valores de K crecientes para distintos valores de ventana (4, 8, y 16 para poder comparar los resultados). Se presentan entonces los resultados obtenidos. Vale la pena destacar que no se muestra información respecto al PSNR debido a que presentaba exactamente el mismo comportamiento y no ofrecia informacion extra alguna.
Como habiamos presupuesto en la introducción, agrandar el tamaño de la ventana solo hace que se tengan en cuenta valores para el punto que se quiere calcular que no depende directamente de este.



Como puede verse a continuación, agrandar el tamaño de la ventana deja de presentar benificio alguno para valores mas altos de K porque los resultados se veulven constantes:







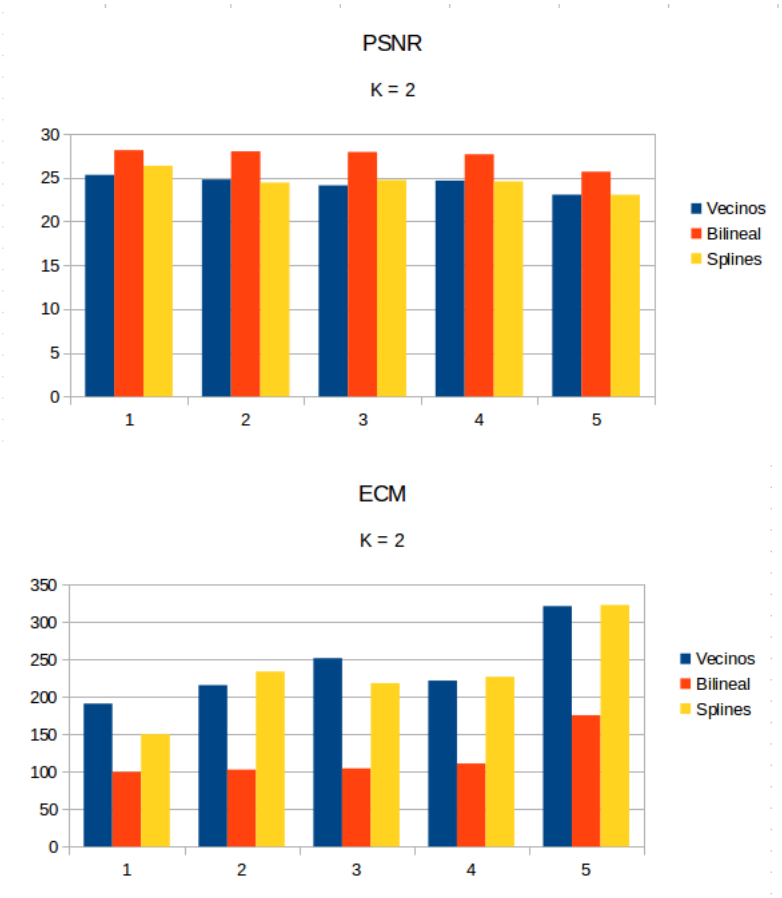
A partir de este punto, el algoritmo de splines utiliza una ventana de tamaño cuatro, dado que es la que presenta mejores resultados. De todos formas, no es cierto que siempre sea preferible una ventana más chica a una que incluya mas puntos, porque en problemas donde se quieren calcular por ejemplo trayectorias, es deseable considerar mas puntos para tener mas información.

3.2. Análisis de los metodos

Empleamos un análisis incremental respecto al valor de los pixeles intermedios introducidos (el valor de k) para poder analizar los distintos métodos de forma escalonada y presentar conclusiones mucho más claras. Dado que nuestros algoritmos solo funcionan cuando los valores de las imágenes son divisibles por k, no deben asumirse una correlación entre los distintos valores de este debido a que las imágenes a analizar no siempre pudieron ser las mismas.

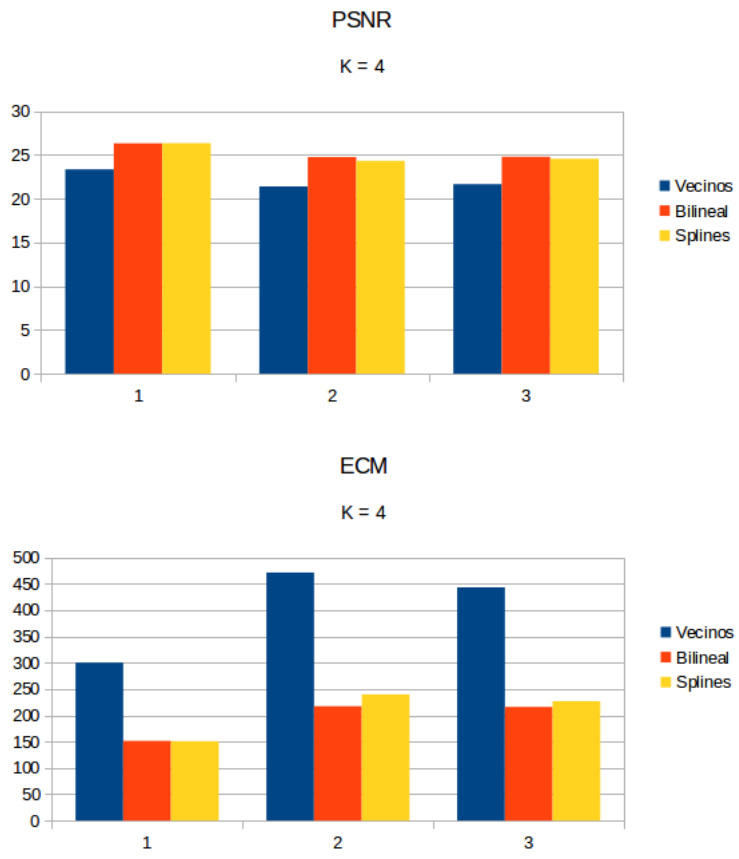
3.2.1. K = 2

Nuestro primer análisis se concentra en el valor mínimo de k para el cual esperabamos que el comportamiento de los tres métodos se mantenga bastante estable. Nuestra intuición proviene de la idea de que todos ellos ofrecian una perdida en la calidad de la imagen bastanta pequeña en relación al zoom pedido. Como podemos apreciar en los gráficos a continuación, nuestra intuición se corresponde con los valores de $PSNR$, donde los tres métodos se comportan relativamente iguales, sin embargo, nos sorprende ver que para el error cuadrático medio (y para $PSNR$ también, pero en menor medida) la técnica de interpolación Bilineal obtuvo resultados muy destacables (de hecho, casi constantes), incluso frente a la técnica de Splines que esperabamos siempre tenga un mejor rendimiento. La conclusión respecto de este fenómeno se explica al final del artículo.



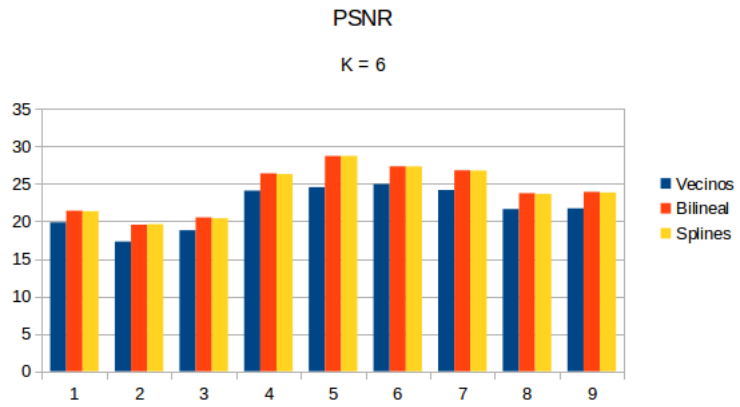
3.2.2. K = 4

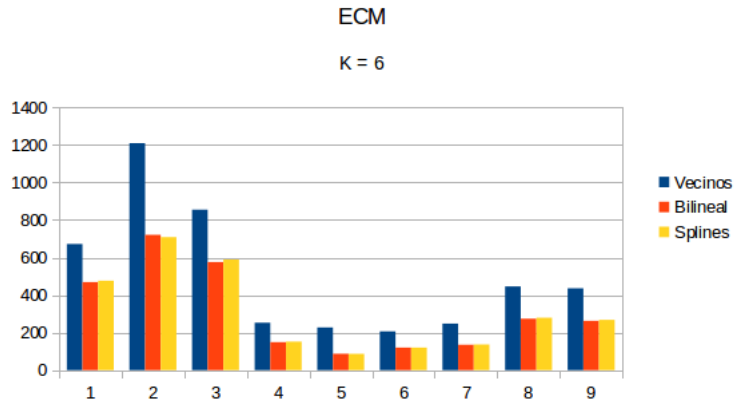
Para este segundo caso es cierto que, como esperabamos, el error cuadrático medio del método de los vecinos se dispara rápidamente mientras que los de interpolación bilineal y splines se mantienen prácticamente iguales. Lo mismo sucede para los valores de PSNR, siendo los del método de vecinos los únicos que disminuyen con una diferencia de casi 5 puntos.



3.2.3. K = 6

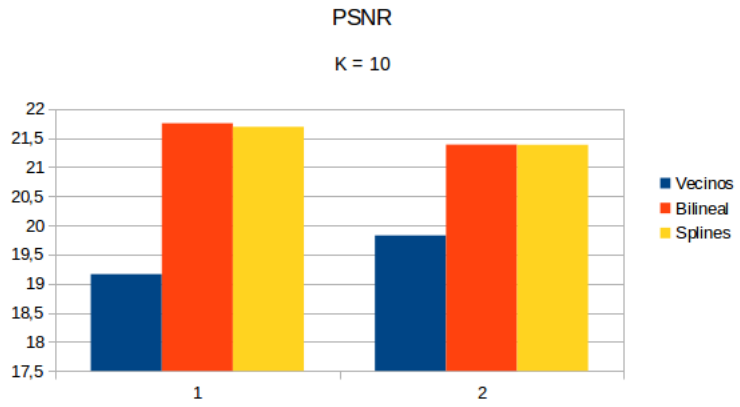
Entrando en valores de k mucho más elevados, nuestro análisis empezó a dejar de coincidir con lo que creiamos en un primer momento serían los resultados finales, debido a que el método de Splines no logró sacar una diferencia notoria frente al de interpolación Bilineal, sino que incluso ambos métodos se mantuvieron prácticamente constantes. Al momento de obtener los resultados nos llamaron poderosamente la atención los valores de ECM obtenidos para las tres primeras imágenes, pero luego de hacer un análisis en conjunto de estas, llegamos a la conclusión de que la suba desmesurada en estos valores se debe a la elevada variabilidad de los contrastes en la escena (el interior de un hogar muy decorado, la foto aérea de un barrio, etc).

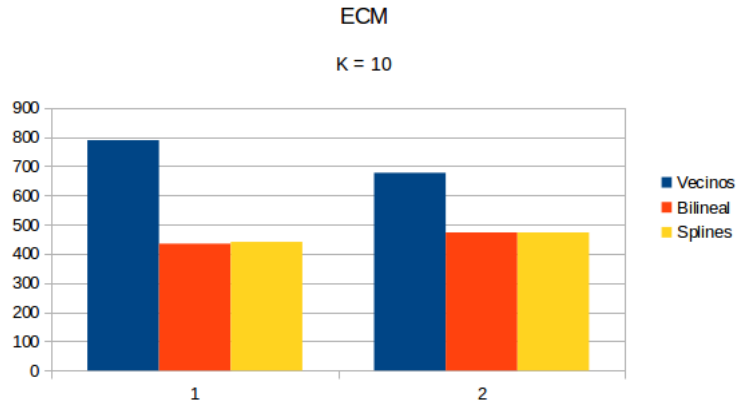




3.2.4. K = 10

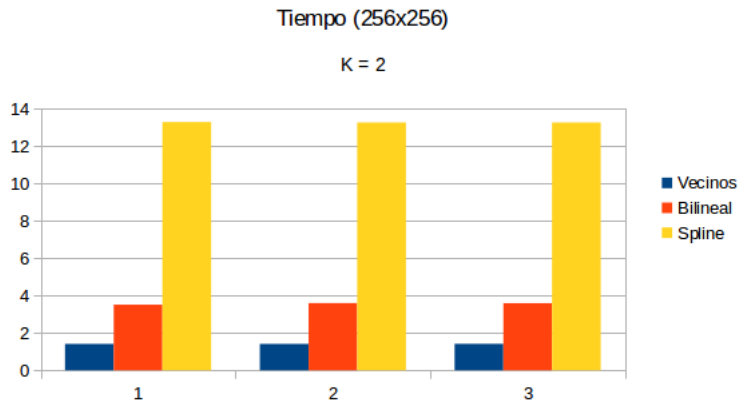
Por último, para valores que ya se consideran altos de k el método de interpolación Bilineal todavía sigue desempeñandose igual o incluso a veces levemente mejor que el de Splines. Esto no solo contradice nuestra intuición, sino que debido a la performance de ambos, estos resultados colocan al método de interpolación como el más apto en relación benenficio/tiempo, muy por encima del de Splines (ambos ya muy por encima del método de vecinos a esta altura).

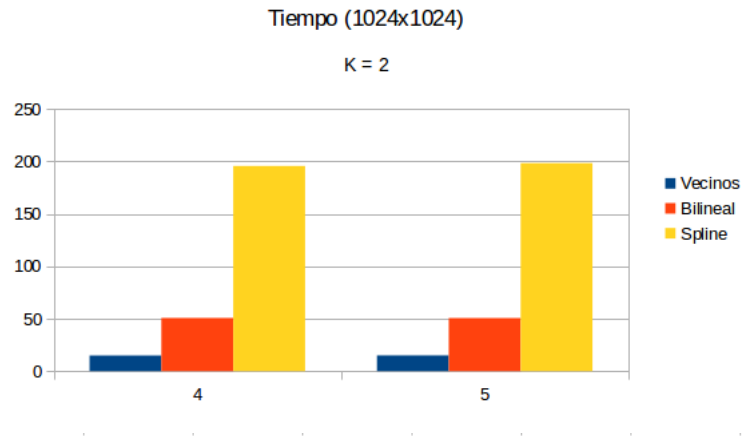


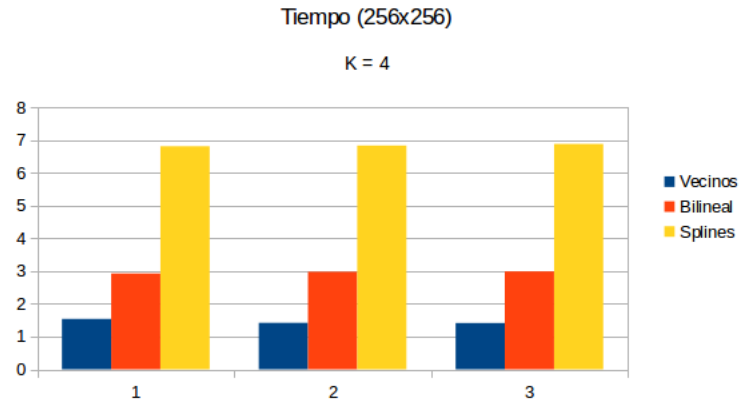


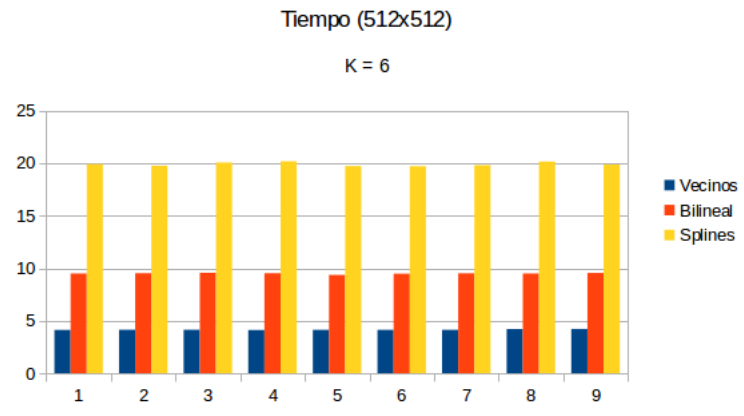
3.3. Análisis de tiempos

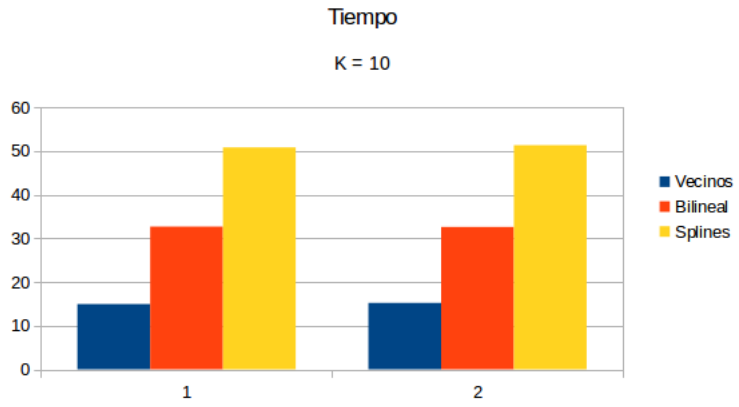
El análisis de tiempo, a diferencia del de los métodos, no ofreció ninguna respuesta que no hayamos podido intuir durante la codificación de los algoritmos. Es claro que a medida que el método se perfecciona en la búsqueda de resultados más suaves, también aumenta el tiempo necesario de cálculo.


Tiempo (256x256)
K = 2
14
12
10
8
6
4
2
0
1
2
3
Vecinos
Bilineal
Spline


Tiempo (1024x1024)
K = 2
250
200
150
100
50
0
4
5
Vecinos
Bilineal
Spline

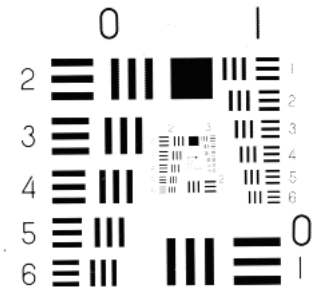

Tiempo (256x256)
K = 4
8
7
6
5
4
3
2
1
0
1
2
3
Vecinos
Bilineal
Splines


Tiempo (512x512)
K = 6
25
20
15
10
5
0
1
2
3
4
5
6
7
8
9
Vecinos
Bilineal
Splines

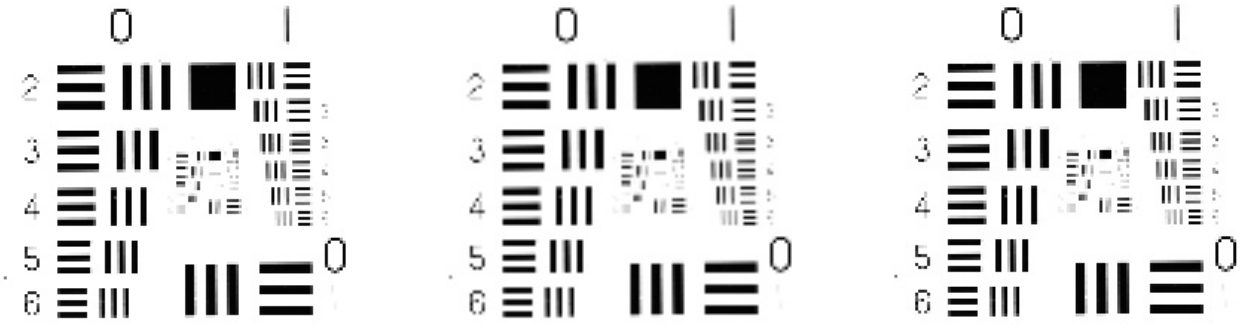


3.4. Analisis De Los metodos Para Imagenes Con Simbolos Alfanumericos

En esta sección analizaremos los tres algoritmos sobre imagenes con simbolos alfanumericos. Para ellos usamos la imagen mostrada mas abajo para la cual aplicaremos los tres metodos con diferentes ks.



Primero lo hacemos para $k = 1$, se obtiene esto:



(a) Metodo 1 (b) Metodo 2 (c) Metodo 3

Figura 1: Comparacón de metodos para $k = 1$

Los artifact (Errores visuales) que vemos aquí son, bla bla bla (COMPLETAR!)
Ahora lo hacemos para $k = 2$, se obtiene esto:

(a) Metodo 1 (b) Metodo 2 (c) Metodo 3

Figura 2: Comparacón de metodos para $k = 2$

Los artifact que vemos aquí son, bla bla bla (COMPLETAR!)
Ahora lo hacemos para $k = 4$, se obtiene esto:

(a) Metodo 1 (b) Metodo 2 (c) Metodo 3

Figura 3: Comparacón de metodos para $k = 4$

Los artifact que vemos aquí son, bla bla bla (COMPLETAR!)

4. Conclusiones

Como se mencionó con anterioridad, nuestra intuición relacionaba fuertemente a los métodos en cuanto a calidad/desempeño. Como se pudo ver a lo largo del análisis realizado, el método de splines nunca logró sacar una diferencia significativa respecto al método de interpolación bilineal. También se puede apreciar como este último tiene un mejor desempeño temporal en todos los casos. Esto coloca a la interpolación bilineal como la mejor opción en cuanto a tiempo y calidad, dado que la ganancia por splines es minima respecto al tiempo extra. En un momento ($K = 2$), nos llamó poderosamente la atención que el método bilineal haya obtenido un mejor resultado que el método de splines, pero teniendo en cuenta como se terminaron comportando ambos métodos a lo largo de todo el análisis, ahora ya no parece un resultado tan anómalo. Sin embargo, no logramos llegar a una conclusión que justifique el porque de una diferencia tan significativa.

En cuanto al método de los vecinos, como bien dijimos al principio, se comporta relativamente bien para valores de k mínimos, pero enseguida que este crece, el método pierde fiabilidad.