Métodos Numéricos
TP3

10 de noviembre de 2015

Integrante	LU	Correo electrónico
Martin Baigorria	575/14	martinbaigorria@gmail.com
Federico Beuter	827/13	federicobeuter@gmail.com
Mauro Cherubini	835/13	cheru.mf@gmail.com
Rodrigo Kapobel	695/12	rok_35@live.com.ar

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Resumen: El siguiente trabajo práctico tiene como objetivo implementar, utilizar y evaluar diferentes métodos de interpolación (entre ellos, Lineal y Splines) para ser aplicados a la generación de videos en slow motion (o cámara lenta). Se abordarán los detalles relacionados a cada método, como por ejemplo, como afectan en la generación de artifacts y que caracteristicas poseen dependiendo del escenario analizado. Además, en base a la calidad del resultado (que será medida tomando el PSNR) y midiendo tiempos de ejecución, compararemos cada método entre si para luego concluir porque, que bajo ciertas condiciones generales, splines es el mejor para resolver este tipo de problematica. Todo esto será realizado bajo la temática de efectos de cámara utilizados en cine y television, analizando su comportamiento al aplicarles alguno de los métodos mencioandos.

Keywords: TODO

Índice

1. Introducción

En la practica, es sumamente comun encontrar situaciones donde se dispone de informacion discreta sobra un sistema o comportamiento en particular. En general, dada algun tipo de discretizacion, muchas veces se desconoce exactamente cual fue el procedimiento o modelo a partir del cual se derivaron estos datos. Aqui es donde los metodos de interpolacion son sumamente utiles para encontrar de alguna manera los valores correspondientes a todo el dominio sobre el cual se quiere trabajar a partir de datos discretos.

Existen diferentes métodos de interpolacion, cuya presicion en general dependera del proceso subyacente que genera los datos. Uno de los principales usos de la interpolación a lo largo de la historia ha sido el de hallar valores intermedios a los calculados en tablas trigonométricas o astronómicas. También puede encontrarse en matemática financiera para toma de desiciones empresariales, como por ejemplo al aproximar la funcion de payoff de una opcion a partir de contratos dados con precios de ejecucion discretos.

Otro tipo de problemas en donde interpolación tiene mucha aplicación es en el area de procesamiento de imágenes. Actualmente los televisores LCD o los últimos LED disponen de una mejor definición que la generación anterior. Esto abarca varios aspectos en la imagen obtenida. Citaremos las más importantes:

- Mayor resolución: Es decir mayor cantidad de pixeles en alto por ancho de la pantalla, logrando así un mayor detalle.
- Colores más nitidos.
- Mayor frecuencia de muestreo: o frame rate, que es la tasa de refresco de las imágenes o cuadros en pantalla. Se mide en herzios (hz) y funciona como cota superior para la cantidad de cuadros por segundo o fps (frames per second).

En cuanto a lo relacionado puramente con la resolución de pantalla podemos encontrar el caso particular de los videojuegos. Lo que sucede es que los antiguos televisores y sistemas de entretenimiento, vertían el vídeo a una resolución determinada. Todos los juegos antiguos se basaban en ella y se veían “definidos”. Al llegar FullHD o el HDReady, las consolas deben interpolar la imagen anterior y mucho más pequeña hasta otra más grande y acorde con la nueva área de visión. La técnica que utilizan se denomina resampling y se basa en copiar el pixel más cercano (nearest-neighbor interpolation). La misma puede variar en su implementación, pudiendo tomar solo un vecino o promediando todos. La elección de uno u otro determinará la graduación de los pixeles generados que tendrá la imágen final, logrando mayor suavidez con el método de los promedios y un acabado algo más iregular en caso contrario.

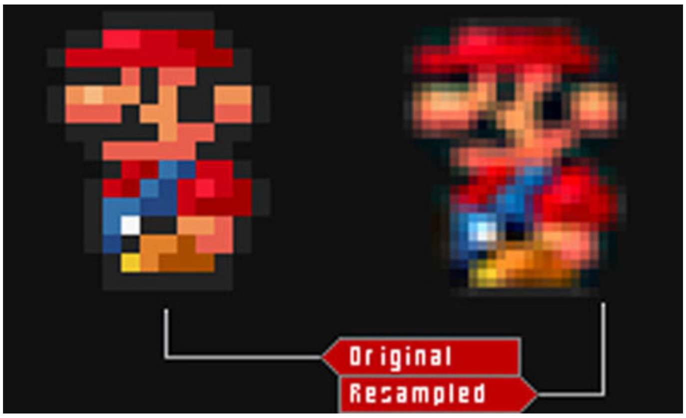

Figura 1: Aplicación de resampling a Mario Bros utilizando el promedio de los vecinos más cercanos.

Debido al alto frame rate del que dispone un televisor LED, es capaz de reproducir peliculas a más de 60 fps, logrando así mayor fluidez en la transición de imágenes. El inconveniente es que no todas las peliculas y series son grabadas a esta frecuencia, si no, a 24 fps que es el estándard historicamente para cine y television. Para poder solucionar este inconveniente los fabricantes de televisores incorporan algoritmos de interpolación que lo que realizan es doblar la cantidad de cuadros por segundo, generando cuadros intermedios, para obtener 60 fps.

Como puede verse en la Figura 2. Se dispone de dos cuadros de un video de un elefante en movimiento. El video fué grabado con pocos fps, por lo cual hay información inexistente. Como resultado, el movimiento del elefante pareciera ser menos fluido entre cuadro y cuadro. Para lograr una transición más suave, se genera un cuadro intermedio mediante interpolación entre el cuadro 1 y 2 obteniendo el cuadro 1a. Si agregaramos más cuadros, obtendriamos una transición aún más fluida, en principio, aunque en la práctica habrá factores que influirán en el resultado.

En particular, si agregamos varios cuadros intermedios y no modificamos los fps lograremos un efecto de slow motion o cámara lenta. Este mismo efecto es el que modelaremos y analizaremos en detalle en este trabajo práctico.

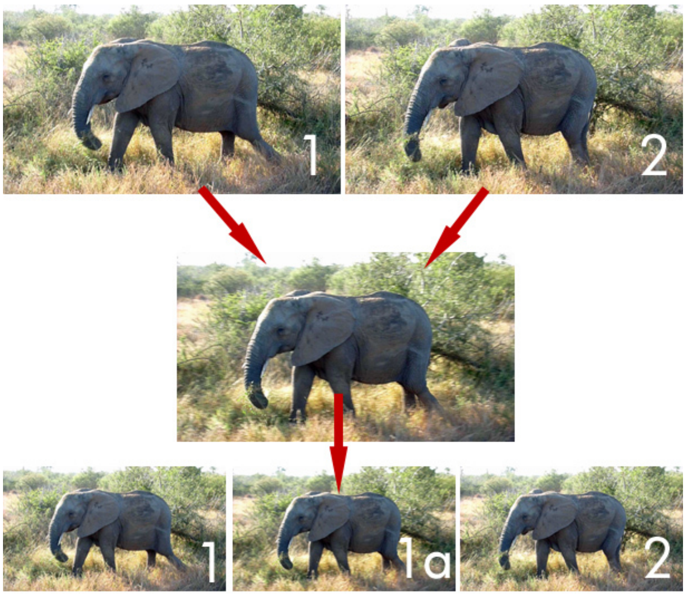

Figura 2: Elefante interpolado.

1.1. Motivación y objetivos

A lo largo de la historia del cine...

Para lograrlo, como se mencionó anteriormente, recurriremos a interpolación. En particular, nos enfocaremos en el estudio de la interpolación polinómica. La misma, como se verá en el desarrollo, se basa en aproximar una función, por un polinomio. Analizaremos tres métodos, los cuales serán: Lineal, Splines y Nearest Neighbour y compararemos cada uno de ellos entre si.

Luego abordaremos la generación de imágenes para obtener videos en slow motion. Plantearemos el procedimiento para llevarlo a cabo y observaremos como se comporta cada uno de los tres métodos en el contexto mencionado. Veremos los comportamientos en términos de la calidad del resultado, analizando el trade-off entre complejidad, eficiencia, suavidad y nitidez. Concluiremos que bajo ciertas condiciones en los datos de entrada, es decir como sea el video que estamos manipulando, utilizar splines será la mejor elección. Además estudiaremos que sucede con la generación de artifacts, así denominadas, a las distorciones procedentes de aplicar estos algoritmos.

2. Desarrollo

2.1. Interpolación

Interpolar consiste en estimar valores desconocidos en base a otros ya conocidos dados por un *data-set* predefinido, que suponemos sugiere el comportamiento de alguna función f para la cual solo conocemos una cantidad determinada de resultados. Estos datos suelen organizarse en tablas de pares (x_i, y_i), donde y_i es el $f(x_i)$ y x_i es algún valor conocido del dominio de f sobre el cual deseamos trabajar; dicho esto el problema entonces se traduce en aproximar dicha función, procurando satisfacer la obvia restricción de *matchear* con esos valores ya conocidos bajo sus respectivas pre-imagenes. Para ello existen diversos metodos algoritmicos que en base a nuestros datos intentan generar esa función aproximada, entre ellos por ejemplo, se encuentran los metodos de interpolación polinomial que como su nombre sugiere intentan lograrlo por medio de algún polinomio (al que llamamos polinomio interpolador).

2.2. Interpolación polinómica

Dada una función f de la cual se conocen sus valores en un número finito de puntos x_0, x_1, ..., x_m, con $m \in \mathbb{N}$, se llama interpolación polinómica al proceso de hallar un polinomio $P_m(x)$ de grado menor o igual a m, cumpliendo $P_m(x_k) = f(x_k)$, $\forall\ k = 0, 1, ..., m$. Este polinomio se conoce como polinomio interpolador y tendrá la siguiente forma:

$$P_m(x) = a_m x^m + a_{m-1} x^{m-1} + \cdots + a_1 x + a_0 \tag{1}$$

Donde a_m, a_{m-1}, ..., a_1, $a_1 \in \mathbb{R}$.

Este tipo de interpolación suele considerarse para aproximar funciones continuas, esto es asi porque los polinomios tiene buenas propiedades analiticas, entre las cuales podemos considerar las siguientes: son continuas, facilmente integrables e infinitamente derivables (pertenecen a la clase $C^\infty[a, b]$), y además sus derivadas y primitivas son a su vez polinomios. Otro propiedad importante de la cual se fundamenta este tipo de interpolación es la de que dada cualquier funcion definida y continua en un intervalo cerrado y acotado, existe un polinomio que aproxima tanto como se desee a esa función; formalmente esto es enunciado mediante el siguiente teorema.

Theorem 2.1

Weierstrass Aproximation Theorem

Supongamos que f es definida y continua en $[a, b]$. Para cada $\epsilon > 0$, existe un polinomio $P(x)$ con la siguiente propiedad:

$$|f(x) - P(x)| < \epsilon, \forall x \in [a, b] \tag{2}$$

Dado el anterior teorema podemos considerar a nuestra función f como algún polinomio interpolador más otra función de corrección a la que denominaremos la expresión del error. Esta última función correjirá la inprecisión del polinomio con respecto a la función original para cada valor de su dominio. No obstante puede que encontrar dicha expresion no sea para nada trivial.

2.3. Cálculo del polinomio interpolador

Existen varios métodos de interpolación polinomial que generan diferentes tipos de polinomios, para este trabajo práctico veremos en particular dos de ellos, Lineal y Splines. Veremos como se construye cada uno y analizaremos su exactitud y propiedades particulares.

2.3.1. Interpolación lineal

Este método consiste en aproximar una función f tomando rectas entre cada par de puntos consecutivos. Parece razonable que la aproximación será mejor cuanto más valores conozcamos. Podemos observar este detalle en los siguientes gráficos de la función $sen(2\pi(x/5))$

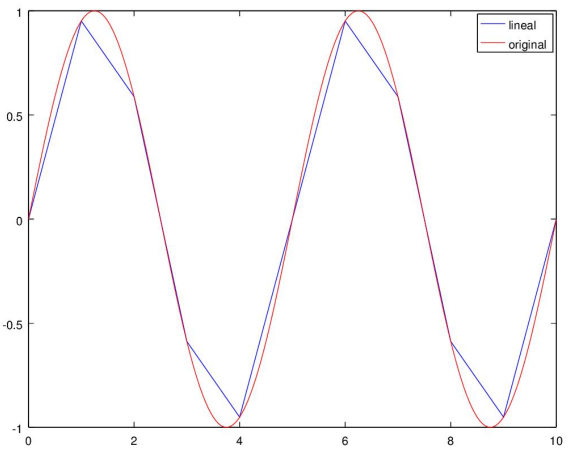

Figura 3: Lineal: Paso: 1.0

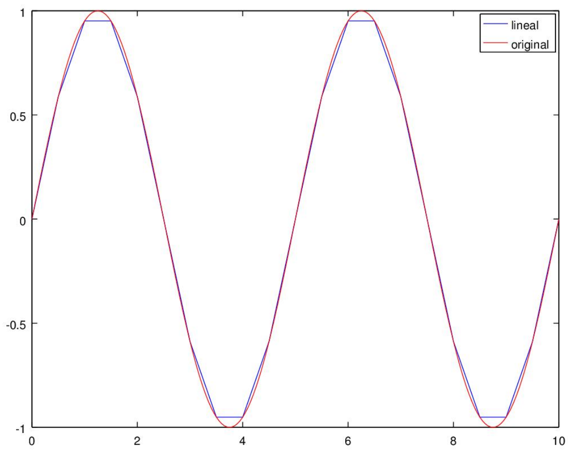

Figura 4: Lineal: Paso: 0.5

Como podemos apreciar, en la Figura 4 tomamos intervalos de la mitad del tamaño que en la Figura 3, al conocer más valores, el polinomio interpolador tiene más información de la función, lo cual hace que las rectas estén más pegadas a la función real y por lo tanto aproxime mejor para valores intermedios en cada intervalo.

Para cada par de puntos (x_i, y_i) y (x_{i+1}, y_{i+1}), la recta que pasa por ellos es nada más y nada menos que la recta secante a la función en esos puntos. La pendiente de la misma será:

$$m = \frac{y_{i+1} - y_i}{x_{i+1} - x_i} \tag{3}$$

Luego en la ecuación de la recta $y = m(x - x_i) + y_i$ podemos sustituir m y obtener:

$$y = P(x) = y_i + (y_{i+1} - y_i)\frac{y_{i+1} - y_i}{x_{i+1} - x_i} \tag{4}$$

Donde (4) es un polinomio de grado 1 y si evaluamos en x_i y x_{i+1} respectivamente obtenemos:

$$P(x_0) = y_i + (y_{i+1} - y_i)(0) = y_i \wedge P(x_{i+1}) = y_i + (y_{i+1} - y_i)(1) = y_{i+1} \tag{5}$$

Luego dados n puntos de la función podemos definir un polinomio como una función partida entre cada par de puntos para los cuales se obtendrá una recta que los interpole

$$P(x) = \begin{cases} y_0 + (y_1 - y_0)\dfrac{y_1 - y_0}{x_1 - x_0} & : x_0 \leq x < x_1 \\ \\ y_1 + (y_2 - y_1)\dfrac{y_2 - y_1}{x_2 - x_1} & : x_1 \leq x < x_2 \\ \vdots & \\ y_{n-1} + (y_n - y_{n-1})\dfrac{y_n - y_{n-1}}{x_n - x_{n-1}} & : x_{n-1} \leq x_i \leq x_n \end{cases}$$

Un detalle importante a tener en cuenta es que el polinomio definido de esta manera es continuo pero no es derivable en los extremos de los intervalos. Esto se traduce directamente en que la aproximación no es "suave". Esto se puede ver con facilidad en las Figuras 3 y 4 (2.3.1).

2.3.2. Spline Cúbico

Este metodo al igual que el anterior se basa en particionar nuestro intervalo de interes en subintervalos más pequeños con la intención de construir un polinimio particular para cada uno de estos. Se busca entonces que para cada subintevalo, su polinomio asociado aproximen a la función estudiada en el interior del mismo, y además coincida con los resultados ya conocidos en ambos extremos.

Nuestra partición a considerar será la inducida por $a = x_0 < x_1 \ldots < x_n = b$, es decir, las pre-imagenes ordenadas de menor a mayor de los valores ya conocidos de f; y nuestros polinomios serán polinomios cúbicos que cumplan ciertas propiedades, entre las cuales, que coincidan en los extremos con los polinomios de los intervalos vecinos, y además que en dichos punto también coincidan sus primeras y segundas derivadas. Como resultado obtenemos sobre todo el intervalo original una función continua y de derivadas primera y segunda también continuas, es decir de clase $C^2[a,b]$. No obstante, no necesariamente ocurre que estas derivadas concuerden con las de la función a interpolar, ni siquiera que las aproximen.

Expresando lo anterior más formalmente, dada una función f definida en $[a,b]$ y los puntos $a = x_0 < x_1 \ldots < x_n = b$, un spline cúbico S para f es un polinomio que satisface las siguientes condiciones:

1. $S(x)$ es un polinomio cúbico, denotado $S_j(x)$ en el intervalo $[x_j, x_{j+1}] \forall j = 0, 1, \ldots, n-1$
2. $S(x_j) = f(x_j)$ y $S_j(x_{j+1}) = f(x_{j+1})$ $\forall j = 0, 1, \ldots, n-1$
3. $S_{j+1}(x_{j+1}) = S_j(x_{j+1})$ $\forall j = 0, 1, \ldots, n-2$
4. $S^{'}_{j+1}(x_{j+1}) = S^{'}_j(x_{j+1})$ $\forall j = 0, 1, \ldots, n-2$
5. $S^{''}_{j+1}(x_{j+1}) = S^{''}_j(x_{j+1})$ $\forall j = 0, 1, \ldots, n-2$
6. Una de las siguientes condiciones necesita ser cumplida
   1) $S^{''}(x_0) = S^{''}(x_n) = 0$ (borde natural)
   2) $S^{'}(x_0) = f^{'}(x_0) \wedge S^{'}(x_n) = f^{'}(x_n)$ (borde sujeto)

Observemos que se necesita cumplir con una de las dos condiciones de (6), ahora bien en particular satisfacer 2) dará como resultado una aproximación más precisa de la función real, dado que estamos utilizando más información de la misma, pero para ello necesitaremos conocer los valores de la derivada de la función en esos puntos y esto en general no suele suceder. Para este trabajo práctico utilizaremos 1) debido a que no conocemos los valores de la derivada, y más aún, no disponemos de la función que aproximaremos.

Dadas estas condiciones podremos plantear un sistema lineal de cuya solución se podrá deducir la función interpolante. Para empezar sabemos por (1) que para cada intervalo $[x_j, x_{j+1}]$ tendremos un polinomio cúbico $S_j(x)$, por lo que podremos asociarles la siguiente forma general.

$$S_j(x) = a_j + b_j(x - x_j) + c_j(x - x_j)^2 + d_j(x - x_j)^3 \tag{6}$$

Para cada $j = 0, 1, \ldots, n-1$, con x_j el elemento $j - esimo$ de la partición del dominio, y con a_j, b_j, c_j y d_j constantes a determinar (particulares para cada $S_j(x)$).

Observemos que al evaluar en x_j en $S(x)$ se tiene que $S(x_j) = S_j(x_j) = a_j$, y por (2) $S(x)$ debe satisfacer que $S(x_j)$ sea igual a $f(x_j)$, por lo que evidentemente se concluye que $a_j = f(x_j)$, el cual conocemos de antemano.

A su vez por (3) debemos pedirle a $S_j(x)$ que satisfaga que $S_j(x_{j+1}) = f(x_{j+1})$, es decir que en cada extremo en común $S_j(x_{j+1})$ debe ser igual a $S_{j+1}(x_{j+1})$, lo que no lleva a que:

$$S_{j+1}(x_{j+1}) = a_{j+1} = a_j + b_j(x_{j+1} - x_j) + c_j(x_{j+1} - x_j)^2 + d_j(x_{j+1} - x_j)^3 = S_j(x_{j+1}) \tag{7}$$

Para alivianar la notación notemos como a $(x_{j+1} - x_j)$ como $h_j = (x_{j+1} - x_j)$, reescribiendo la ecuación anterior como:

$$S_{j+1}(x_{j+1}) = a_{j+1} = a_j + b_j h_j + c_j h_j^2 + d_j h_j^3 = S_j(x_{j+1}) \tag{8}$$

De manera análoga se tiene por (4) que $S^{'}_{j+1}(x_{j+1}) = S^{'}_j(x_{j+1})$, en donde $S^{'}_j(x) = b_j + 2c_j(x - x_j) + 3d_j(x - x_j)^2$, por lo que $S^{'}_{j+1}(x_{j+1}) = b_{j+1}$, pero entonces esto significa que:

$$S_{j+1}(x_{j+1}) = b_{j+1} = b_j + 2c_j h_j + 3d_j h_j^2 = S_j(x_{j+1}) \tag{9}$$

Repitiendo el procedimiento en base a (5) busquemos despejar c_j. Dado que $S_j^{''}(x) = 2c_j + 6d_j(x - x_j)$, debemos pedir que $S_{j+1}''(x_{j+1}) = S_j''(x_{j+1})$, evaluando y dividiendo por 2 a ambos lados se tiene que:

$$S_{j+1}^{''}(x_{j+1})/2 = c_{j+1} = c_j + 3d_jh_j = S_j(x_{j+1})/2 \tag{10}$$

Con lo cual podemos depejar d_j en función de c_j y c_{j+1}, definiendolo como $d_j = (c_{j+1} - c_j)/3h_j$. Luego reemplazandolo en (15) obtendremos que:

$$a_{j+1} = a_j + b_jh_j + c_jh_j^2 + \frac{h_j^2}{3}(c_{j+1} - c_j) \tag{11}$$

$$a_{j+1} = a_j + b_jh_j + \frac{h_j^2}{3}(c_{j+1} + 2c_j) \tag{12}$$

$$b_j = \frac{1}{h_j}(a_{j+1} - a_j) + \frac{h_j^2}{3}(c_{j+1} + 2c_j) \tag{13}$$

Luego con esto y en base a (16), podemos concluir para $j = 1, \dots, n-1$ la siguiente equivalencia:

$$h_{j-1}c_{j-1} + 2(h_{j-1} + h_j)c_j + h_jc_{j+1} = \frac{3}{h_j}(a_{j+1} - a_j) - \frac{3}{h_{j-1}}(a_j - a_{j-1}) \tag{14}$$

Esto nos sugiere un sistema lineal $n-1$ ecuaciones al que agregaremos otras dos que resultan de aplicar la condición de (6), en donde $S^{''}(x_0) = S^{''}(x_n) = 0$, con lo cual tenemos que $c_n = S^{''}(x_n) = 0$ y que:

$$S^{''}(x_0) = c_0 + 3d_0(x_0 - x_0) = c_0 = 0 \tag{15}$$

Luego nuestro sistema lineal queda planteado por $n+1$ ecuaciones de $n+1$ incognitas, por lo que podremos plantearlo como una matriz cuadrada en donde sus columnas representarán los coeficientes de $c_0, \dots, c_n$ para cada ecuación. Mas explícitamente el sistema lineal queda definido por $Ax = b$, donde A, x y b serán de la siguiente forma.

$$A = \begin{pmatrix} 1 & 0 & 0 & \dots & \dots & 0 \\ h_0 & 2(h_0+h_1) & h_1 & \ddots & & \vdots \\ 0 & h_1 & 2(h_1+h_2) & h_2 & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & h_{n-2} & 2(h_{n-2}+h_{n-1}) & h_{n-1} \\ 0 & \dots & \dots & 0 & 0 & 1 \end{pmatrix} \quad x = \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ \vdots \\ \vdots \\ c_n \end{pmatrix}$$

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

Una cárcteristica importante a notar de la matriz A es que se trata de una matriz estrictamente diagonal dominante, que por lo visto en la primera mitad del curso, tiene solución y es única. Esta cárcteristica nos abre además una nueva posibilidad sobre este método, ya que si una matriz es estrictamente diagonal dominante, entonces ésta posee factorización LU. Dado esto, notemos que para generar la matriz A hacemos uso únicamente de los h_j, por ende A solo va a depender de la partición inicial de nuestro dominio de interes; luego tras calcular la factorización LU podremos reducir notablemente la complejidad en el caso en que queramos aplicar este alogritmo para interpolar una segunda función, siempre y cuando se considere una partición de la cual se deduzcan los mismos h_j.

Veamos a modo de ejemplo como se comporta la función $sen(2\pi(x/5))$ con spline natural.

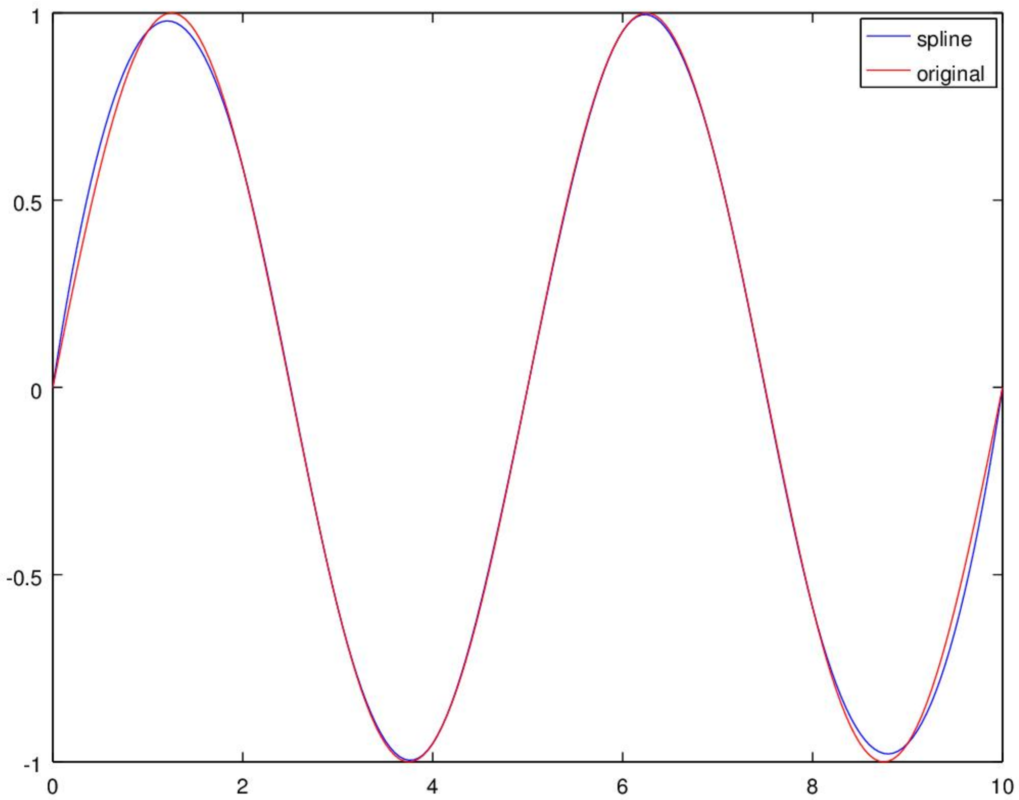

Figura 5: Splines cúbicos: Paso: 1.0

Parece bastante claro que para este método la función interpolante se ajusta mucho mejor a la función original en comparación a la propuesta por la interpolación lineal. Esto se debe a las condiciones pedidas sobre los extremos de los intervalos, que dan como resultado una función con curvas mas suaves, que nos evitan problemas como los que ocurren por los picos obtenidos en la interpolación lineal.

2.4. Nearest Neighbour

Este método aunque podría considerarse poco efectivo, nos servirá para tener un margen más amplio de comparación. El mismo se basa en la idea de que tambien podemos interpolar copiando el valor más cercano al punto que estamos evaluando. Por lo tanto, dados x_i y x_{i+1} y sus respectivas imágenes $f(x_i)$ y $f(x_{i+1})$ y un punto x^* que deseamos interpolar, se define $P_{i,i+1}(x)$ como

$$P(x) = \begin{cases} f(x_i) & : |x_i - x^*| < |x_{i+1} - x^*| \\ \\ f(x_{i+1}) & : |x_{i+1} - x^*| \leq |x_i - x^*| \end{cases}$$

Luego por cada par x_i y x_{i+1} obtendriamos un polinomio de estas caracteristicas quedando el polinomio interpolador definido de la siguiente manera:

$$P(x) = \begin{cases} P_{0,1}(x) & : x_0 \leq x < x_1 \\ \\ P_{1,2}(x) & : x_1 \leq x < x_2 \\ \vdots & \\ P_{n-1,n}(x) & : x_{n-1} \leq x_i \leq x_n \end{cases}$$

Veamos que sucede con la función $sen(2\pi(x/5))$ al interpolar con este método

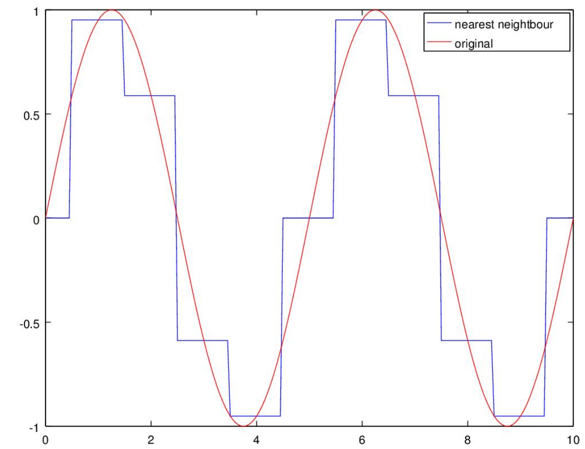

Figura 6: Nearest neighbour: Paso: 1.0

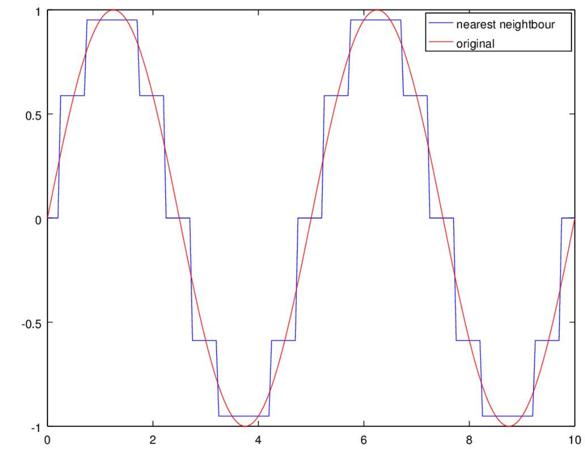

Figura 7: Nearest neighbour: Paso: 0.5

Como se puede apreciar, al igual que en el método lineal, al tomar un paso más pequeño el polinomio se vuelve más preciso. Cabe destacar que igualmente este método desaprovecha en gran medida la información disponible sobre la función. Una mejora posible es aplicar un promedio con los extremos del intervalo.

En general, Spline cúbico será lo ideal para trabajar con funciones continuas debido a que no siempre contaremos con la cantidad suficiente de información sobre la función que estamos interpolando como para lograr que Lineal o Nearest neighbour aproximen correctamente.

3. Slow Motion: Modelado

3.1. El objetivo inicial

Como mencionamos en la introducción el objetivo de este trabajo práctico es generar videos en cámara lenta basandonos en la idea de introducir cuadros intermedios generados con los métodos mencionados de interpolación. Veremos ahora como representar un video para poder lograrlo y luego aplicaremos cada método y observaremos que sucede en cada caso.

3.2. Manipulación de los cuadros: interpolando imágenes

Un video está compuesto por cierta cantidad de cuadros (imágenes de las que se compone el video) que será determinada por la cantidad de cuadros por segundo o fps a la que el video se reproduce, y la duración del mismo. Por ejemplo si el video se reproduce a 30 fps y dura 5 segundos, tendremos 150 cuadros en total y dado que 1 sg = 1000 ms, cada imágen se verá en pantalla por aproximadamente 33,33 ms.

Cada imágen está compuesta por pixeles que se pueden representar en matrices donde cada posicion i, j de un cuadro k representa un color en la escala de grises $[0.,255] \in \mathbb{N}_0$. Los videos que utilizaremos en la experimentación serán por simplicidad en escala de grises, la idea es que esto disminuirá los tiempos de cómputo involucrados en comparación al costo que generaría trabajar con toda la representación de colores rgb. Aunque esta misma idea puede generalizarse y adaptarse para todos los colores sin inconveniente alguno.

La matriz que describe el cuadro número k, de n filas y m columnas puede expresarse de la siguiente manera

$$\begin{pmatrix} f(1,1,k) & f(1,2,k) & \dots & f(1,n,k) \\ f(2,1,k) & f(2,2,k) & \dots & f(2,n,k) \\ \vdots & \vdots & \ddots & \vdots \\ f(m,1,k) & f(m,2,k) & \dots & f(m,n,k) \end{pmatrix}$$

Donde $f(i,j,k) \in [0.,255]$

Lo que haremos una vez obtenidos los cuadros que representan el video será interpolar entre cada par de posiciones pertenecientes a la misma fila y columna. Es decir para cada $f(i,j,k)$, $f(i,j,k+1)$ $\forall\ k = 0,1,\dots,r-1$ siendo r la cantidad de cuadros del video, generaremos un polinomio interpolador para poder aproximar los valores intermedios a los cuadros k y $k+1$. Dicho polinomio puede expresarse de la siguiente manera, idependientemente del método aplicado.

$$P^{i,j}(x) = \begin{cases} P^{i,j}_{0,1}(x) & : x_0 \leq x < x_1 \\ P^{i,j}_{1,2}(x) & : x_1 \leq x < x_2 \\ \vdots \\ P^{i,j}_{n-1,m}(x) & : x_{r-1} \leq x \leq x_r \end{cases}$$

Donde $P^{i,j}_{k,k+1}$ es el polinomio que interpola las posiciones i, j para los cuadros k y $k+1$, siendo x_k el pixel del cuadro k en esa posición.

3.3. Discretización

El tamaño del paso del polinomio estará determinado por la cantidad de cuadros que queramos generar entre cada par de cuadros, es decir que podriamos generar más de un cuadro intermedio, ergo, el paso será más pequeño. Como condición general el paso será el mismo para todos los polinomios.

Dependiendo del método elegido, tomar más cuadros puede resultar beneficioso para la fluidez del resultado, pero esto en principio no será siempre así. Veremos que para los polinomios lineales esto no producirá ningún tipo de mejora y en cambio serán mejores los resultados obtenidos en interpolación de spline cúbico. Además en general, cuanto más pequeño sea el paso involucrado, más costoso será obtener el resultado de la interpolación. Existirá como no podia ser de otra manera un trade-off entre complejidad, eficiencia, suavidad y nitidez del resultado final.

3.4. El modelo empírico

Para poder valorar el resultado de las experimentaciones correctamente no basta con un análisis subjetivo de comparación de los métodos entre si. Necesitamos poder comparar con un módelo empírico que nos de una idea más objetiva de como se comporta cada interpolación. Para esto mismo consideraremos un video del que obtendremos sus cuadros tomando saltos equidistantes, por ejemplo, eliminado los cuadros impares, sin eliminar el primero y el último ya que los bordes siempre estarán incluidos. Luego interpolaremos los cuadros faltantes con cada uno de los métodos y tomaremos las siguientes medidas:

$$PSNR = 10 \times log_{10}(\frac{MAX_u^2}{ECM}) \tag{16}$$

Donde MAX_u define el rango máximo de la imágen, es decir, 255 y ECM el error cuadrático medio que se define como

$$\frac{1}{N}\sum_{i,j}(u^*_{ij} - u_{ij})^2 \tag{17}$$

Donde N es la cantidad de pixeles de la imágen, u^* es la imágen ideal y u es la imágen que construimos. Este error es bastante intuitivo y tiene mucha aplicación en estadistica. Nos da una idea del error cometido al aproximar todos los valores de la imágen ideal.

El $PSNR$ es algo menos intuitivo. Sus siglas provienen de Peak Signal-to-Noise Ratio que en español significa Relación Señal a Ruido de Pico. Suele usarse en el ámbito de la compresión de imágenes y es la medida cuantitativa de la calidad de la reconstrucción. Lo que define es la relación entre la máxima energía posible de una señal y el ruido que afecta a su representación fidedigna.

Tendrá aplicación en esta experimentación dado que lo que haremos tambien será reconstruir parte del video.

De esta manera podremos saber que método aproxima mejor al resultado ideal de manera más objetiva.

4. Experimentación

5. Conclusiones

6. Apéndice A: Enunciado

Métodos Numéricos
Segundo Cuatrimestre 2015
Trabajo Práctico 3

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Un juego de niños

Introducción

¿Quién nunca ha visto un video gracioso de bebés? El éxito de esas producciones audiovisuales ha sido tal que el sitio youborn.com es uno de los más visitados diariamente. Los dueños de este gran sitio, encargado de la importantísima tarea de llevar videos graciosos con bebés a todo el mundo, nos ha pedido que mejoremos su sistema de reproducción de videos.

Su objetivo es tener videos en cámara lenta (ya que todos deseamos tener lujo de detalle en las expresiones de los chiquilines en esos videos) pero teniendo en cuenta que las conexiones a internet no necesariamente son capaces de transportar la gran cantidad de datos que implica un video en *slow motion*. La gran idea es minimizar la dependencia de la velocidad de conexión y sólo enviar el video original. Una vez que el usuario recibe esos datos, todo el trabajo de la cámara lenta puede hacerse de modo offline del lado del cliente, optimizando los tiempos de transferencia. Para tal fin utilizaremos técnicas de interpolación, buscando generar, entre cada par de cuadros del video original, otros ficticios que nos ayuden a generar un efecto de slow motion.

Definición del problema y metodología

Para resolver el problema planteado en la sección anterior, se considera el siguiente contexto. Un video está compuesto por cuadros (denominados también *frames* en inglés) donde cada uno de ellos es una imagen. Al reproducirse rápidamente una después de la otra percibimos el efecto de movimiento a partir de tener un “buen frame rate”, es decir una alta cantidad de cuadros por segundo o fps (frames per second). Por lo general las tomas de cámara lenta se generan con cámaras que permiten tomar altísimos números de cuadros por segundo, unos 100 o más en comparación con entre 24 y 30 que se utilizan normalmente.

En el caso del trabajo práctico crearemos una cámara lenta sobre un video grabado normalmente. Para ello colocaremos más cuadros entre cada par de cuadros consecutivos del video original de forma que representen la información que debería haber en la transición y reproduciremos el resultado a la misma velocidad que el original. Las imágenes correspondientes a cada cuadro están conformadas por píxeles. En particular, en este trabajo utilizaremos imágenes en escala de grises para disminuir los costos en tiempo necesarios para procesar los datos y simplificar la implementación; sin embargo, la misma idea puede ser utilizada para videos en color.

El objetivo del trabajo es generar, para cada posición (i, j), los valores de los cuadros agregados en función de los cuadros conocidos. Lo que haremos será interpolar en el tiempo y para ello, se propone considerar al menos los siguientes tres métodos de interpolación:

1. *Vecino más cercano:* Consiste en rellenar el nuevo cuadro replicando los valores de los píxeles del cuadro original que se encuentra más cerca.

2. *Interpolación lineal:* Consiste en rellenar los píxeles utilizando interpolaciones lineales entre píxeles de cuadros originales consecutivos.

3. *Interpolación por Splines:* Simliar al anterior, pero considerando interpolar utilizando splines y tomando una cantidad de cuadros mayor. Una alternativa a considerar es tomar la información de bloques de un tamaño fijo (por ejemplo, 4 cuadros, 8 cuadros, etc.), con el tamaño de bloque a ser determinado experimentalmente.

Cada método tiene sus propias características, ventajas y desventajas particulares. Para realizar un análisis cuantitativo, llamamos F al frame del video real (ideal) que deberíamos obtener con nuestro algoritmo, y sea $\bar{F}$ al frame del video efectivamente construido. Consideramos entonces dos medidas, directamente relacionadas entre ellas, como el *Error Cuadrático Medio* (ECM) y *Peak to Signal Noise Ratio* (PSNR), denotados por $\texttt{ECM}(F, \bar{F})$ y $\texttt{PSNR}(F, \bar{F})$, respectivamente, y definidos como:

$$\texttt{ECM}(F, \bar{F}) = \frac{1}{mn} \sum_{i=1}^{m} \sum_{j=1}^{n} |F_{k_{ij}} - \bar{F}_{k_{ij}}|^2 \tag{1}$$

y

$$\texttt{PSNR}(F, \bar{F}) = 10 \log_{10} \left(\frac{255^2}{\texttt{ECM}(F, \bar{F})} \right). \tag{2}$$

Donde m es la cantidad de filas de píxeles en cada imagen y n es la cantidad de columnas. Esta métrica puede extenderse para todo el video.

En conjunto con los valores obtenidos para estas métricas, es importante además realizar un análisis del tiempo de ejecución de cada método y los denominados *artifacts* que produce cada uno de ellos. Se denominan *artifacts* a aquellos errores visuales resultantes de la aplicación de un método o técnica. La búsqueda de este tipo de errores complementa el estudio cuantitativo mencionado anteriormente incorporando un análisis cualitativo (y eventualmente subjetivo) sobre las imágenes generadas.

Enunciado

Se pide implementar un programa en `C` o `C++` que implemente como mínimo los tres métodos mencionados anteriormente y que dado un video y una cantidad de cuadros a agregar aplique estas técnicas para generar un video de cámara lenta. A su vez, es necesario explicar en detalle cómo se utilizan y aplican los métodos descriptos en 1, 2 y 3 (y todos aquellos otros métodos que decidan considerar opcionalmente) en el contexto propuesto. Los grupos deben a su vez plantear, describir y realizar de forma adecuada los experimentos que consideren pertinentes para la evaluación de los métodos, justificando debidamente las decisiones tomadas y analizando en detalle los resultados obtenidos así como también plantear qué pruebas realizaron para convencerse de que los métodos funcionan correctamente.

Programa y formato de entrada

Se deberán entregar los archivos fuentes que contengan la resolución del trabajo práctico. El ejecutable tomará cuatro parámetros por línea de comando que serán el archivo de entrada, el archivo de salida, el método a ejecutar (0 para vecinos más cercanos, 1 para lineal, 2 para splines y otros números si consideran más métodos) y la cantidad de cuadros a agregar entre cada par del video original.

Tanto el archivo de entrada como el de salida tendrán la siguiente estructura:

- En la primera línea está la cantidad de cuadros que tiene el video (`c`).
- En la segunda línea está el tamaño del cuadro donde el primer número es la cantidad de filas y el segundo es la cantidad de columnas (`height width`).
- En la tercera línea está el framerate del video (`f`).
- A partir de allí siguen las imágenes del video una después de la otra en forma de matriz. Las primeras `height` líneas son las filas de la primera imagen donde cada una tiene `width` números correspondientes a los valores de cada píxel en esa fila. Luego siguen las filas de la siguiente imagen y así sucesivamente.

Además se presentan herramientas en Matlab para transformar videos (la herramienta fue probada con la extensión .avi pero es posible que funcione para otras) en archivos de entrada para el enunciado y archivos de salida en videos para poder observar el resultado visualmente. También se recomienda leer el archivo de README sobre la utilización.

Sobre la entrega

- Formato electrónico: Martes 10 de Noviembre de 2015, **hasta las 23:59**, enviando el trabajo (`informe` + `código`) a `metnum.lab@gmail.com`. El `asunto` del email debe comenzar con el texto `[TP3]` seguido de la lista de apellidos de los integrantes del grupo. Ejemplo: `[TP3] Artuso, Belloli, Landini`
- Formato físico: Miércoles 11 de Noviembre de 2015, en la clase práctica.

7. Apéndice B: Código

7.1. matrix.h

```
/*
 * File:   matrix.h
 * Author: Federico
 *
 * Created on August 16, 2015, 9:54 PM
 */

#ifndef MATRIX_H
#define MATRIX_H

#include <algorithm>
#include <math.h>
#include <vector>
#include <stdio.h>

using namespace std;

// La matriz respeta la notacion de la catedra, es decir, el primer subindice
// es la fila y el segundo es la columna

template<class T>
class Matrix {
    public:
        Matrix();
        Matrix(int rows); // Columnas impllicitas (col = 1)
        Matrix(int rows, int col);
        Matrix(int rows, int col, const T& init);
        Matrix(const Matrix<T>& other);
        ~Matrix();

        Matrix<T>& operator=(const Matrix<T>& other);
        Matrix<T> operator*(const Matrix<T>& other);
        Matrix<T>& operator*=(const Matrix<T>& other);
        Matrix<T> operator+(const Matrix<T>& other);
        Matrix<T>& operator+=(const Matrix<T>& other);
        Matrix<T> operator-(const Matrix<T>& other);
        Matrix<T>& operator-=(const Matrix<T>& other);

        Matrix<T> operator*(const T& scalar);
        Matrix<T> operator/(const T& scalar);

        T& operator()(int a, int b);
        const T& operator()(const int a, const int b) const;
        T& operator()(int a);
        const T& operator()(const int a) const;

        int rows();
        int columns();
        void printMatrix();

    private:
        vector<vector<T> > _values;
        int _rows;
        int _columns;

```

```
};

template<class T>
Matrix<T>::Matrix()
    : _values(1), _rows(1), _columns(1)
{
    _values[0].resize(1);
}

template<class T>
Matrix<T>::Matrix(int rows)
    : _values(rows), _rows(rows), _columns(1)
{
    for(int i = 0; i < rows; i++) {
        _values[i].resize(1);
    }
}

template<class T>
Matrix<T>::Matrix(int rows, int col)
    : _values(rows), _rows(rows), _columns(col)
{
    for(int i = 0; i < rows; i++) {
        _values[i].resize(col);
    }
}

template<class T>
Matrix<T>::Matrix(int rows, int col, const T& init)
    : _values(rows), _rows(rows), _columns(col)
{
    for(int i = 0; i < rows; i++) {
        _values[i].resize(col, init);
    }
}

template<class T>
Matrix<T>::Matrix(const Matrix<T>& other)
    : _values(other._values), _rows(other._rows), _columns(other._columns)
{}

template<class T>
Matrix<T>::~Matrix() {}

template<class T>
Matrix<T>& Matrix<T>::operator=(const Matrix<T>& other) {
  if (&other == this)
    return *this;

  int new_rows = other._rows;
  int new_columns = other._columns;

  _rows = new_rows;
  _columns = new_columns;

  _values.resize(new_rows);
  for (int i = 0; i < new_columns; i++) {
      _values[i].resize(new_columns);
  }
```

```

  for(int i = 0; i < new_rows; i++) {
    for(int j = 0; j < new_columns; j++) {
      _values[i][j] = other(i, j);
    }
  }

  return *this;
}

template<class T>
Matrix<T> Matrix<T>::operator*(const Matrix<T>& other) {
    // ASUME QUE LAS DIMENSIONES DAN
    Matrix<T> result(_rows, other._columns);

    int innerDim = _columns; // Tambien podria ser other._rows

    for(int i = 0; i < result._rows; i++) {
        for(int j = 0; j < result._columns; j++) {
            result(i,j) = 0;
            for(int k = 0; k < innerDim; k++) {
                result(i,j) += _values[i][k] * other(k,j);
            }
        }
    }

    return result;
}

template<class T>
Matrix<T>& Matrix<T>::operator*=(const Matrix<T>& other) {
    Matrix<T> result = (*this) * other;
    (*this) = result;
    return (*this);
}

template<class T>
Matrix<T> Matrix<T>::operator+(const Matrix<T>& other) {
    // ASUME QUE LAS DIMENSIONES DAN
    Matrix<T> result(_rows, other._columns);

    for(int i = 0; i < result._rows; i++) {
        for(int j = 0; j < result._columns; j++) {
            result(i,j) = _values[i][j] + other(i,j);
        }
    }

    return result;
}

template<class T>
Matrix<T>& Matrix<T>::operator+=(const Matrix<T>& other) {
    Matrix<T> result = (*this) + other;
    (*this) = result;
    return (*this);
}

template<class T>
Matrix<T> Matrix<T>::operator-(const Matrix<T>& other) {
```

```
    // ASUME QUE LAS DIMENSIONES DAN
    Matrix<T> result(_rows, other._columns);

    for(int i = 0; i < result._rows; i++) {
        for(int j = 0; j < result._columns; j++) {
            result(i,j) = _values[i][j] - other(i,j);
        }
    }

    return result;
}

template<class T>
Matrix<T>& Matrix<T>::operator-=(const Matrix<T>& other) {
    Matrix<T> result = (*this) - other;
    (*this) = result;
    return (*this);
}

template<class T>
Matrix<T> Matrix<T>::operator*(const T& scalar) {
    Matrix<T> result(_rows, _columns);

    for(int i = 0; i < result._rows; i++) {
        for(int j = 0; j < result._columns; j++) {
            result(i,j) = _values[i][j] * scalar;
        }
    }

    return result;
}

template<class T>
Matrix<T> Matrix<T>::operator/(const T& scalar) {
    Matrix<T> result(_rows, _columns);

    for(int i = 0; i < result._rows; i++) {
        for(int j = 0; j < result._columns; j++) {
            result(i,j) = _values[i][j] / scalar;
        }
    }

    return result;
}

template<class T>
T& Matrix<T>::operator ()(int a, int b) {
    return _values[a][b];
}

template<class T>
const T& Matrix<T>::operator ()(const int a, const int b) const {
    return _values[a][b];
}

template<class T>
T& Matrix<T>::operator ()(int a) {
    return _values[a][0];
}
```

```

template<class T>
const T& Matrix<T>::operator ()(const int a) const {
    return _values[a][0];
}

template<class T>
int Matrix<T>::rows() {
    return _rows;
}

template<class T>
int Matrix<T>::columns() {
    return _columns;
}

template<class T>
void Matrix<T>::printMatrix() {
    for(int i = 0; i < _rows; i++) {;
        for(int j = 0; j < _columns; j++) {
            printf("%3.3f ", _values[i][j]);
            // cout << _values[i][j] << " ";
        }
        cout << endl;
    }
    cout << endl;
}

#endif  /* MATRIX_H */
```

7.2. eqsys.h

```
/*
 * File:   eqsys.h
 * Author: Federico
 *
 * Created on August 17, 2015, 5:57 PM
 */

#ifndef EQSYS_H
#define EQSYS_H

#include <algorithm>
#include <math.h>
#include <vector>
#include "matrix.h"

template<class T>
class EquationSystemLU {
    public:
        EquationSystemLU(const Matrix<T>& inicial);

        Matrix<T> solve(Matrix<T>& b_values);

    private:
        Matrix<T> lower;
        Matrix<T> upper;
        bool isPermutated;
        Matrix<T> permutation;
```

```cpp
};

template<class T>
EquationSystemLU<T>::EquationSystemLU(const Matrix<T>& inicial)
    : upper(inicial), isPermutated(false)
{
    T coef;
    int i, j, k, l;

    // Armar la matriz lower
    lower = Matrix<T>(upper.rows(), upper.columns(), 0);

    for(i = 0; i < upper.columns(); i++) {
        for(j = i + 1; j < upper.rows(); j++) {
            if(upper(i, i) == 0) {
                // Hay que buscar la proxima fila sin cero
                for(k = i + 1; k < upper.rows(); k++) {
                    if(upper(k, i) != 0) {
                        break;
                    }
                }

                if(k == upper.rows()) { // No hay filas para permutar
                    abort();
                } else {
                    if(!isPermutated){
                        // Generamos la matriz de permutacion con uno en la diagonal
                        isPermutated = true;
                        permutation = Matrix<T>(upper.rows(), upper.columns(), 0);

                        for(l = 0; l < permutation.rows(); l++) {
                            permutation(l,l) = 1;
                        }
                    }
                    // Permutamos las filas
                    for(l = 0; l < permutation.columns(); l++) {
                        if(l == k) {
                            permutation(i, l) = 1;
                        } else {
                            permutation(i, l) = 0;
                        }
                        if(l == i) {
                            permutation(k, l) = 1;
                        } else {
                            permutation(k, l) = 0;
                        }
                    }
                    // Hacemos el producto para efectivamente permutar
                    upper = permutation * upper;
                    lower = permutation * lower;
                }
            }

            // Calculamos y guardamos el coeficiente
            // cout << upper(j,i) << " , " << upper(i,i) << endl;
            coef = upper(j, i) / upper(i, i);
            lower(j, i) = coef;
            // Colocamos cero en la columna bajo la diagonal
            upper(j,i) = 0;
```

```
            for(k = i + 1; k < upper.columns(); k++) {
                upper(j, k) = upper(j, k) - coef * upper(i, k);
            }
        }
    }
    // Agrego la diagonal de unos a lower
    for(i = 0; i < lower.rows(); i++){
        lower(i,i) = 1;
    }
}

template<class T>
Matrix<T> EquationSystemLU<T>::solve(Matrix<T>& b_values) {

    Matrix<T> temp_values = Matrix<T>(b_values);
    Matrix<T> y_values = Matrix<T>(b_values.rows());
    Matrix<T> x_values = Matrix<T>(b_values.rows());

    if(isPermutated) {
        temp_values = permutation * temp_values;
    }

    // Resuelvo el sistema L * y = b
    for(int i = 0; i < temp_values.rows(); i++) {
        for (int j = 0; j < i; j++) {
            temp_values(i) -= y_values(j) * lower(i,j);
        }
        if(i == 0) {
            y_values(0) = temp_values(0) / lower(0,0); // Calculo aparte el primer valor
        } else {
            y_values(i) = temp_values(i) / lower(i,i);
        }
    }

    // Resuelvo el sistema U * x = y
    temp_values = y_values;
    for(int i = temp_values.rows() - 1; i >= 0; i--) {
        for (int j = temp_values.rows() - 1; j > i; j--) {
            temp_values(i) -= x_values(j) * upper(i,j);
        }
        if(i == x_values.rows() - 1) {
            x_values(x_values.rows() - 1) = temp_values(temp_values.rows() - 1) / upper(upper.rows() - 1, upper.columns() - 1);
        } else {
            x_values(i) = temp_values(i) / upper(i,i);
        }
    }

    // Retorno la solucion al sistema LU * x = b
    return x_values;

}



template<class T>
class EquationSystem{
    public:
```

```cpp
        EquationSystem(const Matrix<T>& inicial);

        Matrix<T> solve(const Matrix<T>& b_values);

    private:
        Matrix<T> _matrix;
};

template<class T>
EquationSystem<T>::EquationSystem(const Matrix<T>& inicial)
    : _matrix(inicial)
{}

template<class T>
Matrix<T> EquationSystem<T>::solve(const Matrix<T>& b_values) {
    T coef;
    int i, j, k, l;
    bool isPermutated;
    Matrix<T> temp_matrix(_matrix);
    Matrix<T> temp_values(b_values);
    Matrix<T> permutation;

    for(i = 0; i < temp_matrix.columns(); i++) {
        for(j = i + 1; j < temp_matrix.rows(); j++) {
            if(temp_matrix(i, i) == 0) {
                // Hay que buscar la proxima fila sin cero
                for(k = i + 1; k < temp_matrix.rows(); k++) {
                    if(temp_matrix(k, i) != 0) {
                        break;
                    }
                }

                if(k == temp_matrix.rows()) { // No hay filas para permutar
                    abort();
                } else {
                    if(!isPermutated){
                        // Generamos la matriz de permutacion con uno en la diagonal
                        isPermutated = true;
                        permutation = Matrix<T>(temp_matrix.rows(), temp_matrix.
                            columns(), 0);

                        for(l = 0; l < permutation.rows(); l++) {
                            permutation(l,l) = 1;
                        }
                    }
                    // Permutamos las filas
                    for(l = 0; l < permutation.columns(); l++) {
                        if(l == k) {
                            permutation(i, l) = 1;
                        } else {
                            permutation(i, l) = 0;
                        }
                        if(l == i) {
                            permutation(k, l) = 1;
                        } else {
                            permutation(k, l) = 0;
                        }
                    }
                    // Hacemos el producto para efectivamente permutar
```

```
                    temp_matrix = permutation * temp_matrix;
                    temp_values = permutation * temp_values;
                }
            }

            // Calculamos y guardamos el coeficiente
            coef = temp_matrix(j, i) / temp_matrix(i, i);
            // Colocamos cero en la columna bajo la diagonal
            temp_matrix(j, i) = 0;
            for(k = i + 1; k < temp_matrix.columns(); k++) {
                temp_matrix(j, k) = temp_matrix(j, k) - coef * temp_matrix(i, k);
            }
            temp_values(j) = temp_values(j) - coef * temp_values(i);
        }
    }

    Matrix<T> x_values = Matrix<T>(temp_values.rows());

    // Resuelvo el sistema A * x = b, con A triangular superior
    for(int i = temp_values.rows() - 1; i >= 0; i--) {
        for (int j = temp_values.rows() - 1; j > i; j--) {
            temp_values(i) -= x_values(j) * temp_matrix(i,j);
        }
        if(i == x_values.rows() - 1) {
            x_values(x_values.rows() - 1) = temp_values(temp_values.rows() - 1) /
                temp_matrix(temp_matrix.rows() - 1, temp_matrix.columns() - 1);
        } else {
            x_values(i) = temp_values(i) / temp_matrix(i,i);
        }
    }

    // Retorno la solucion a A * x = b
    return x_values;
}

#endif  /* EQSYS_H */
```

7.3. generate.cpp

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <vector>

#include "eqsys.h"

using namespace std;

void fprintframe(FILE* outputFile, int frame, int videoWidth, int videoHeight, vector
    <vector<int> >& video);
void fprintlinearframe(FILE* outputFile, int startFrame, int currentFrame, int
    framesToGenerate,
                        int videoWidth, int videoHeight, vector<vector<int> >& video)
                            ;
void fprintframefromspline(FILE* outputFile, int frame, int currentNewFrame, int
    framesToGenerate,
                            int videoWidth, int videoHeight, vector<vector<int> >
                                video, vector<vector<int> > storage);
```

```
void naturalCubicSplineBuildA(int framesToGenerate, int videoFrames, Matrix<double>& A);
void naturalCubicSplineBuildB(int pixel, int framesToGenerate, int videoFrames, Matrix<double>& b, vector<vector<int> >& video);

int main(int argc, char* argv[]) {

    if (argc != 5) {
        printf("Use: %s input.txt output.txt interpolationMethod framesToGenerate \n"
               "Where interpolationMethod:\n"
               "0: nearest neighbour      \n"
               "1: lineal  \n"
               "2: splines \n"
               "3: other...\n"
               "e.g. %s input.txt output.txt 0 5\n", argv[0], argv[0]);
        return 0;
    }

    // load input file
    FILE* inputFile = fopen(argv[1], "r");
    if (inputFile == NULL) {
        printf("Error: %s is not a valid input file.\n", argv[1]);
        return 0;
    }

    // video metadata
    int videoFrames, videoHeight, videoWidth, videoFrameRate;
    fscanf(inputFile, "%d", &videoFrames);
    fscanf(inputFile, "%d,%d", &videoHeight, &videoWidth);
    fscanf(inputFile, "%d", &videoFrameRate);
    printf("videoFrames: %d, videoHeight: %d, videoWidth: %d, videoFrameRate: %d\n",
        videoFrames, videoHeight, videoWidth, videoFrameRate);

    // video data
    vector<vector<int> > video(videoFrames, vector<int>(videoWidth*videoHeight));

    for (int frame = 0; frame < videoFrames; ++frame) {
        for (int i = 0; i < videoHeight; ++i) {
            for (int j = 0; j < videoWidth; ++j) {
                fscanf(inputFile, "%d,", &video[frame][i*videoWidth + j]);
            }
        }
    }

    // output file
    FILE* outputFile = fopen(argv[2], "w");
    if (outputFile == NULL) {
        printf("Error: Failed to create output file.\n");
        return 0;
    }

    // select method
    int interpolationMethod = atoi(argv[3]);
    int framesToGenerate = atoi(argv[4]);
    int totalFrames = videoFrames + framesToGenerate * (videoFrames - 1);

    switch (interpolationMethod) {
        case 0:
            printf("interpolationMethod: Nearest Neighbour\n");
```

```
            // write header
            fprintf(outputFile, "%d\n%d,%d\n%d\n", totalFrames, videoHeight, videoWidth, videoFrameRate);

            // generate frames
            for (int frame = 0; frame < videoFrames - 1; ++frame) {

                // print current frame
                fprintframe(outputFile, frame, videoWidth, videoHeight, video);

                // closer to first frame
                for (int j = 0; j < framesToGenerate/2; ++j) {
                    fprintframe(outputFile, frame, videoWidth, videoHeight, video);
                }

                // closer to second frame
                for (int j = framesToGenerate/2; j < framesToGenerate; ++j) {
                    fprintframe(outputFile, frame + 1, videoWidth, videoHeight, video);
                }

            }

            // print last frame
            fprintframe(outputFile, videoFrames-1, videoWidth, videoHeight, video);

            break;
        case 1:
            printf("interpolationMethod: Lineal\n");

            // write header
            fprintf(outputFile, "%d\n%d,%d\n%d\n", totalFrames, videoHeight, videoWidth, videoFrameRate);

            for (int frame = 0; frame < videoFrames - 1; ++frame) {

                // print current frame
                fprintframe(outputFile, frame, videoWidth, videoHeight, video);

                // generate frame, initial frame: frame 0 (current frame)
                for (int j = 1; j <= framesToGenerate; ++j) {
                    fprintlinearframe(outputFile, frame, j, framesToGenerate, videoWidth, videoHeight, video);
                }

            }

            // print last frame
            fprintframe(outputFile, videoFrames-1, videoWidth, videoHeight, video);

            break;
        case 2: {
            printf("interpolationMethod: Splines\n");

            // save values of the polinomial coefficients for each pixel (c_j)
            vector<vector<int> > storage(videoWidth*videoHeight, vector<int>(videoFrames));

```

```
            Matrix<double> A(videoFrames, videoFrames, 0);
            naturalCubicSplineBuildA(framesToGenerate, videoFrames, A);
            EquationSystemLU<double> e(A);

            Matrix<double> b(videoFrames);

            // fit a spline on every pixel
            for (int i = 0; i < videoWidth*videoHeight; ++i) {
                naturalCubicSplineBuildB(i, framesToGenerate, videoFrames, b, video);

                Matrix<double> result(e.solve(b));

                // store results
                for (int j = 0; j < videoFrames; ++j) {
                    storage[i][j] = result(j);
                }

            }

            // write header
            fprintf(outputFile, "%d\n%d,%d\n%d\n", totalFrames, videoHeight, videoWidth, videoFrameRate);

            // generate frames
            for (int frame = 0; frame < videoFrames - 1; ++frame) {

                // print current frame
                fprintframe(outputFile, frame, videoWidth, videoHeight, video);

                // generate new frames
                for (int j = 1; j <= framesToGenerate; ++j) {
                    fprintframefromspline(outputFile, frame, j, framesToGenerate, videoWidth, videoHeight, video, storage);
                }

            }

            // print last frame
            fprintframe(outputFile, videoFrames-1, videoWidth, videoHeight, video);

            break;
        }
        default:
            printf("Error: Invalid interpolation method\n");
            return 0;
    }

    fclose(inputFile);
    fclose(outputFile);

    return 0;
}

// new frames are counted from 1.
void fprintframefromspline(FILE* outputFile, int frame, int currentNewFrame, int framesToGenerate,
                           int videoWidth, int videoHeight, vector<vector<int> > video, vector<vector<int> > storage) {

```

```
    int h = framesToGenerate + 1;

    for (int pixel = 0; pixel < videoWidth*videoHeight; ++pixel) {
        int c_0 = storage[pixel][frame]; // !
        int c_1 = storage[pixel][frame+1];
        // frame >= 1
        int a_0 = video[frame]  [pixel]; // !
        int a_1 = video[frame+1][pixel];
        int b_0 = (1/h)*(a_1 - a_0) - (h/3)*(2*c_0 + c_1); // !
        int d_0 = (c_1 - c_0) / (3*h); // !

        int x   = frame*h;
        int x_j = frame*h + currentNewFrame;
        int res = a_0 + b_0*(x-x_j) + c_0*pow(x-x_j,2) + d_0*pow(x-x_j,3);

        if ((pixel+1) % videoWidth == 0) {
            fprintf(outputFile, "%d\n",  res);
        } else {
            fprintf(outputFile, "%d,", res);
        }

    }
}

/* Possible improvements:
 * 1. Improve cache locality by saving one vector per pixel, right now a single pixel is in several vectors.
 * 2. LU Factorization on matrix A. It doesn't depend on the pixel being processed. Only b changes. Ax = b
 * 3. A is sparse! Better representations!
 * Reminder: A is strictly diagonally dominant, LU factorization without pivoting is valid.
 */
void naturalCubicSplineBuildA(int framesToGenerate, int videoFrames, Matrix<double>& A) {
    A(0,0) = 1;
    A(videoFrames-1,videoFrames-1) = 1;
    /* h_i = x_{j+1} - x_{j}
     * h is CONSTANT! (pixels are equidistant)
     */
    int h_0 = framesToGenerate + 1;
    int h_1 = framesToGenerate + 1;
    int h_2 = framesToGenerate + 1;
    for (int i = 1; i < videoFrames - 1; ++i) {
        // h_i = x_{j+1} - x_{j}
        A(i, i-1) = h_0;
        A(i, i)   = 2*(h_0+h_1);
        A(i, i+1) = h_1;
    }
}

void naturalCubicSplineBuildB(int pixel, int framesToGenerate, int videoFrames, Matrix<double>& b, vector<vector<int> >& video) {
    int h = framesToGenerate + 1;
    b(0) = 0;
    b(videoFrames-1) = 0;
    for (int i = 1; i < videoFrames -1; ++i) {
        // a_i = f(x_i)
        int a_0 = video[i-1][pixel];
```

```
        int a_1 = video[i  ][pixel];
        int a_2 = video[i+1][pixel];
        b(i) = (3/h) * (a_2 - a_1) - (3/h) * (a_1 - a_0);
    }
}

void fprintframe(FILE* outputFile, int frame, int videoWidth, int videoHeight, vector<vector<int> >& video) {
    for (int i = 0; i < videoHeight; ++i) {
        for (int j = 0; j < videoWidth -1; ++j) {
            fprintf(outputFile, "%d,", video[frame][i*videoWidth + j]);
        }
        fprintf(outputFile, "%d\n", video[frame][i*videoWidth + videoWidth -1]);
    }
}

void fprintlinearframe(FILE* outputFile, int startFrame, int currentFrame, int framesToGenerate,
                        int videoWidth, int videoHeight, vector<vector<int> >& video) {
    for (int i = 0; i < videoHeight; ++i) {
        for (int j = 0; j < videoWidth -1; ++j) {
            int y_0 = video[startFrame  ][i*videoWidth + j];
            int y_1 = video[startFrame+1][i*videoWidth + j];
            int m = (y_1 - y_0) / (framesToGenerate + 1 - 0);
            int b = y_0 - m*0;
            fprintf(outputFile, "%d,", m*currentFrame + b);
        }
        int y_0 = video[startFrame  ][i*videoWidth + videoWidth -1];
        int y_1 = video[startFrame+1][i*videoWidth + videoWidth -1];
        int m = (y_1 - y_0) / (framesToGenerate + 1 - 0);
        int b = y_0 - m*0;
        fprintf(outputFile, "%d,", m*currentFrame + b);
    }
}
```