

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 características 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 problemática. 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	3
1.1. Motivación y objetivos	4
2. Desarrollo	5
2.1. Interpolación	5
2.2. Interpolación polinómica	5
2.3. Cálculo del polinomio interpolador	6
2.3.1. Interpolación lineal	6
2.3.2. Spline Cúbico	7
2.4. Nearest Neighbour	9
3. Slow Motion: Modelado	9
3.1. El objetivo inicial	9
3.2. Manipilación de los cuadros: interpolando imágenes	10
3.3. Discretización	10
3.4. El modelo empírico	10
4. Experimentación	12
5. Conclusiones	13
6. Apéndice A: Enunciado	14
7. Apéndice B: Código	16
7.1. matrix.h	16
7.2. eqsys.h	20
7.3. generate.cpp	24

1. Introducción

En la practica, es sumamente comun encontrar situaciones donde se dispone de informacion discreta sobre 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 hercios (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 imagen final, logrando mayor suavidad con el método de los promedios y un acabado algo más irregular en caso contrario.

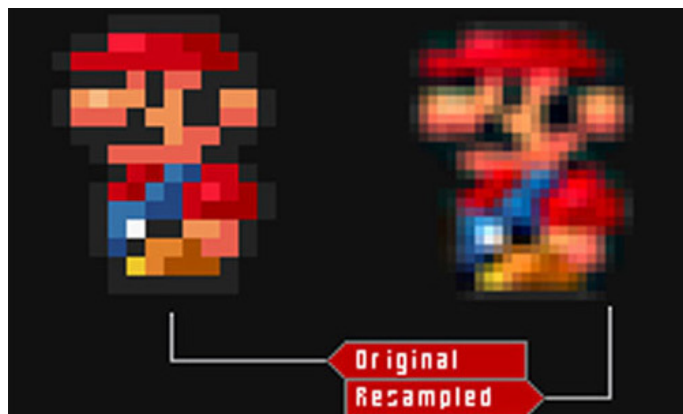


Figura 1: Aplicación de resampling a Mario Bros utilizando el promedio de los vecinos más cercanos.

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 estándar 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 distorsiones procedentes de aplicar estos algoritmos.

2. Desarrollo

2.1. Interpolación

Interpolación 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, \dots, 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 $P_m(x_k) = f(x_k)$, $\forall k = 0, 1, \dots, m$. Este polinomio se conoce como polinomio interpolador y tendrá la siguiente forma:

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

Donde $a_m, a_{m-1}, \dots, a_1, a_0 \in \mathbb{R}$.

Este tipo de interpolación suele considerarse para aproximar funciones continuas, esto es así porque los polinomios tienen buenas propiedades analíticas, entre las cuales podemos considerar las siguientes: son continuas, fácilmente integrables e infinitamente derivables (pertenecen a la clase $C^\infty[a, b]$), y además sus derivadas y primitivas son a su vez polinomios. Otra propiedad importante de la cual se fundamenta este tipo de interpolación es la de que dada cualquier función 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 Approximation Theorem

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

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

La demostración de este teorema puede encontrarse en textos de análisis real o documentos universitarios. (crear una cita a <http://www.math.harvard.edu/waffle/wapproxt.pdf>).

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 corregirá la imprecisión del polinomio con respecto a la función original para cada valor de su dominio. No obstante puede que encontrar dicha expresión 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 $\text{sen}(2\pi(x/5))$

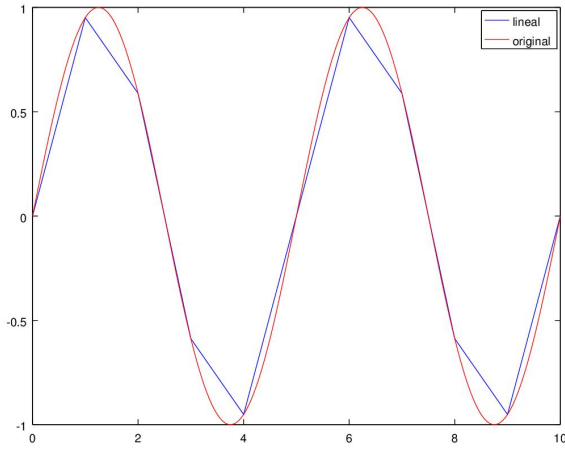


Figura 3: Lineal: Paso: 1.0

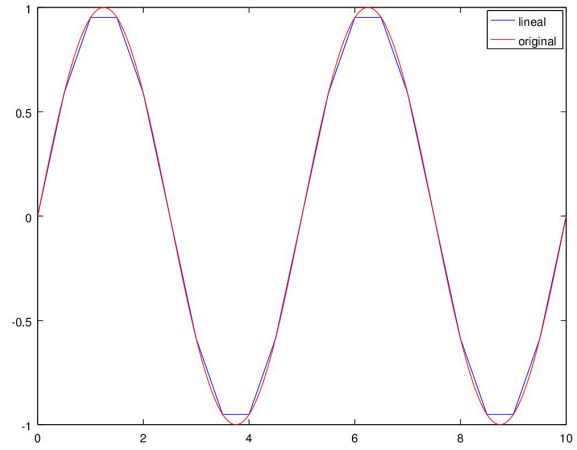


Figura 4: Lineal: Paso: 0.5

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 se será:

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

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

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

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

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

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

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

Un detalle importante 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 método al igual que el anterior se basa en particionar nuestro intervalo de interés en subintervalos más pequeños con la intención de construir un polinomio particular para cada uno de estos. Se busca entonces que para cada subintervalo, su polinomio asociado aproxime 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 $a = x_0 < x_1 < \dots < x_n = b$, 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 los extremos coincidan con los polinomios de los intervalos vecinos, y además 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 de manera formal, tenemos que, dada una función f definida en $[a, b]$ y los puntos $a = x_0 < x_1 < \dots < x_n = b$ 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 $[x_j, x_{j+1}] \forall j = 0, 1, \dots, n-1$
2. $S_j(x) = f(x_j)$ y $S_j(x_{j+1}) = f(x_{j+1}) \forall j = 0, 1, \dots, n-1$ con $S_j(x) = a_j + b_j(x - x_j) + c_j(x - x_j)^2 + d_j(x - x_j)^3$
3. $S_{j+1}(x_{j+1}) = S_j(x_{j+1}) \forall j = 0, 1, \dots, n-2$
4. $S'_{j+1}(x_{j+1}) = S'_j(x_{j+1}) \forall j = 0, 1, \dots, n-2$
5. $S''_{j+1}(x_{j+1}) = S''_j(x_{j+1}) \forall j = 0, 1, \dots, n-2$
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. Este será el caso de este trabajo práctico, por lo cual utilizaremos 1), dado que solo debemos garantizar una propiedad sobre datos conocidos.

Veamos como se comporta la función $\sin(2\pi(x/5))$ con spline natural

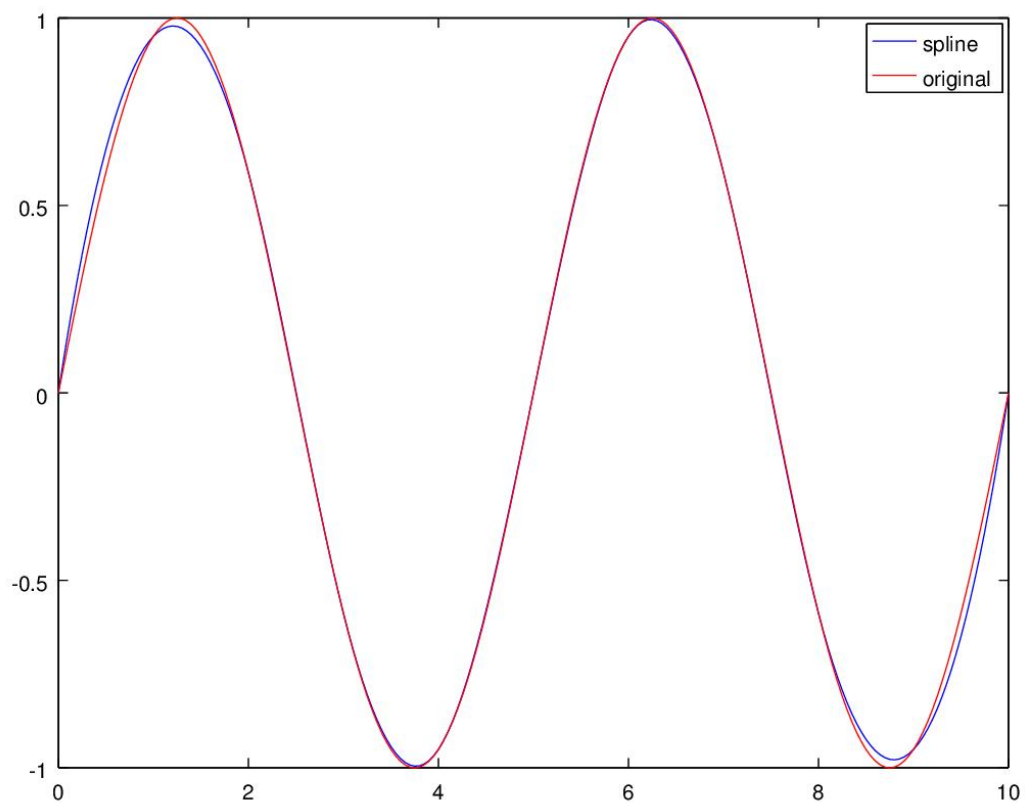


Figura 5: Splines cúbicos: Paso: 1.0

Parece bastante claro que se ajusta mucho mejor a la función original que interpolación lineal. Esto se debe a las condiciones pedidas sobre los extremos de los intervalos, logrando que las curvas sean suaves en comparación a los picos obtenidos con 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 también 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} obtendríamos un polinomio de estas características quedando el polinomio interpolador de definido de la siguiente manera

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

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

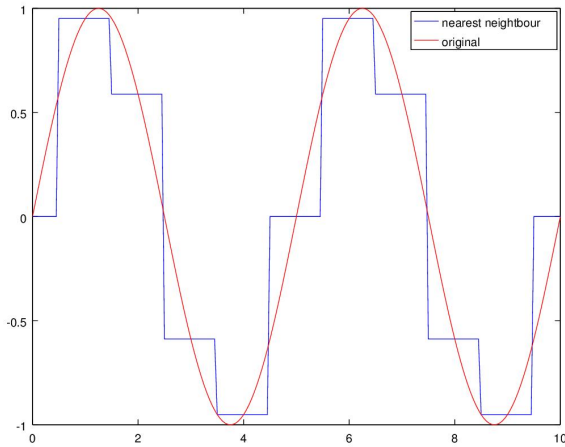


Figura 6: Nearest neighbour: Paso: 1.0

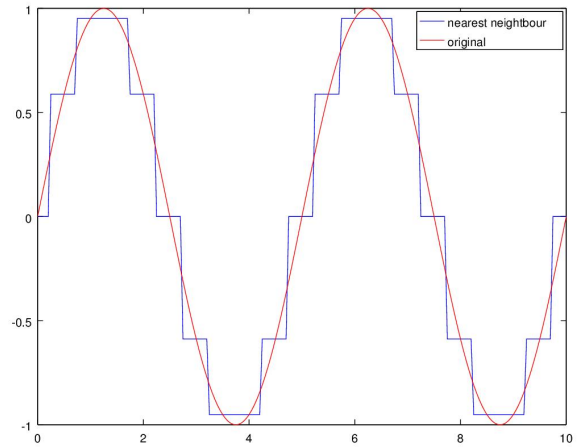


Figura 7: Nearest neighbour: Paso: 0.5

Como se puede apreciar, al igual que en el método lineal, al tomar un paso más pequeño el polinomio 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 entre los extremos del intervalo.

En general, trabajar con splines siempre será lo ideal debido a que no siempre contaremos con la cantidad suficiente de información sobre la función que estamos interpolando como para lograr que lineal aproxime 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 basándonos 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 cuadros, que no son más que las imágenes de las que se compone el video y que serán determinadas 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 imagen se verá en pantalla por aproximadamente 33,33 ms.

Cada imagen está compuesta por pixeles que se pueden representar en matrices donde cada posición i, j de un cuadro k representa un color en la escala de grises $[0., 255] \in \mathbb{N}$. Los videos que utilizaremos en la experimentación serán en escala de grises por el único motivo de que disminuirá los tiempos de cómputo involucrados en comparación a si trabajáramos 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 un cuadro puede escribirse 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 $f(i, j, k), f(i, j, k+1) \forall k = 0, 1, \dots, n-1$ 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, independientemente del método aplicado

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

Donde $P_{k,k+1}^{i,j}$ es el polinomio que interpola las posiciones i, j para los cuadros k y $k+1$.

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 podríamos 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 cuadrática y spline cúbico. Además en general, cuanto más pequeño sea el paso involucrado, más costoso será obtener el resultado de la interpolación. Existirá como no podía 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 modelo 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, eliminando 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:

$$PSNR = 10 \times \log_{10}\left(\frac{MAX_u^2}{ECM}\right) \quad (6)$$

Donde MAX_u define el rango máximo de la imagen, es decir, 255 y ECM el error cuadrático medio

$$\frac{1}{N} \sum_{i,j} (u_{ij}^* - u_{ij})^2 \quad (7)$$

Donde N es la cantidad de pixeles de la imagen, u^* es la imagen ideal y u es la imagen que construimos. Este error es bastante intuitivo y tiene mucha aplicación en estadística. Nos da una idea del error cometido al aproximar todos los valores de la imagen 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 también será reconstruir parte del video.

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

(redondear un poco más todo esto...le falta)

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*: Similiar 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 $\text{ECM}(F, \bar{F})$ y $\text{PSNR}(F, \bar{F})$, respectivamente, y definidos como:

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

y

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

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 descritos 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

```
1  /*
2  * File:    matrix.h
3  * Author:  Federico
4  *
5  * Created on August 16, 2015, 9:54 PM
6  */
7
8  #ifndef MATRIX_H
9  #define MATRIX_H
10
11 #include <algorithm>
12 #include <math.h>
13 #include <vector>
14 #include <stdio.h>
15
16 using namespace std;
17
18 // La matriz respeta la notacion de la catedra, es decir, el primer subindice
19 // es la fila y el segundo es la columna
20
21 template<class T>
22 class Matrix {
23     public:
24         Matrix();
25         Matrix(int rows); // Columnas implícitas (col = 1)
26         Matrix(int rows, int col);
27         Matrix(int rows, int col, const T& init);
28         Matrix(const Matrix<T>& other);
29         ~Matrix();
30
31         Matrix<T>& operator=(const Matrix<T>& other);
32         Matrix<T> operator*(const Matrix<T>& other);
33         Matrix<T>& operator*=(const Matrix<T>& other);
34         Matrix<T> operator+(const Matrix<T>& other);
35         Matrix<T>& operator+=(const Matrix<T>& other);
36         Matrix<T> operator-(const Matrix<T>& other);
37         Matrix<T>& operator-=(const Matrix<T>& other);
38
39         Matrix<T> operator*(const T& scalar);
40         Matrix<T> operator/(const T& scalar);
41
42         T& operator()(int a, int b);
43         const T& operator()(const int a, const int b) const;
44         T& operator()(int a);
45         const T& operator()(const int a) const;
46
47         int rows();
48         int columns();
49         void printMatrix();
50
51     private:
52         vector<vector<T> > _values;
53         int _rows;
54         int _columns;
55 }
```



```

56 };
57
58 template<class T>
59 Matrix<T>::Matrix()
60     : _values(1), _rows(1), _columns(1)
61 {
62     _values[0].resize(1);
63 }
64
65 template<class T>
66 Matrix<T>::Matrix(int rows)
67     : _values(rows), _rows(rows), _columns(1)
68 {
69     for(int i = 0; i < rows; i++) {
70         _values[i].resize(1);
71     }
72 }
73
74 template<class T>
75 Matrix<T>::Matrix(int rows, int col)
76     : _values(rows), _rows(rows), _columns(col)
77 {
78     for(int i = 0; i < rows; i++) {
79         _values[i].resize(col);
80     }
81 }
82
83 template<class T>
84 Matrix<T>::Matrix(int rows, int col, const T& init)
85     : _values(rows), _rows(rows), _columns(col)
86 {
87     for(int i = 0; i < rows; i++) {
88         _values[i].resize(col, init);
89     }
90 }
91
92 template<class T>
93 Matrix<T>::Matrix(const Matrix<T>& other)
94     : _values(other._values), _rows(other._rows), _columns(other._columns)
95 {}
96
97 template<class T>
98 Matrix<T>::~~Matrix() {}
99
100 template<class T>
101 Matrix<T>& Matrix<T>::operator=(const Matrix<T>& other) {
102     if (&other == this)
103         return *this;
104
105     int new_rows = other._rows;
106     int new_columns = other._columns;
107
108     _rows = new_rows;
109     _columns = new_columns;
110
111     _values.resize(new_rows);
112     for (int i = 0; i < new_columns; i++) {
113         _values[i].resize(new_columns);
114     }

```

```

115
116     for(int i = 0; i < new_rows; i++) {
117         for(int j = 0; j < new_columns; j++) {
118             _values[i][j] = other(i, j);
119         }
120     }
121
122     return *this;
123 }
124
125 template<class T>
126 Matrix<T> Matrix<T>::operator*(const Matrix<T>& other) {
127     // ASUME QUE LAS DIMENSIONES DAN
128     Matrix<T> result(_rows, other._columns);
129
130     int innerDim = _columns; // Tambien podria ser other._rows
131
132     for(int i = 0; i < result._rows; i++) {
133         for(int j = 0; j < result._columns; j++) {
134             result(i,j) = 0;
135             for(int k = 0; k < innerDim; k++) {
136                 result(i,j) += _values[i][k] * other(k,j);
137             }
138         }
139     }
140
141     return result;
142 }
143
144 template<class T>
145 Matrix<T>& Matrix<T>::operator*=(const Matrix<T>& other) {
146     Matrix<T> result = (*this) * other;
147     (*this) = result;
148     return (*this);
149 }
150
151 template<class T>
152 Matrix<T> Matrix<T>::operator+(const Matrix<T>& other) {
153     // ASUME QUE LAS DIMENSIONES DAN
154     Matrix<T> result(_rows, other._columns);
155
156     for(int i = 0; i < result._rows; i++) {
157         for(int j = 0; j < result._columns; j++) {
158             result(i,j) = _values[i][j] + other(i,j);
159         }
160     }
161
162     return result;
163 }
164
165 template<class T>
166 Matrix<T>& Matrix<T>::operator+=(const Matrix<T>& other) {
167     Matrix<T> result = (*this) + other;
168     (*this) = result;
169     return (*this);
170 }
171
172 template<class T>
173 Matrix<T> Matrix<T>::operator-(const Matrix<T>& other) {

```

```

174 // ASUME QUE LAS DIMENSIONES DAN
175 Matrix<T> result(_rows, other._columns);
176
177 for(int i = 0; i < result._rows; i++) {
178     for(int j = 0; j < result._columns; j++) {
179         result(i,j) = _values[i][j] - other(i,j);
180     }
181 }
182
183 return result;
184 }
185
186 template<class T>
187 Matrix<T>& Matrix<T>::operator-=(const Matrix<T>& other) {
188     Matrix<T> result = (*this) - other;
189     (*this) = result;
190     return (*this);
191 }
192
193 template<class T>
194 Matrix<T> Matrix<T>::operator*(const T& scalar) {
195     Matrix<T> result(_rows, _columns);
196
197     for(int i = 0; i < result._rows; i++) {
198         for(int j = 0; j < result._columns; j++) {
199             result(i,j) = _values[i][j] * scalar;
200         }
201     }
202
203     return result;
204 }
205
206 template<class T>
207 Matrix<T> Matrix<T>::operator/(const T& scalar) {
208     Matrix<T> result(_rows, _columns);
209
210     for(int i = 0; i < result._rows; i++) {
211         for(int j = 0; j < result._columns; j++) {
212             result(i,j) = _values[i][j] / scalar;
213         }
214     }
215
216     return result;
217 }
218
219 template<class T>
220 T& Matrix<T>::operator()(int a, int b) {
221     return _values[a][b];
222 }
223
224 template<class T>
225 const T& Matrix<T>::operator()(const int a, const int b) const {
226     return _values[a][b];
227 }
228
229 template<class T>
230 T& Matrix<T>::operator()(int a) {
231     return _values[a][0];
232 }

```

```

233
234 template<class T>
235 const T& Matrix<T>::operator ()(const int a) const {
236     return _values[a][0];
237 }
238
239 template<class T>
240 int Matrix<T>::rows() {
241     return _rows;
242 }
243
244 template<class T>
245 int Matrix<T>::columns() {
246     return _columns;
247 }
248
249 template<class T>
250 void Matrix<T>::printMatrix() {
251     for(int i = 0; i < _rows; i++) {;
252         for(int j = 0; j < _columns; j++) {
253             printf("%3.3f ", _values[i][j]);
254             // cout << _values[i][j] << " ";
255         }
256         cout << endl;
257     }
258     cout << endl;
259 }
260
261 #endif /* MATRIX.H */

```

7.2. eqsys.h

```

1  /*
2   * File:    eqsys.h
3   * Author:  Federico
4   *
5   * Created on August 17, 2015, 5:57 PM
6   */
7
8  #ifndef EQSYS_H
9  #define EQSYS_H
10
11  #include <algorithm>
12  #include <math.h>
13  #include <vector>
14  #include "matrix.h"
15
16  template<class T>
17  class EquationSystemLU {
18      public:
19          EquationSystemLU(const Matrix<T>& inicial);
20
21          Matrix<T> solve(Matrix<T>& b_values);
22
23      private:
24          Matrix<T> lower;
25          Matrix<T> upper;
26          bool isPermutated;
27          Matrix<T> permutation;

```

```

28 };
29
30 template<class T>
31 EquationSystemLU<T>::EquationSystemLU(const Matrix<T>& inicial)
32 : upper(inicial), isPermutated(false)
33 {
34     T coef;
35     int i, j, k, l;
36
37     // Armar la matriz lower
38     lower = Matrix<T>(upper.rows(), upper.columns(), 0);
39
40     for(i = 0; i < upper.columns(); i++) {
41         for(j = i + 1; j < upper.rows(); j++) {
42             if(upper(i, i) == 0) {
43                 // Hay que buscar la proxima fila sin cero
44                 for(k = i + 1; k < upper.rows(); k++) {
45                     if(upper(k, i) != 0) {
46                         break;
47                     }
48                 }
49
50                 if(k == upper.rows()) { // No hay filas para permutar
51                     abort();
52                 } else {
53                     if(!isPermutated){
54                         // Generamos la matriz de permutacion con uno en la diagonal
55                         isPermutated = true;
56                         permutation = Matrix<T>(upper.rows(), upper.columns(), 0);
57
58                         for(l = 0; l < permutation.rows(); l++) {
59                             permutation(l, l) = 1;
60                         }
61                     }
62                     // Permutamos las filas
63                     for(l = 0; l < permutation.columns(); l++) {
64                         if(l == k) {
65                             permutation(i, l) = 1;
66                         } else {
67                             permutation(i, l) = 0;
68                         }
69                         if(l == i) {
70                             permutation(k, l) = 1;
71                         } else {
72                             permutation(k, l) = 0;
73                         }
74                     }
75                     // Hacemos el producto para efectivamente permutar
76                     upper = permutation * upper;
77                     lower = permutation * lower;
78                 }
79             }
80
81             // Calculamos y guardamos el coeficiente
82             // cout << upper(j,i) << " , " << upper(i,i) << endl;
83             coef = upper(j, i) / upper(i, i);
84             lower(j, i) = coef;
85             // Colocamos cero en la columna bajo la diagonal
86             upper(j, i) = 0;

```

```

87         for(k = i + 1; k < upper.columns(); k++) {
88             upper(j, k) = upper(j, k) - coef * upper(i, k);
89         }
90     }
91 }
92 // Agrego la diagonal de unos a lower
93 for(i = 0; i < lower.rows(); i++){
94     lower(i, i) = 1;
95 }
96 }
97
98 template<class T>
99 Matrix<T> EquationSystemLU<T>::solve(Matrix<T>& b_values) {
100
101     Matrix<T> temp_values = Matrix<T>(b_values);
102     Matrix<T> y_values = Matrix<T>(b_values.rows());
103     Matrix<T> x_values = Matrix<T>(b_values.rows());
104
105     if(isPermutated) {
106         temp_values = permutation * temp_values;
107     }
108
109     // Resuelvo el sistema L * y = b
110     for(int i = 0; i < temp_values.rows(); i++) {
111         for (int j = 0; j < i; j++) {
112             temp_values(i) -= y_values(j) * lower(i, j);
113         }
114         if(i == 0) {
115             y_values(0) = temp_values(0) / lower(0,0); // Calculo aparte el primer
116                 valor
117         } else {
118             y_values(i) = temp_values(i) / lower(i, i);
119         }
120     }
121
122     // Resuelvo el sistema U * x = y
123     temp_values = y_values;
124     for(int i = temp_values.rows() - 1; i >= 0; i--) {
125         for (int j = temp_values.rows() - 1; j > i; j--) {
126             temp_values(i) -= x_values(j) * upper(i, j);
127         }
128         if(i == x_values.rows() - 1) {
129             x_values(x_values.rows() - 1) = temp_values(temp_values.rows() - 1) /
130                 upper(upper.rows() - 1, upper.columns() - 1);
131         } else {
132             x_values(i) = temp_values(i) / upper(i, i);
133         }
134     }
135
136     // Retorno la solucion al sistema LU * x = b
137     return x_values;
138 }
139
140
141 template<class T>
142 class EquationSystem{
143     public:

```

```

144     EquationSystem(const Matrix<T>& inicial);
145
146     Matrix<T> solve(const Matrix<T>& b_values);
147
148     private:
149         Matrix<T> _matrix;
150 };
151
152 template<class T>
153 EquationSystem<T>::EquationSystem(const Matrix<T>& inicial)
154     : _matrix(inicial)
155 {}
156
157 template<class T>
158 Matrix<T> EquationSystem<T>::solve(const Matrix<T>& b_values) {
159     T coef;
160     int i, j, k, l;
161     bool isPermutated;
162     Matrix<T> temp_matrix(_matrix);
163     Matrix<T> temp_values(b_values);
164     Matrix<T> permutation;
165
166     for(i = 0; i < temp_matrix.columns(); i++) {
167         for(j = i + 1; j < temp_matrix.rows(); j++) {
168             if(temp_matrix(i, i) == 0) {
169                 // Hay que buscar la proxima fila sin cero
170                 for(k = i + 1; k < temp_matrix.rows(); k++) {
171                     if(temp_matrix(k, i) != 0) {
172                         break;
173                     }
174                 }
175
176                 if(k == temp_matrix.rows()) { // No hay filas para permutar
177                     abort();
178                 } else {
179                     if(!isPermutated){
180                         // Generamos la matriz de permutacion con uno en la diagonal
181                         isPermutated = true;
182                         permutation = Matrix<T>(temp_matrix.rows(), temp_matrix.
183                                                 columns(), 0);
184
185                         for(l = 0; l < permutation.rows(); l++) {
186                             permutation(l, l) = 1;
187                         }
188                         // Permutamos las filas
189                         for(l = 0; l < permutation.columns(); l++) {
190                             if(l == k) {
191                                 permutation(i, l) = 1;
192                             } else {
193                                 permutation(i, l) = 0;
194                             }
195                             if(l == i) {
196                                 permutation(k, l) = 1;
197                             } else {
198                                 permutation(k, l) = 0;
199                             }
200                         }
201                         // Hacemos el producto para efectivamente permutar

```

```

202         temp_matrix = permutation * temp_matrix;
203         temp_values = permutation * temp_values;
204     }
205 }
206
207 // Calculamos y guardamos el coeficiente
208 coef = temp_matrix(j, i) / temp_matrix(i, i);
209 // Colocamos cero en la columna bajo la diagonal
210 temp_matrix(j, i) = 0;
211 for(k = i + 1; k < temp_matrix.columns(); k++) {
212     temp_matrix(j, k) = temp_matrix(j, k) - coef * temp_matrix(i, k);
213 }
214 temp_values(j) = temp_values(j) - coef * temp_values(i);
215 }
216 }
217
218 Matrix<T> x_values = Matrix<T>(temp_values.rows());
219
220 // Resuelvo el sistema A * x = b, con A triangular superior
221 for(int i = temp_values.rows() - 1; i >= 0; i--) {
222     for (int j = temp_values.rows() - 1; j > i; j--) {
223         temp_values(i) -= x_values(j) * temp_matrix(i, j);
224     }
225     if(i == x_values.rows() - 1) {
226         x_values(x_values.rows() - 1) = temp_values(temp_values.rows() - 1) /
            temp_matrix(temp_matrix.rows() - 1, temp_matrix.columns() - 1);
227     } else {
228         x_values(i) = temp_values(i) / temp_matrix(i, i);
229     }
230 }
231
232 // Retorno la solucion a A * x = b
233 return x_values;
234 }
235
236 #endif /* EQSYS_H */

```

7.3. generate.cpp

```

1  #include <iostream>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <vector>
5
6  #include "eqsys.h"
7
8  using namespace std;
9
10 void fprintframe(FILE* outputFile, int frame, int videoWidth, int videoHeight, vector
    <vector<int>> & video);
11 void fprintlinearframe(FILE* outputFile, int startFrame, int currentFrame, int
    framesToGenerate,
12     int videoWidth, int videoHeight, vector<vector<int>> & video)
    ;
13 void fprintframefromspline(FILE* outputFile, int frame, int currentNewFrame, int
    framesToGenerate,
14     int videoWidth, int videoHeight, vector<vector<int>>
        video, vector<vector<int>> storage);

```



```

15 void naturalCubicSplineBuildA(int framesToGenerate, int videoFrames, Matrix<double>&
    A);
16 void naturalCubicSplineBuildB(int pixel, int framesToGenerate, int videoFrames,
    Matrix<double>& b, vector<vector<int>>& video);
17
18 int main(int argc, char* argv[]) {
19
20     if (argc != 5) {
21         printf("Use: %s input.txt output.txt interpolationMethod framesToGenerate \n"
22             "Where interpolationMethod:\n"
23             "0: nearest neighbour \n"
24             "1: lineal \n"
25             "2: splines \n"
26             "3: other... \n"
27             "e.g. %s input.txt output.txt 0 5\n", argv[0], argv[0]);
28         return 0;
29     }
30
31     // load input file
32     FILE* inputFile = fopen(argv[1], "r");
33     if (inputFile == NULL) {
34         printf("Error: %s is not a valid input file.\n", argv[1]);
35         return 0;
36     }
37
38     // video metadata
39     int videoFrames, videoHeight, videoWidth, videoFrameRate;
40     fscanf(inputFile, "%d", &videoFrames);
41     fscanf(inputFile, "%d,%d", &videoHeight, &videoWidth);
42     fscanf(inputFile, "%d", &videoFrameRate);
43     printf("videoFrames: %d, videoHeight: %d, videoWidth: %d, videoFrameRate: %d\n",
44         videoFrames, videoHeight, videoWidth, videoFrameRate);
45
46     // video data
47     vector<vector<int>> video(videoFrames, vector<int>(videoWidth*videoHeight));
48
49     for (int frame = 0; frame < videoFrames; ++frame) {
50         for (int i = 0; i < videoHeight; ++i) {
51             for (int j = 0; j < videoWidth; ++j) {
52                 fscanf(inputFile, "%d,", &video[frame][i*videoWidth + j]);
53             }
54         }
55     }
56
57     // output file
58     FILE* outputFile = fopen(argv[2], "w");
59     if (outputFile == NULL) {
60         printf("Error: Failed to create output file.\n");
61         return 0;
62     }
63
64     // select method
65     int interpolationMethod = atoi(argv[3]);
66     int framesToGenerate = atoi(argv[4]);
67     int totalFrames = videoFrames + framesToGenerate * (videoFrames - 1);
68
69     switch (interpolationMethod) {
70         case 0:
71             printf("interpolationMethod: Nearest Neighbour\n");

```

```

72
73 // write header
74 fprintf(outputFile, "%d\n%d,%d\n%d\n", totalFrames, videoHeight,
    videoWidth, videoFrameRate);
75
76 // generate frames
77 for (int frame = 0; frame < videoFrames - 1; ++frame) {
78
79     // print current frame
80     fprintfframe(outputFile, frame, videoWidth, videoHeight, video);
81
82     // closer to first frame
83     for (int j = 0; j < framesToGenerate/2; ++j) {
84         fprintfframe(outputFile, frame, videoWidth, videoHeight, video);
85     }
86
87     // closer to second frame
88     for (int j = framesToGenerate/2; j < framesToGenerate; ++j) {
89         fprintfframe(outputFile, frame + 1, videoWidth, videoHeight, video
            );
90     }
91 }
92
93 // print last frame
94 fprintfframe(outputFile, videoFrames-1, videoWidth, videoHeight, video);
95
96 break;
97 case 1:
98     printf("interpolationMethod: Lineal\n");
99
100 // write header
101 fprintf(outputFile, "%d\n%d,%d\n%d\n", totalFrames, videoHeight,
    videoWidth, videoFrameRate);
102
103 for (int frame = 0; frame < videoFrames - 1; ++frame) {
104
105     // print current frame
106     fprintfframe(outputFile, frame, videoWidth, videoHeight, video);
107
108     // generate frame, initial frame: frame 0 (current frame)
109     for (int j = 1; j <= framesToGenerate; ++j) {
110         fprintflinearframe(outputFile, frame, j, framesToGenerate,
            videoWidth, videoHeight, video);
111     }
112 }
113
114 // print last frame
115 fprintfframe(outputFile, videoFrames-1, videoWidth, videoHeight, video);
116
117 break;
118 case 2: {
119     printf("interpolationMethod: Splines\n");
120
121     // save values of the polinomial coefficients for each pixel (c_j)
122     vector<vector<int>> > storage(videoWidth*videoHeight, vector<int>>(
        videoFrames));
123
124
125

```

```

126     Matrix<double> A(videoFrames, videoFrames, 0);
127     naturalCubicSplineBuildA(framesToGenerate, videoFrames, A);
128     EquationSystemLU<double> e(A);
129
130     Matrix<double> b(videoFrames);
131
132     // fit a spline on every pixel
133     for (int i = 0; i < videoWidth*videoHeight; ++i) {
134         naturalCubicSplineBuildB(i, framesToGenerate, videoFrames, b, video);
135
136         Matrix<double> result(e.solve(b));
137
138         // store results
139         for (int j = 0; j < videoFrames; ++j) {
140             storage[i][j] = result(j);
141         }
142     }
143
144     // write header
145     fprintf(outputFile, "%d\n%d,%d\n%d\n", totalFrames, videoHeight,
146         videoWidth, videoFrameRate);
147
148     // generate frames
149     for (int frame = 0; frame < videoFrames - 1; ++frame) {
150
151         // print current frame
152         fprintfframe(outputFile, frame, videoWidth, videoHeight, video);
153
154         // generate new frames
155         for (int j = 1; j <= framesToGenerate; ++j) {
156             fprintfframefromspline(outputFile, frame, j, framesToGenerate,
157                 videoWidth, videoHeight, video, storage);
158         }
159     }
160
161     // print last frame
162     fprintfframe(outputFile, videoFrames-1, videoWidth, videoHeight, video);
163
164     break;
165 }
166 default:
167     printf("Error: Invalid interpolation method\n");
168     return 0;
169 }
170
171 fclose(inputFile);
172 fclose(outputFile);
173
174 return 0;
175 }
176
177 // new frames are counted from 1.
178 void fprintfframefromspline(FILE* outputFile, int frame, int currentNewFrame, int
    framesToGenerate,
179     int videoWidth, int videoHeight, vector<vector<int>>
    video, vector<vector<int>> storage) {
180

```

```

181     int h = framesToGenerate + 1;
182
183     for (int pixel = 0; pixel < videoWidth*videoHeight; ++pixel) {
184         int c_0 = storage[pixel][frame]; // !
185         int c_1 = storage[pixel][frame+1];
186         // frame >= 1
187         int a_0 = video[frame][pixel]; // !
188         int a_1 = video[frame+1][pixel];
189         int b_0 = (1/h)*(a_1 - a_0) - (h/3)*(2*c_0 + c_1); // !
190         int d_0 = (c_1 - c_0) / (3*h); // !
191
192         int x = frame*h;
193         int x_j = frame*h + currentNewFrame;
194         int res = a_0 + b_0*(x-x_j) + c_0*pow(x-x_j,2) + d_0*pow(x-x_j,3);
195
196         if ((pixel+1) % videoWidth == 0) {
197             fprintf(outputFile, "%d\n", res);
198         } else {
199             fprintf(outputFile, "%d,", res);
200         }
201     }
202 }
203 }
204
205 /* Possible improvements:
206 * 1. Improve cache locality by saving one vector per pixel, right now a single pixel
207   is in several vectors.
208 * 2. LU Factorization on matrix A. It doesn't depend on the pixel being processed.
209   Only b changes. Ax = b
210 * 3. A is sparse! Better representations!
211 * Reminder: A is strictly diagonally dominant, LU factorization without pivoting is
212   valid.
213 */
214 void naturalCubicSplineBuildA(int framesToGenerate, int videoFrames, Matrix<double>&
A) {
215     A(0,0) = 1;
216     A(videoFrames-1,videoFrames-1) = 1;
217     /* h_i = x_{j+1} - x_{j}
218     * h is CONSTANT! (pixels are equidistant)
219     */
220     int h_0 = framesToGenerate + 1;
221     int h_1 = framesToGenerate + 1;
222     int h_2 = framesToGenerate + 1;
223     for (int i = 1; i < videoFrames - 1; ++i) {
224         // h_i = x_{j+1} - x_{j}
225         A(i, i-1) = h_0;
226         A(i, i) = 2*(h_0+h_1);
227         A(i, i+1) = h_1;
228     }
229 }
230
231 void naturalCubicSplineBuildB(int pixel, int framesToGenerate, int videoFrames,
Matrix<double>& b, vector<vector<int>>>& video) {
232     int h = framesToGenerate + 1;
233     b(0) = 0;
234     b(videoFrames-1) = 0;
235     for (int i = 1; i < videoFrames - 1; ++i) {
236         // a_i = f(x_i)
237         int a_0 = video[i-1][pixel];

```

```

235         int a_1 = video[i][pixel];
236         int a_2 = video[i+1][pixel];
237         b(i) = (3/h) * (a_2 - a_1) - (3/h) * (a_1 - a_0);
238     }
239 }
240
241 void fprintframe(FILE* outputFile, int frame, int videoWidth, int videoHeight, vector
    <vector<int>>& video) {
242     for (int i = 0; i < videoHeight; ++i) {
243         for (int j = 0; j < videoWidth - 1; ++j) {
244             fprintf(outputFile, "%d,", video[frame][i*videoWidth + j]);
245         }
246         fprintf(outputFile, "%d\n", video[frame][i*videoWidth + videoWidth - 1]);
247     }
248 }
249
250 void fprintlinearframe(FILE* outputFile, int startFrame, int currentFrame, int
    framesToGenerate,
251                        int videoWidth, int videoHeight, vector<vector<int>>& video)
    {
252     for (int i = 0; i < videoHeight; ++i) {
253         for (int j = 0; j < videoWidth - 1; ++j) {
254             int y_0 = video[startFrame][i*videoWidth + j];
255             int y_1 = video[startFrame+1][i*videoWidth + j];
256             int m = (y_1 - y_0) / (framesToGenerate + 1 - 0);
257             int b = y_0 - m*0;
258             fprintf(outputFile, "%d,", m*currentFrame + b);
259         }
260         int y_0 = video[startFrame][i*videoWidth + videoWidth - 1];
261         int y_1 = video[startFrame+1][i*videoWidth + videoWidth - 1];
262         int m = (y_1 - y_0) / (framesToGenerate + 1 - 0);
263         int b = y_0 - m*0;
264         fprintf(outputFile, "%d,", m*currentFrame + b);
265     }
266 }

```
