DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

Métodos Numéricos
Segundo Cuatrimestre de 2015

Integrante	LU	Correo electrónico
Iván Arcuschin	678/13	iarcuschin@gmail.com
Martín Jedwabny	885/13	martiniedva@gmail.com
José Massigoge	954/12	jmmassigoge@gmail.com
Iván Pondal	078/14	ivan.pondal@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Resumen

Hecho: los videos de bebes son graciosos. Y son mucho más graciosos cuando se miran en *slow motion*.

En este trabajo, nos centraremos en 4 diferentes métodos de Interpolación para generar automáticamente videos en cámara lenta. Esto se logra interpolando una cierta cantidad de cuadros nuevos entre cada par de cuadros originales. Dependiendo del método, este procedimiento genera diferentes niveles de calidad.

Una vez que hayamos explicado los métodos junto con su implementación, presentaremos varios experimentos, dentro de dos categorías. La primera, asegurará que los métodos funcionen correctamente, en el caso general de interpolación de funciones. La segunda, proveerá diferentes formas de comparar la calidad de los videos producidos por cada método.

Keywords: Video, slow motion, interpolation methods, splines

Índice

1. Introducción

El objetivo principal de este Trabajo Práctico es estudiar, implementar y analizar métodos de Interpolación para generar Videos en *slow motion*.

Comenzaremos haciendo una breve introducción a los distintos métodos de Interpolación, para luego explicar cual es el modelo que subyace en cada uno:

- Interpolación por Vecinos.
- Interpolación Fragmentaria Lineal.
- Interpolación por Splines.
- Interpolación por Splines (bloques de tamaño fijo).

Una vez finalizada la parte del Modelo, pasaremos a describir la Implementación de los diferentes métodos presentados, realizadas en `C++`.

Ya llegando al final, pasaremos a presentar la Experimentación realizada, a la vez que iremos analizando y discutiendo los resultados obtenidos.

Los experimentos realizados pueden dividirse en dos categorías. La primera, relacionada con el costo temporal y la correctitud de los diversos algoritmos utilizados:

- Funcionamiento de los métodos implementados al tratar de interpolar diferentes familias de funciones.
- Comparación de tiempos de ejecución entre los distintos métodos.

La segunda, relacionada con el aspecto cualitativo de los métodos:

- Comparación del *Error Cuadrático Medio* y *Peak to Signal Noise Rate* entre los distintos métodos.
- Análisis del fenómeno de Artifacts.

Para finalizar, cerraremos el presente informe con una conclusión, en la cual discutiremos acerca de los métodos vistos, así como de la experimentación realizada. También, contaremos las dificultades encontradas al realizar el Trabajo Práctico, las posibles continuaciones que se podrían realizar, y si los objetivos planteados fueron alcanzados.

2. Modelo

2.1. Video

Definiremos un modelo para los videos con el cual sea fácil de trabajar a la hora de realizar el *slow motion*. Dado un video, definiremos:

- w el ancho en píxeles de cada frame.
- h el ancho en píxeles de cada frame.
- f_i el i-ésimo frame, con $0 < i < k$, donde k es la cantidad de frames totales.
- $p(x, y, f_i)$, con $0 < x < w$, $0 < y < h$, el píxel en la posición (x, y) del frame f_i.

Luego, si tomamos $p(x, y, f_i)$ y $p(x, y, f_{i+1})$ querremos agregar una cierta cantidad de píxeles entre ambos, de forma que haya una transición del primero al segundo y se produzca el *slow motion*.

Para elegir que valores agregar entre los píxeles, utilizaremos diferentes métodos de interpolación.

- *Vecinos*: Consiste en rellenar los nuevos frames replicando los valores de los píxeles del frame original más cercano.
- *Interpolación Lineal*: Consiste en rellenar los píxeles utilizando interpolaciones lineales entre píxeles de frames originales consecutivos.
- *Interpolación por Splines*: Consiste en rellenar los píxeles utilizando Splines entre píxeles de frames originales consecutivos. En este método, utilizaremos la información provista por todos los frames del video, y generaremos $k - 1$ funciones, cada una de a lo sumo grado cúbico.
- *Interpolación por Splines con tamaño de bloque variante*: Similar al anterior, pero con la posibilidad de variar la cantidad de frames tomados en cuenta al generar las funciones.

En las siguientes secciones explicaremos con mayor detalle cada uno de los métodos.

2.2. Vecinos

En este método, elegiremos para cada nuevo píxel el valor del frame original que se encuentre más cercano. Si definimos c como la cantidad de frames a agregar entre cada par original, y $g_0, \ldots, g_{c-1}$ los nuevos frames. Tenemos que:

$$\begin{aligned}
p(x, y, f_i) &= p(x, y, f_i) \\
p(x, y, g_0) &= p(x, y, f_i) \\
&\vdots \\
p(x, y, g_{c/2-1}) &= p(x, y, f_i) \\
p(x, y, g_{c/2}) &= p(x, y, f_{i+1}) \\
&\vdots \\
p(x, y, g_{c-1}) &= p(x, y, f_{i+1}) \\
p(x, y, f_{i+1}) &= p(x, y, f_{i+1})
\end{aligned}$$

2.3. Interpolación Fragmentaria Lineal

En este método, buscaremos interpolar los píxeles de frames contiguos con una función lineal. Para ello, construiremos un Polinomio Interpolante de grado 1 utilizando *diferencias divididas*, ya que ofrece una construcción más sencilla que al seguir el método de Lagrange.

Luego, si llamamos f a la función (desconocida excepto en los puntos x_j), definimos:

- Diferencia dividida de orden cero en x_j:

$$f[x_j] = f(x_j)$$

- Diferencia dividida de orden uno en x_j, x_{j+1}:

$$f[x_j, x_{j+1}] = \frac{f[x_{j+1}] - f[x_j]}{x_{j+1} - x_j} = \frac{f(x_{j+1}) - f(x_j)}{x_{j+1} - x_j}$$

- Polinomio Interpolante de grado 1 para x_j, x_{j+1}:

$$P_1(x) = f[x_j] + f[x_j, x_{j+1}](x - x_j) = f(x_j) + \frac{f(x_{j+1}) - f(x_j)}{x_{j+1} - x_j} * (x - x_j)$$

2.4. Interpolación por Splines

El método de interpolación por splines se basa en dados n puntos la construcción de $n-1$ funciones que interpolan los puntos y además cumplen una serie de condiciones que aseguran que la función por tramos resultante no posea las irregularidades de trabajar con polinomios de alto grado generando además uniones suaves entre cada segmento.

Dada una función f definida en el intervalo $[a, b]$ y un conjunto de nodos $a = x_0 < x_1 < \ldots < x_n = b$.

1. Definimos $S(x)$ como un polinomio cúbico denominándolo $S_j(x)$ en el subintervalo $[x_j, x_{j+1}]$ con $j \in [0, \ldots, n-1]$.
2. $S_j(x_j) = f(x_j)$ y $S_j(x_{j+1}) = f(x_{j+1})$ para todo $j \in [0, \ldots, n-1]$.
3. $S_j(x_{j+1}) = S_j(x_{j+1})$ para todo $j \in [0, \ldots, n-2]$.
4. $S'_j(x_{j+1}) = S'_j(x_{j+1})$ para todo $j \in [0, \ldots, n-2]$.
5. $S''_j(x_{j+1}) = S''_j(x_{j+1})$ para todo $j \in [0, \ldots, n-2]$.
6. Por último, se cumple una de la siguientes condiciones
   a) $S''(x_0) = S''(x_n) = 0$ (natural o de libre frontera).
   b) $S'(x_0) = f'(x_0)$ y $S'(x_n) = f'(x_n)$ (sujeta).

Un spline definido en un intervalo que está dividido en n subintervalos requiere determinar $4n$ constantes. Se aplican las condiciones descritas previamente a los siguiente polinomios cúbicos:

$$S_j(x) = a_j + b_j(x - x_j) + c_j(x - x_j)^2 + d_j(x - x_j)^3$$
$$\text{para cada } j \in [0, \ldots, n-1]$$

Para este trabajo, dado que no conocemos la función f que estamos interpolando se decidió utilizar la condición 6*a*, que define al spline como natural o libre.

Una vez establecidas las ecuaciones resultantes de aplicar las condiciones y despejando todas las variables en función de c_j[1], nos quedan las siguientes igualdades:

$$\begin{aligned} h_j &= x_{j+1} - x_j \\ a_j &= f(x_j) \text{ para cada } j \in [0, \ldots, n] \\ b_j &= \frac{a_{j+1} - a_j}{h_j} - \frac{2c_j h_j - c_{j+1} h_j}{3} \\ d_j &= \frac{c_{j+1} - c_j}{3h_j} \\ &\text{para cada } j \in [0, \ldots, n-1] \end{aligned}$$

Donde los c_j nos quedan determinados por el siguiente sistema de ecuaciones:

$$c_0 = 0$$
$$c_n = 0$$
$$h_{j-1}c_{j-1} + 2(h_{j-1} + h_j)c_j + h_j c_{j+1} = \frac{3(a_{j+1} - a_j)}{h_j} + \frac{3(a_{j-1} - a_j)}{h_{j-1}}$$

para cada $j \in [1, \ldots, n-1]$

El mismo se puede representar como la siguiente matriz:

$$\begin{array}{c} \\ c_0 \\ c_1 \\ \vdots \\ c_j \\ \vdots \\ c_{n-1} \\ c_n \end{array} \begin{array}{c} \begin{array}{ccccccccccc} c_0 & c_1 & c_2 & \ldots & c_{j-1} & c_j & c_{j+1} & \ldots & c_{n-2} & c_{n-1} & c_n \end{array} \\ \left[\begin{array}{ccccccccccc} 1 & 0 & 0 & \ldots & 0 & 0 & 0 & \ldots & 0 & 0 & 0 \\ h_0 & 2(h_0+h_1) & h_1 & \ldots & 0 & 0 & 0 & \ldots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & h_{j-1} & 2(h_{j-1}+h_j) & h_j & \ldots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & 0 & 0 & 0 & \ldots & h_{n-2} & 2(h_{n-2}+h_{n-1}) & h_{n-1} \\ 0 & 0 & 0 & \ldots & 0 & 0 & 0 & \ldots & 0 & 0 & 1 \end{array}\right] \end{array} \begin{array}{c} b \\ \left|\begin{array}{c} 0 \\ \frac{3(a_2-a_1)}{h_1} + \frac{3(a_0-a_1)}{h_0} \\ \vdots \\ \frac{3(a_{j+1}-a_j)}{h_j} + \frac{3(a_{j-1}-a_j)}{h_{j-1}} \\ \vdots \\ \frac{3(a_n-a_{n-1})}{h_{n-1}} + \frac{3(a_{n-2}-a_{n-1})}{h_{n-2}} \\ 0 \end{array}\right] \end{array}$$

Ahora para el problema planteado, que es la interpolación de los cuadros de un video, nuestro h_j será la distancia entre cada uno de ellos. Esta distancia la podemos pensar como el tiempo entre cada captura del video, y como la duración del mismo se define por la cantidad de cuadros por segundo, podemos afirmar que son equidistantes, por lo tanto tenemos $h_j = 1$ para todo $j \in [0, \ldots, n-1]$.

Reemplazando en la matriz anterior nos queda:

$$\begin{array}{c} \\ c_0 \\ c_1 \\ \vdots \\ c_j \\ \vdots \\ c_{n-1} \\ c_n \end{array} \begin{array}{c} \begin{array}{ccccccccccc} c_0 & c_1 & c_2 & \ldots & c_{j-1} & c_j & c_{j+1} & \ldots & c_{n-2} & c_{n-1} & c_n \end{array} \\ \left[\begin{array}{ccccccccccc} 1 & 0 & 0 & \ldots & 0 & 0 & 0 & \ldots & 0 & 0 & 0 \\ 1 & 4 & 1 & \ldots & 0 & 0 & 0 & \ldots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & 1 & 4 & 1 & \ldots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & 0 & 0 & 0 & \ldots & 1 & 4 & 1 \\ 0 & 0 & 0 & \ldots & 0 & 0 & 0 & \ldots & 0 & 0 & 1 \end{array}\right] \end{array} \begin{array}{c} b \\ \left|\begin{array}{c} 0 \\ 3(a_2 - a_1) + 3(a_0 - a_1) \\ \vdots \\ 3(a_{j+1} - a_j) + 3(a_{j-1} - a_j) \\ \vdots \\ 3(a_n - a_{n-1}) + 3(a_{n-2} - a_{n-1}) \\ 0 \end{array}\right] \end{array}$$

Como se puede observar resulta en una matriz cuadrada tridiagonal estrictamente dominante por filas. Como consecuencia tenemos que la misma y sus submatrices principales son inversibles, llevando a que el sistema tenga una única solución y que además A posea factorización LU.

Una vez que se tiene la factorización $A = LU$, sólo queda resolver el sistema planteado y así calcular los coeficientes de cada $S_j(x)$. Por último resta evaluar el spline en los puntos donde se agregaron cuadros nuevos para así poder lograr el efecto buscado.

El beneficio de utilizar la factorización LU es que el spline que se utiliza para interpolar cada pixel del video comparte el mismo sistema, lo que cambia son los valores de los a_j que definen nuestro vector b. De esta manera, recalculamos el vector b, utilizamos la factorización LU para resolver el sistema y nuevamente calculamos los coeficientes del spline.

2.5. Interpolación por Splines con tamaño de bloque fijo

Este método tiene el mismo fondo teórico que el anterior ya que también trabaja con splines. La diferencia es que en el modelo anterior, dados todos los valores que tomaba cada pixel del video se generaba un único spline para cada pixel, mientras que con splines de a bloques lo que hacemos es para cada pixel generar varios splines conectados entre si.

La motivación para realizar esto es la hipótesis de que si estamos trabajando con un video que contiene cambios bruscos, el utilizar toda la duración del video para alimentar un spline puede resultar impreciso,

mientras que si generamos splines asociados a tramos del video, creemos que podría comportarse mejor a variaciones abruptas.

Dada una función f definida en el intervalo $[a, b]$ y un conjunto de nodos $a = x_0 < x_1 < \ldots < x_n = b$, definimos B como el tamaño de cada bloque y $Sb(x)$ como un spline de a bloques con $Sb_j(x)$ siendo el spline definido en el subintervalo $[x_j, x_{j+B}]$ con $j \in [0, \ldots, n - B - 1]$ que interpola los puntos de f en ese mismo intervalo.

3. Implementación

3.1. Interpolación por vecinos

Este método consiste en reemplazar los cuadros intermedios a ser rellenados por el cuadro original mas cercano en el tiempo. Es decir, dados los cuadros del video sin cámara lenta, generamos otro video en cámara lenta copiando los cuadros originales de la siguiente manera:

Sean Frame1 y Frame2 dos cuadros consecutivos del video original:

Frame1	Frame2

Si queremos ahora 6 cuadros entre cada 2 del archivo original lo transformamos a:

Frame1	Frame1	Frame1	Frame1	Frame2	Frame2	Frame2	Frame2

El pseudocódigo sería el siguiente:

```
Sean W,H,I el ancho, alto y la cantidad de frames del video original
Sea video[W][H][I] el triple vector de números enteros que representa el
    video original
Sea K la cantidad de frames que queremos agregar entre cuadro y cuadro

Crear un triple vector de enteros new_video[W][H][I+(I-1)*K]
Para w = 0 hasta W-1 hacer
  Para h = 0 hasta H-1 hacer
    Para i = 0 hasta I-2 hacer
      Para j = 0 hasta K/2 hacer
        new_video[w][h].push_back(video[w][h][i])
      Fin para
      Para j = (K/2)+1 hasta K hacer
        new_video[w][h].push_back(video[w][h][i+1])
      Fin para
    Fin para
    new_video[w][h].push_back(video[w][h][I-1])
  Fin para
Fin para
Devolver new_video
```

3.2. Interpolación lineal

En este caso, usamos el polinomio interpolador de Lagrange entre cada par de puntos/píxeles consecutivos para aproximar los valores intermedios que irían en el video de cámara lenta. Esto genera una función lineal para los píxeles consecutivos en la misma posición.

Por ejemplo, sean dos píxeles con valores 1 y 4:

1	4

Si queremos un video en cámara lenta con 5 cuadros intermedios por cada 2 del original, estos se replicarán de la siguiente forma:

1	1.5	2	2.5	3	3.5	4

El procedimiento es el siguiente:

```
Sean W,H,I el ancho, alto y la cantidad de frames del video original
Sea video[W][H][I] el triple vector de números enteros que representa el
    video original
Sea K la cantidad de frames que queremos agregar entre cuadro y cuadro

Crear un triple vector de enteros new_video[W][H][I+(I-1)*K]
Para w = 0 hasta W-1 hacer
  Para h = 0 hasta H-1 hacer
    Para i = 0 hasta I-2 hacer
      coef_cero = video[w][h][i]
      coef_uno = (video[w][h][i+1] - video[w][h][i]) / (K+1);
      Para k = 0 hasta K hacer
        píxel = coef_cero + coef_uno*k;
        Si (píxel < 0) píxel = 0
        Si (píxel > 255) píxel = 255
        new_video.push_back(píxel)
    Fin para
    new_video.push_back(video[w][h][I-1])
  Fin para
Fin para
Devolver new_video
```

3.3. Interpolación por Splines

En este método aplicamos la técnica de Splines. Esta consiste en generar un sistema de ecuaciones para encontrar una función por partes que interpole cada par de puntos con polinomios de forma que la curva resultante sea continua y dos veces derivable. Como el sistema es tridiagonal, podemos aprovechar para guardar los valores de la matriz de forma más eficiente. El pseudocódigo es el siguiente:

```
Sean W,H,I el ancho, alto y la cantidad de frames del video original
Sea video[W][H][I] el triple vector de números enteros que representa el
    video original
Sea K la cantidad de frames que queremos agregar entre cuadro y cuadro

Crear un triple vector de enteros new_video[W][H][I+(I-1)*K]
Para w = 0 hasta W-1 hacer
  Para h = 0 hasta H-1 hacer
    Crear un vector de enteros valores[I+(I-1)*K]
    GenerarSpline(video[w][h], valores, I, K)
    new_video[w][h] = valores
  Fin para
Fin para
Devolver new_video

GenerarSpline(y, valores, I, K):
  n = y.size()
  Crear doble vector de enteros sistema[n][2]
  sistema[0] = {1,0}
  Para j = 1 hasta n-2 hacer
    sistema[j] = {1, 4}
  Fin para
  sistema[n-1] = {0,1}
  // Factorización LU
  Para j = 1 hasta n-2 hacer
```

```
    coef = sistema[i + 1][0]/sistema[i][1];
    sistema[i + 1][0] = coef
    sistema[i + 1][1] -= coef
  Fin para
  Crear vectores x[n], a[n], b[n], c[n], d[n]
  a = y
  Para i = 1 hasta n-2 hacer
    x[i] = 3*(a[i + 1] - 2*a[i] + a[i - 1]);
    x[i] -= x[i - 1]*sistema[i][0];
  Fin para
  // Resuelvo triangular superior (Uc = x)
  Para i = n - 2 hasta 1 hacer
    c[i] = x[i];
    c[i] -= c[i + 1];
    c[i] /= sistema[i][1];
  Fin para
  // Calculo mis coeficientes "b" y "d"
  Para i = 0 hasta n-2 hacer
    b[i] = a[i + 1] - a[i] - (2*c[i] + c[i + 1])/3;
    d[i] = (c[i + 1] - c[i])/3;
  Fin para
  //Calculo los píxeles resultantes con el Spline
  Para i = 0 hasta n-1 hacer
    Para j = 0 hasta K hacer
      dif = j/(K+1)
      val = a[i] + b[i]*(dif) + c[i]*(dif)*(dif) + d[i]*(dif)*(dif)*(dif
          )
      valores.push_back(val)
    Fin para
  Fin para
  valores.push_back(y[n-1])
```

3.4. Interpolación por Splines de a bloques

A diferencia del método anterior, en este caso vamos a querer aplicar la técnica de Splines a bloques de píxeles de tamaño fijo. Es decir, para una misma posición del video, en vez de utilizar todos los valores del píxel a través del tiempo, generamos splines entre particiones de la misma longitud. El procedimiento es:

```
Sean W,H,I el ancho, alto y la cantidad de frames del video original
Sea video[W][H][I] el triple vector de números enteros que representa el
    video original
Sea K la cantidad de frames que queremos agregar entre cuadro y cuadro
Sea T el tamanio de bloque de los Splines
Observación: reutilizamos el método GenerarSpline() de los Splines
   normales
Observación: asumimos que I es divisible por T para simplificar este
   pseudocódigo

Crear un triple vector de enteros new_video[W][H][I+(I-1)*K]
Para w = 0 hasta W-1 hacer
  Para h = 0 hasta H-1 hacer
    Para t = 0 hasta T-1 hacer
      Crear subvector aux[T] = video[w][h][T*t..T*(t+1)-1]
```

```
      Crear vector bloque[T+(T-1)*K]
      GenerarSpline(aux, bloque, T, K)
      Pushear a new_video[w][h] todos los elementos de 'bloque' excepto
         el ultimo
    Fin para
    new_video[w][h].push_back(video[w][h][I-1])
  Fin para
Fin para
Devolver new_video
```

4. Experimentación

En esta sección, se detallan los diferentes experimentos que realizamos para medir el funcionamiento, la eficiencia y calidad de resultados, tanto de forma cuantitativa como cualitativa, de los métodos implementados.

Para lograr tal fin realizamos los siguientes tipos de experimentos:

- Funcionamiento de los métodos implementados: Mostraremos que los métodos de interpolación funcionan correctamente comparándolos contra diferentes familias de funciones. A su vez, basándonos en las precisiones obtenidas, determinaremos el tamaño de bloque óptimo para el método Interpolación por Splines.
- Medición del ECM y PSNR de los métodos: Compararemos los errores obtenidos en varias instancias de pruebas para los distintos métodos.
- Medición de los tiempos de ejecución de los métodos: Compararemos los tiempos de ejecución utilizando los distintos métodos.
- Análisis cualitativos de los métodos, fenómeno de artifacts: Buscaremos reconocer defectos de interpolación en los videos generados.

Los videos utilizados para los diversos experimentos, fueron los siguientes:

- **Video 1 - Skate**: 426x240, cantidad de cuadros originales: 151 , fps: 30, duración: 5s.
- **Video 2 - Messi**: 426x240, cantidad de cuadros originales: 151, fps: 30, duración: 5s.
- **Video 3 - Amanecer**: 426x240, cantidad de cuadros originales: 151, fps: 30, duración: 5s.

Es importante mencionar que cada video representa una clase de video distinto, en donde el Video 1 contiene movimientos bruscos, el Video 2 cambios de cámara, y el Video 3 movimientos suaves.

El motivo de estas elecciones se debe a la búsqueda de diversos *artifacts* a partir de las características de cada clase.

4.1. Detalles generales de la experimentación

- En los experimentos que se utilizaron números aleatorios, se generaron utilizando la función *rand*, provista por la librería `stdlib.h`.
- La semilla para los números aleatorios se seteo utilizando el método *srand(time(NULL))*, para evitar repeticiones de números en diferentes corridas.
- Las instancias de prueba fueron generadas con los archivos provistos por la cátedra. Adicionalmente, hicimos nuestras propias instancias emulando diferentes funciones (por ejemplo una función constante, lineal y cuadrática) para realizar el control de calidad de los métodos.
- Para medir los tiempos utilizamos la librería *chrono* y medimos los resultados en nanosegundos.
- A su vez, utilizamos el nivel de optimización *O2* de *C++* a la hora de compilar el código.
- Todos los tests fueron corridos en la misma máquina bajo las mismas condiciones.

4.2. Funcionamiento de los métodos implementados

En este experimento nuestro objetivo fue asegurarnos el correcto funcionamiento de nuestra implementación de la interpolación fragmentaria lineal, interpolación por splines, e interpolación por splines con tamaño de bloque fijo, tomando bloques de 2, 4, 8, 16, 32 y 64 cuadros.

Con este fin, realizamos una serie de tests que muestran el correcto funcionamiento de cada método para distintas familias de funciones:

- Función constante.
- Función lineal.
- Función cuadrática.
- Función cúbica.

Luego, cada método de interpolación fue testeado contra cada una de las familias de funciones mencionadas de la siguiente forma:

- Dada una familia de funciones, se generan aleatoriamente los coeficientes necesarios para definir una función de esa familia, i.e.: para una constante se genera solo el coeficiente independiente, mientras que para una cuadrática se generan 3 coeficientes.
- Una vez generada la función, se la evalúa en un rango de valores para obtener un array de valores esperados.
- Luego, a partir del array de valores esperados se construye otro array quitándole elementos a intervalos fijos. Este nuevo array será el utilizado para realizar la interpolación, y lo que testearemos es la aproximación de la interpolación a los elementos que quitamos.
- Una vez que tenemos la interpolación con cualquiera de los métodos mencionados, basta recorrer los elementos del array de valores esperados a la vez que evaluamos la interpolación obtenida. Para cada par de valores: esperado e interpolado, querremos ver que la diferencia absoluta es menor que un épsilon/cota de precisión que definiremos dependiendo del método utilizado y la función a interpolar.

Es importante mencionar algunas características de las instancias utilizadas:

- Cantidad de puntos generados con la función(tamaño del array de valores esperados): 100
- Cantidad de puntos a interpolar: 50.
- Todos los coeficientes generados aleatoriamente están en el rango $[1, 10]$, para evitar que las funciones generadas crezcan de forma desmedida.

Luego, se obtuvieron las siguientes cotas de precisión para los distintos métodos y funciones:

	F. Constante	F. Lineal	F. Cuadrática	F. Cúbica
Interpolación por Vecinos	0.0001	10	1000	100000
Interpolación Fragmentaria Lineal	0.0001	0.0001	10	1000
Interpolación por Splines (bloques tamaño 2)	0.0001	0.0001	10	1000
Interpolación por Splines (bloques tamaño 4)	0.0001	0.0001	5	500
Interpolación por Splines (bloques tamaño 8)	0.0001	0.0001	1	200
Interpolación por Splines (bloques tamaño 16)	0.0001	0.0001	1	200
Interpolación por Splines (bloques tamaño 32)	0.0001	0.0001	1	200
Interpolación por Splines (bloques tamaño 64)	0.0001	0.0001	1	200
Interpolación por Splines (1 solo bloque)	0.0001	0.0001	1	200

Analizando dichas cotas vemos que:

- La Interpolación por Vecinos resulta razonable solo para funciones con muy poca variación (derivada a lo sumo constante). Esto se ve claramente en la cota de precisión al interpolar una función Cuadrática.
- La Interpolación Fragmentaria Lineal es sustancialmente mejor que por Vecinos, y devuelve resultados razonables para funciones a lo sumo Cuadráticas.
- La Interpolación por Splines utilizando solo 2 puntos para cada bloque es equivalente a interpolar utilizando funciones lineales, y queda evidenciado al tener las mismas cotas de precisión.

- A partir de los tamaños de bloque 4 a 16, la Interpolación por Splines realiza una mejora "asintótica" de su cota de precisión.
- Entre los tamaños de bloque 16 a 64, no se notó una mejora significativa en la cota de precisión de la Interpolación por Splines.
- Al realizar Interpolación por Splines *standard* (1 solo bloque), vemos que su cota de precisión concuerda con las cotas a las cuales "convergen" los métodos de Interpolación por Splines con bloque de tamaño fijo. Es por esta razón que para los experimentos cualitativos, utilizaremos solamente la Interpolación por Splines *standard*, y no la de tamaño de bloques fijo.

4.3. Medición de los tiempos de ejecución de los métodos

A partir de la implementaciones descriptas en la Sección 3, podemos inferir una complejidad temporal para los métodos vecinos, interpolación fragmentaria lineal e interpolación por splines.

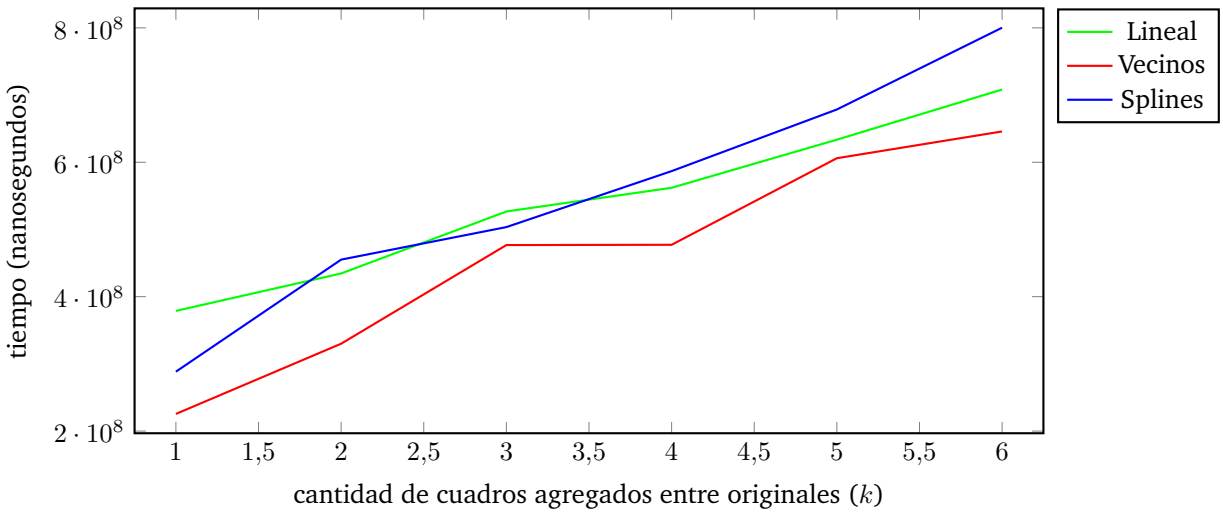
Sea w la cantidad de filas de píxeles en cada imagen, h la cantidad de columnas, i la cantidad de cuadros originales y sea k la cantidad de cuadros a agregar entre los originales:

Todos los métodos enumerados previamente tienen la misma complejidad temporal que es $\Theta(w * h * i * k*)$. Esta conclusión surge del hecho de que, en todos los casos, tenemos 4 ciclos anidados, en donde el primero se ejecuta w veces, el segundo h veces, el tercero i veces, y el cuarto k veces.

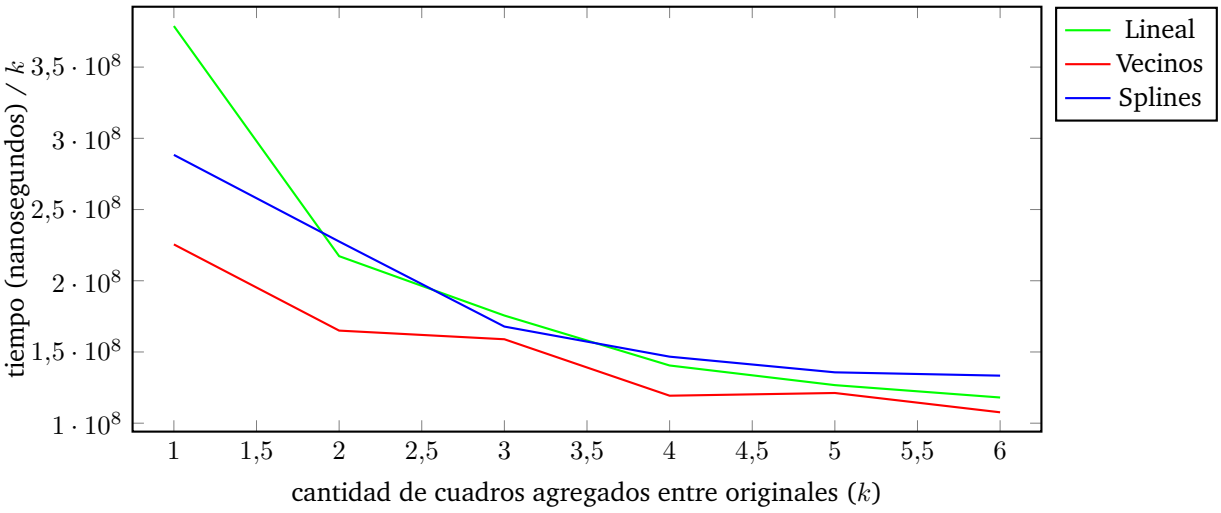
Para corroborar esta hipótesis planteamos un experimento con las siguientes características:

- Utilizamos el video provisto por la cátedra llamado *funnybaby*, el cual tiene 44 cuadros y es de 240x320.
- Solo medimos la resolución del sistema, no su creación o los respectivos pasajes de video a texto y viceversa.
- Dado que w, h y i son valores fijos que no podemos cambiar, variamos el k.
- Generamos instancias para valores de k entre 1 y 6.
- Para cada valor distinto de k generamos 4 instancias, cuyos valores vamos a promediar para mitigar los posibles valores distorsionados por algún procedimiento del procesador.

Los resultamos obtenidos fueron los siguientes:



Si tomamos los tiempos que arrojó la experimentación, y los dividimos por su respectivo k, obtenemos el siguiente resultado:



A partir de los gráficos, queda de manifiesto que los métodos tienen la misma complejidad temporal, ya que solo difieren en una constante.

4.4. Medición del ECM y PSNR de los métodos.

Sea F un frame del vídeo real (ideal) , y $\bar{F}$ el mismo frame del vídeo efectivamente construidos por alguno de los métodos. Sea m la cantidad de filas de píxeles en cada imagen y n la cantidad de columnas.

Definimos el Error Cuadrático Medio, `ECM`, como el real dado por:

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

A su vez definimos *Peak to Signal Noise Ratio*, `PSNR`, como el real dado por:

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

Ambas medidas nos sirven para realizar un análisis cuantitativo de la calidad de los resultados obtenidos con los distintos métodos.

En este experimento utilizamos los videos propuestos al inicio de la experimentación, variando la cantidad de cuadros que agregamos. Dichos videos fueron elegidos específicamente para variar la dificultad de Interpolación:

- El video **Amanecer** no contiene movimientos bruscos ni cambios de cámaras por lo que, para cualquier método, el error debería ser en lineas generales menor que pare el resto de los videos.
- El video **Skate** contiene movimientos bruscos pero no cambios de cámaras por lo que, se espera un mayor error que en el video anterior. Además, deberíamos ver un aumento del error en los frames donde hay movimientos bruscos.
- El video **Skate** contiene movimientos bruscos y cambios de cámaras por lo que, se espera un mayor error que en el resto de los videos. Además, deberíamos ver un aumento *importante* del error en los frames donde se produce el cambio de cámara.

A continuación, presentaremos e iremos analizando los resultados obtenidos:

4.4.1. ECM - Agregando 1 cuadro

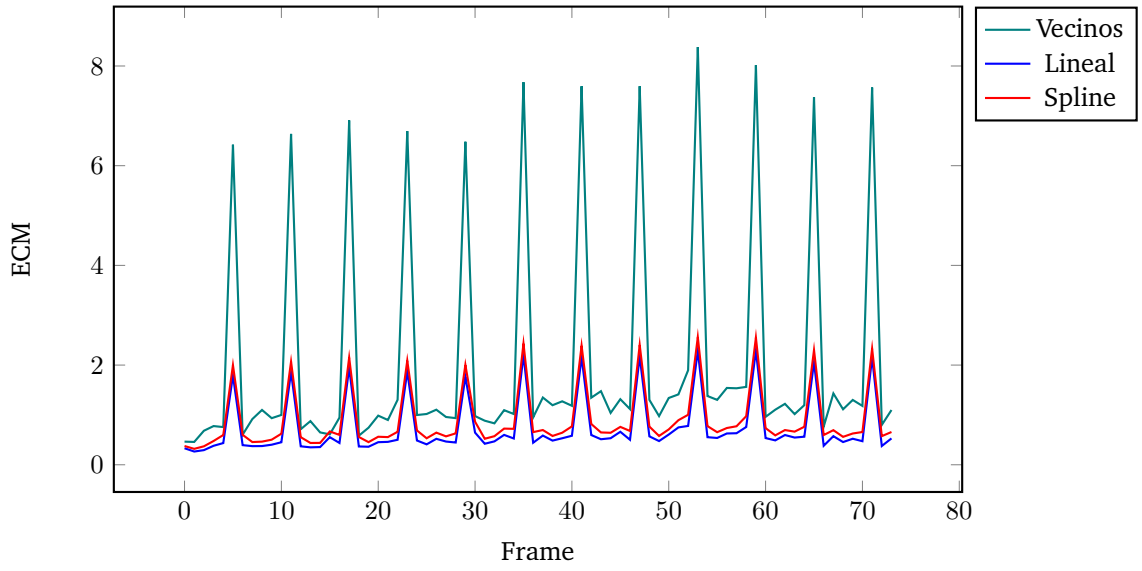


Figura 1: ECM para el video **Amanecer** al agregar 1 cuadro con distintos métodos de Interpolación.

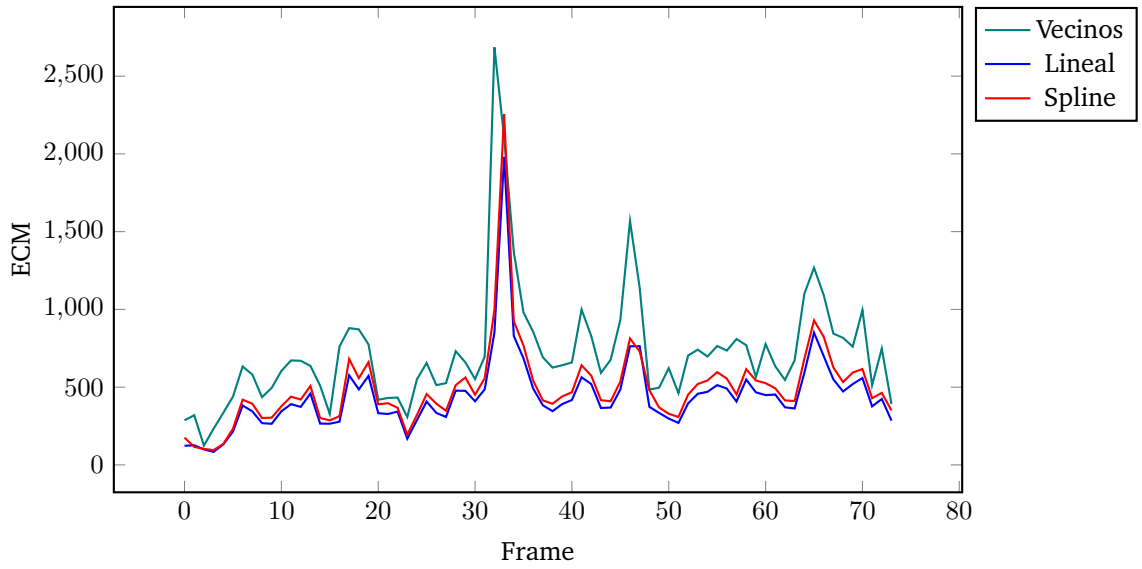


Figura 2: ECM para el video **Skate** al agregar 1 cuadro con distintos métodos de Interpolación.

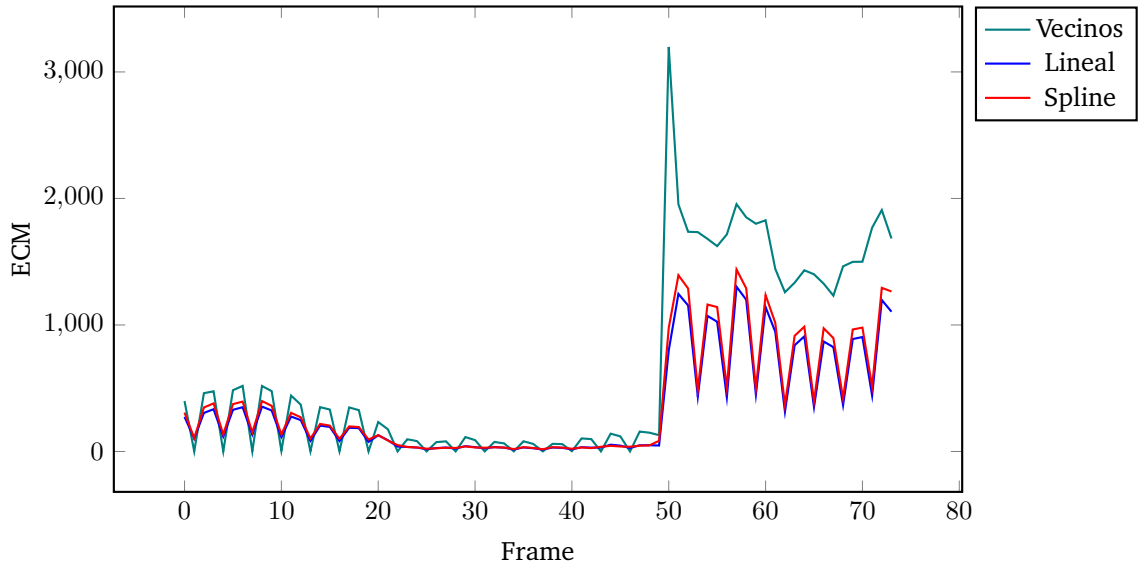


Figura 3: ECM para el video **Messi** al agregar 1 cuadro con distintos métodos de Interpolación.

Viendo las Figuras 1, 2 y 3 podemos decir que:

- La Interpolación por Vecinos obtiene consistentemente un mayor error que el resto de los métodos.
- La Interpolación Fragmentaria Lineal y la Interpolación por Splines obtienen en general un error similar, siendo la primera ligeramente mejor.
- El rango de error obtenido por todos los métodos en el video **Amanecer**, es significativamente menor que en el resto de los videos. Además, se puede ver que el error obtenido en ese video es en cierta forma “regular”, lo cual tiene sentido ya que tiene movimientos suaves y repetitivos.
- El rango de error obtenido por todos los métodos en los videos **Skate** y **Messi**, son similares, y ambos mayores al rango obtenido en el video **Amanecer**.
- El error obtenido en el video **Skate** es bastante irregular al compararlo con el video **Amanecer**, encontrándose un pico de error en el movimiento más brusco del video.
- A diferencia del caso anterior, el error obtenido en el video **Messi** es bastante regular, a excepción del pico de error que encontramos cuando ocurre el cambio de cámara, el cual es notorio para todos los métodos.

4.4.2. PSNR - Agregando 1 cuadro

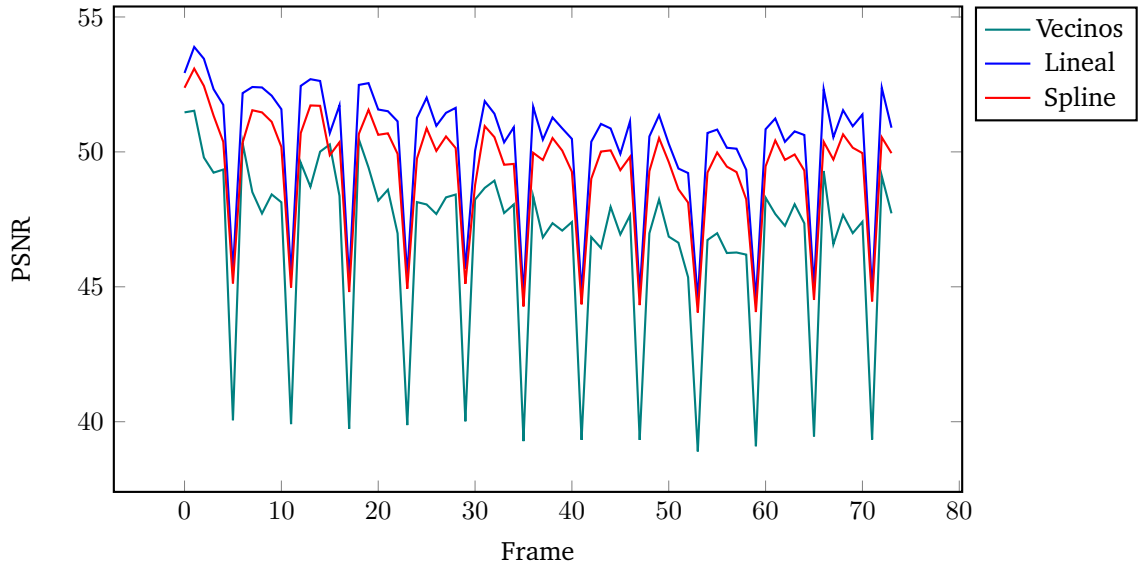


Figura 4: PSNR para el video **Amanecer** al agregar 1 cuadro con distintos métodos de Interpolación.

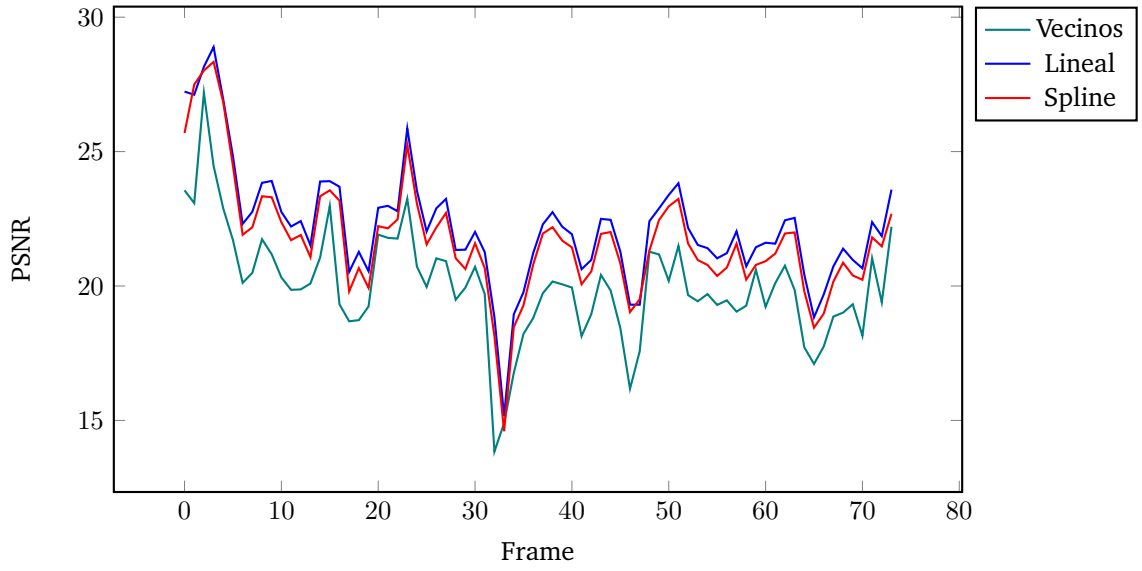


Figura 5: PSNR para el video **Skate** al agregar 1 cuadro con distintos métodos de Interpolación.

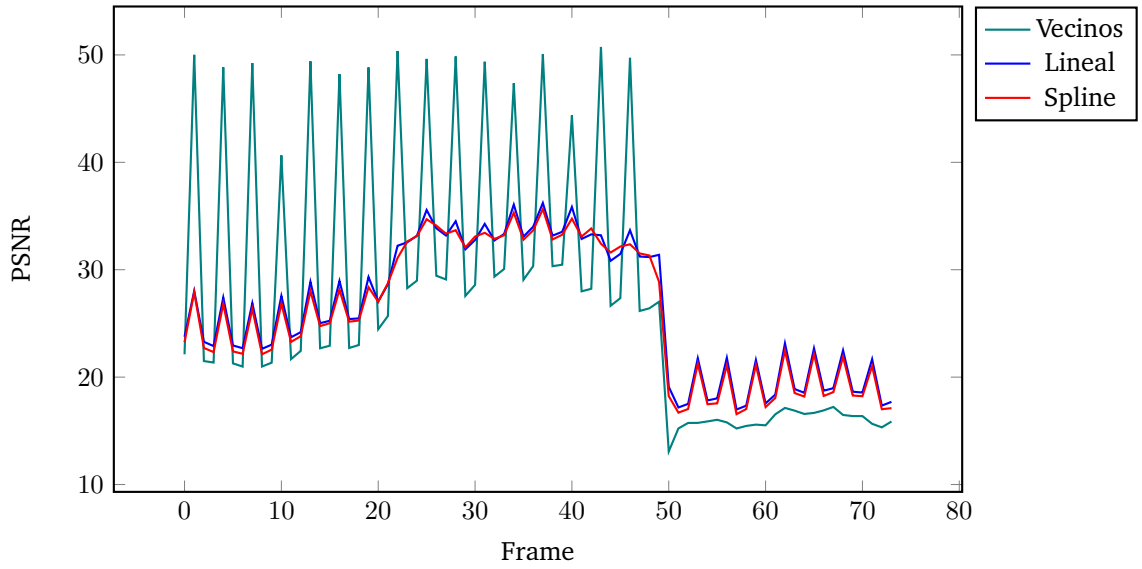


Figura 6: PSNR para el video **Messi** al agregar 1 cuadro con distintos métodos de Interpolación.

Sabiendo que, mientras más grande es el `ECM` más chico es el `PSNR`, encontramos que la información provista por este último no aporta nuevos elementos al análisis, ya que se condice con lo analizado previamente utilizando el `ECM`.

4.4.3. ECM - Agregando 5 cuadros

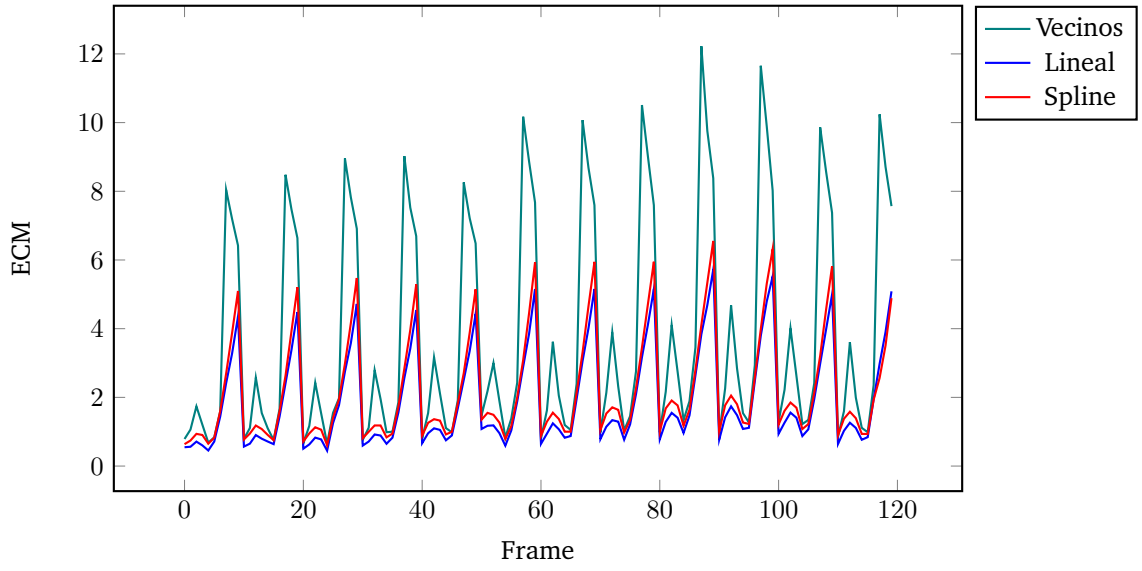


Figura 7: ECM para el video **Amanecer** al agregar 5 cuadros con distintos métodos de Interpolación.

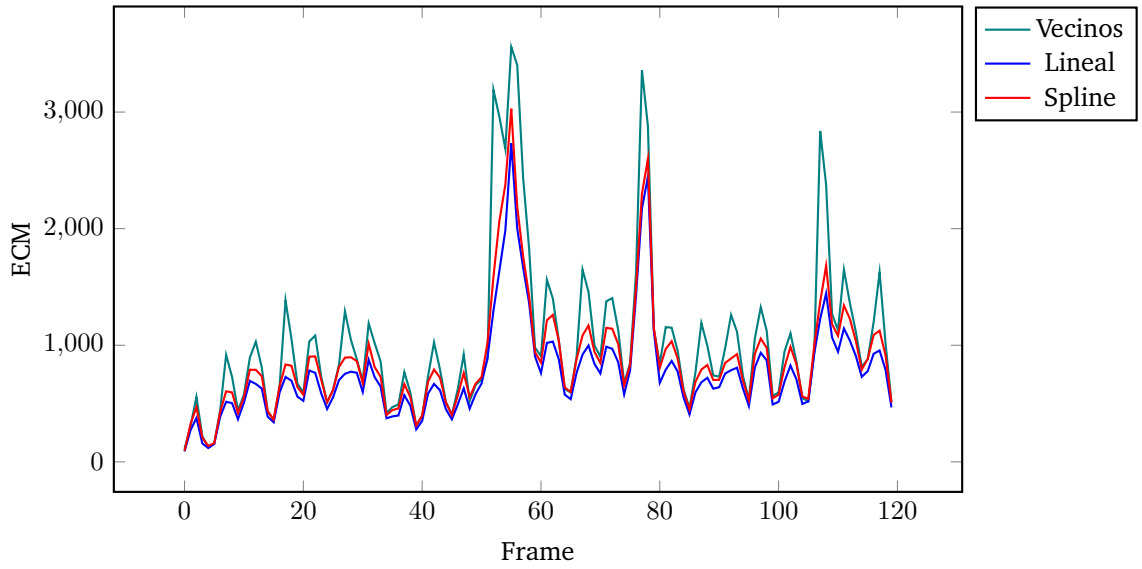


Figura 8: ECM para el video **Skate** al agregar 5 cuadros con distintos métodos de Interpolación.

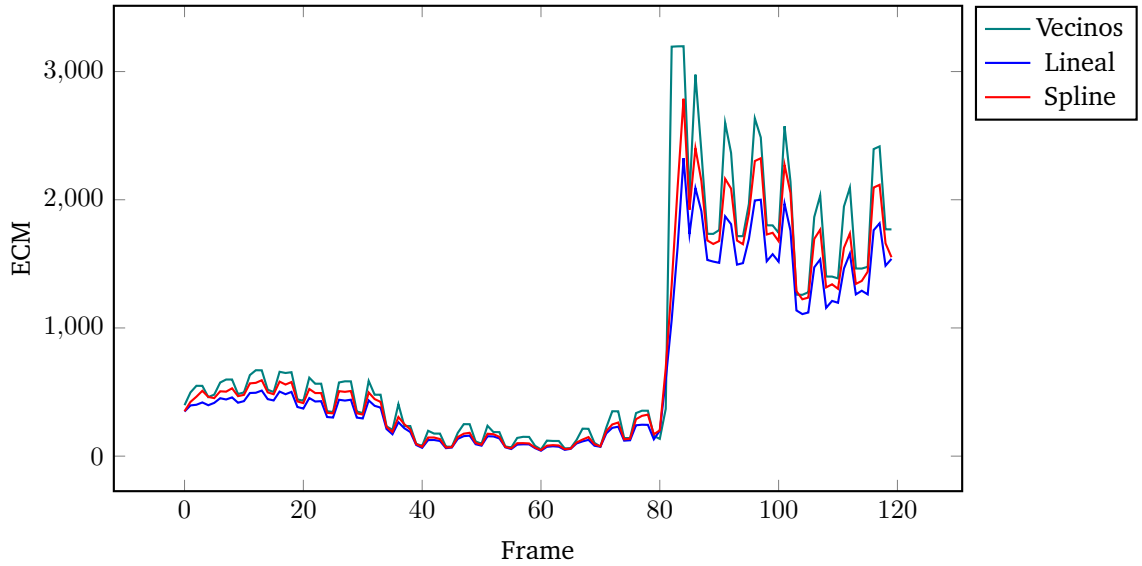


Figura 9: ECM para el video **Messi** al agregar 5 cuadros con distintos métodos de Interpolación.

Viendo las Figuras 7, 8 y 9 podemos decir que:

- El comportamiento general de los métodos para cada video no varió con respecto al escenario en el cual agregábamos 1 cuadro. Los análisis de movimiento brusco y cambios de cámaras se reafirman.
- En lineas generales, y en todos los videos, el error se incrementó con respecto al escenario en el cual agregábamos 1 cuadro, a excepción del método de Interpolación por Vecinos, que mantiene su rango de error.

4.4.4. PSNR - Agregando 5 cuadros

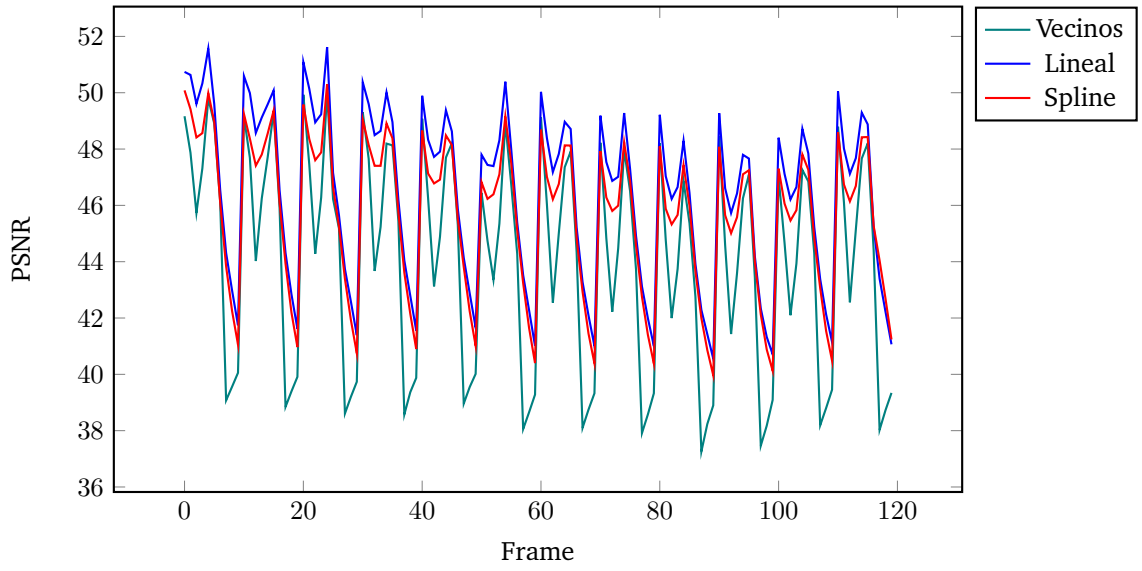


Figura 10: PSNR para el video **Amanecer** al agregar 5 cuadros con distintos métodos de Interpolación.

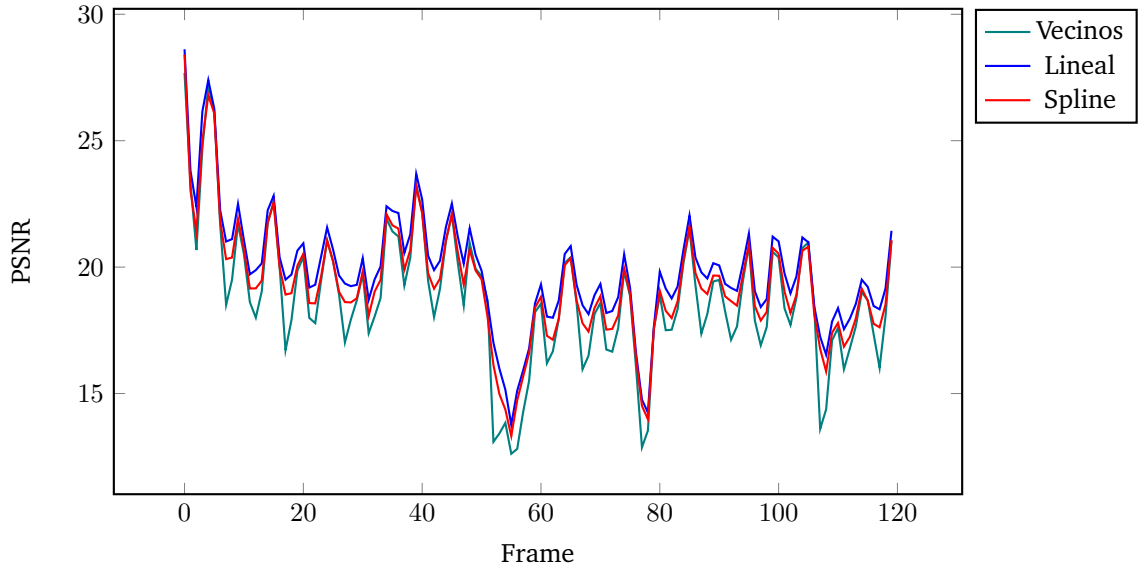


Figura 11: PSNR para el video **Skate** al agregar 5 cuadros con distintos métodos de Interpolación.

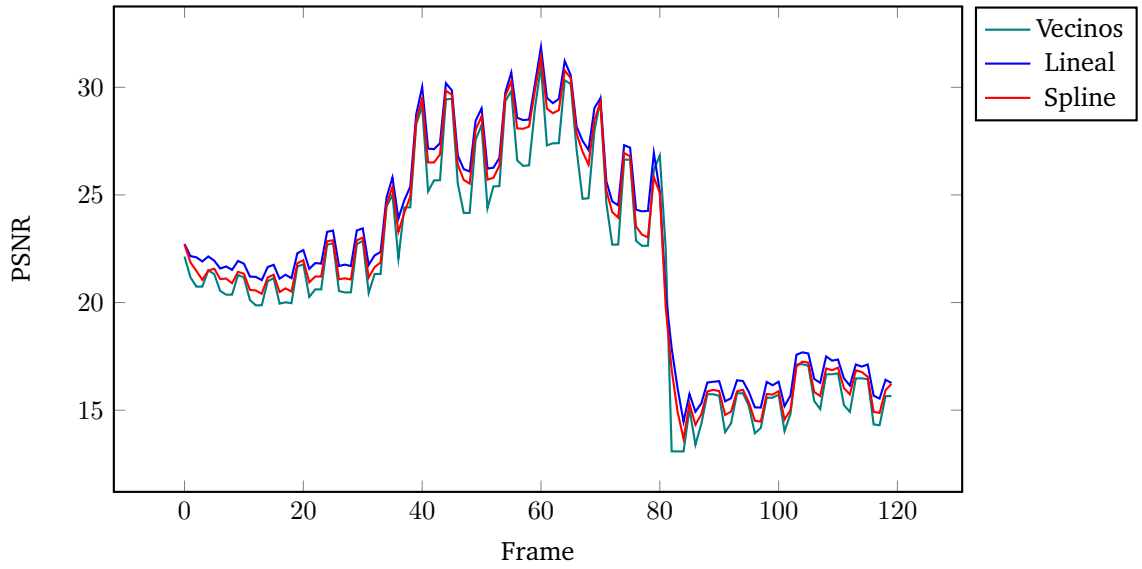


Figura 12: PSNR para el video **Messi** al agregar 5 cuadros con distintos métodos de Interpolación.

Sabiendo que, mientras más grande es el ECM más chico es el PSNR, encontramos que la información provista por este último no aporta nuevos elementos al análisis, ya que se condice con lo analizado previamente utilizando el ECM.

4.4.5. Conclusiones

En base a lo analizado, podemos concluir que:

- El error máximo cometido por el método de Interpolación por Vecinos, en comparación con los otros dos, lo hace inusable en la mayoría de los casos, y empeora a medida que agregamos más cuadros.
- Si bien en el Experimento de la Sección 4.2 pudimos lograr una mayor precisión con la Interpolación por Splines que con la Interpolación Fragmentaria Lineal, no sucedió lo mismo cuando aplicamos los métodos a la interpolación de videos. En este escenario, los métodos obtuvieron errores muy similares, siendo la Interpolación Fragmentaria Lineal mejor en algunos casos.

  Esto lo atribuimos a que los píxeles de de los videos no respetan ninguna familia de funciones, obviamente, y por lo tanto la función que infiere la Interpolación por Splines no es sustancialmente mejor que la obtenida mediante Interpolación Fragmentaria Lineal.

  Sin embargo, ya que las cotas de precisión definidas previamente no nos permitieron proyectar que la Interpolación por Splines iba a dar resultados similares a la Interpolación Fragmentaria Lineal, queda la duda de si la Interpolación por Splines de tamaño de bloque fijo puede dar mejores resultados. Para eso, haremos un experimento extra a continuación para comparar el error de dichos métodos.
- El video con movimientos suaves es mucho más sencillo de interpolar que los demás, y permite que incluso el método de Interpolación por Vecinos logre errores bajos. Al ir agregando movimientos bruscos y cambios de cámaras, los videos se vuelven más difíciles de interpolar, aumentando el error obtenido con los distintos métodos.

4.4.6. Interpolación por Splines: bloques de tamaño variable vs fijo

A continuación presentamos los resultados de comparar los errores de los siguientes métodos, solo para el video **Messi**:

- Interpolación por Splines *standard*, o de bloque variable.

- Interpolación por Splines (bloques de tamaño 2)
- Interpolación por Splines (bloques de tamaño 8)
- Interpolación por Splines (bloques de tamaño 32)

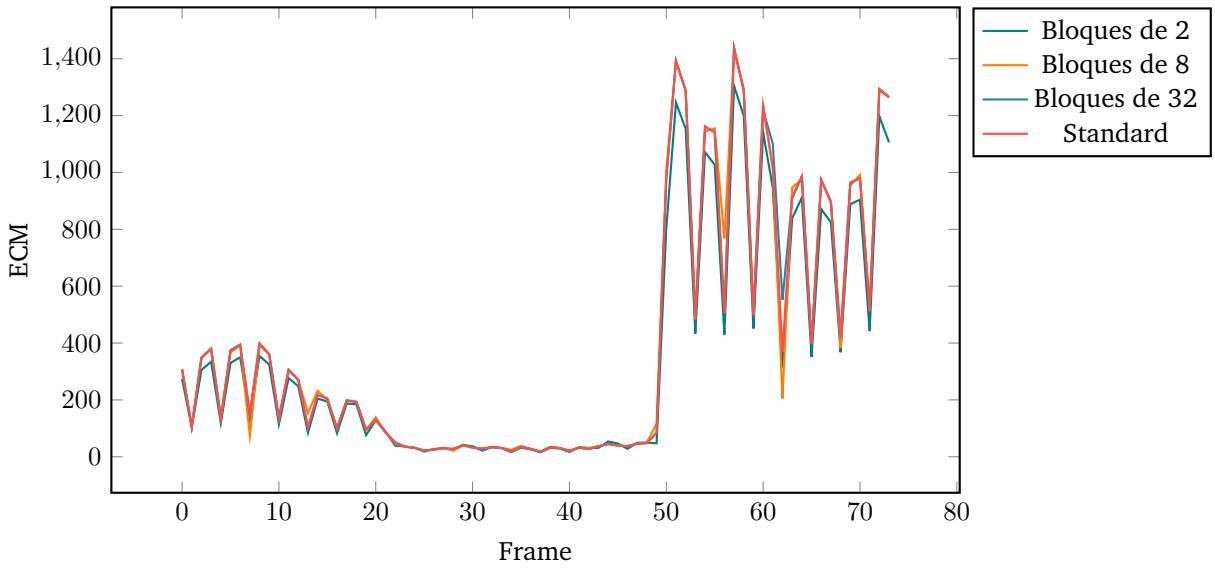


Figura 13: ECM para el video **Messi** al agregar 1 cuadro con variantes del método de Interpolación por Splines.

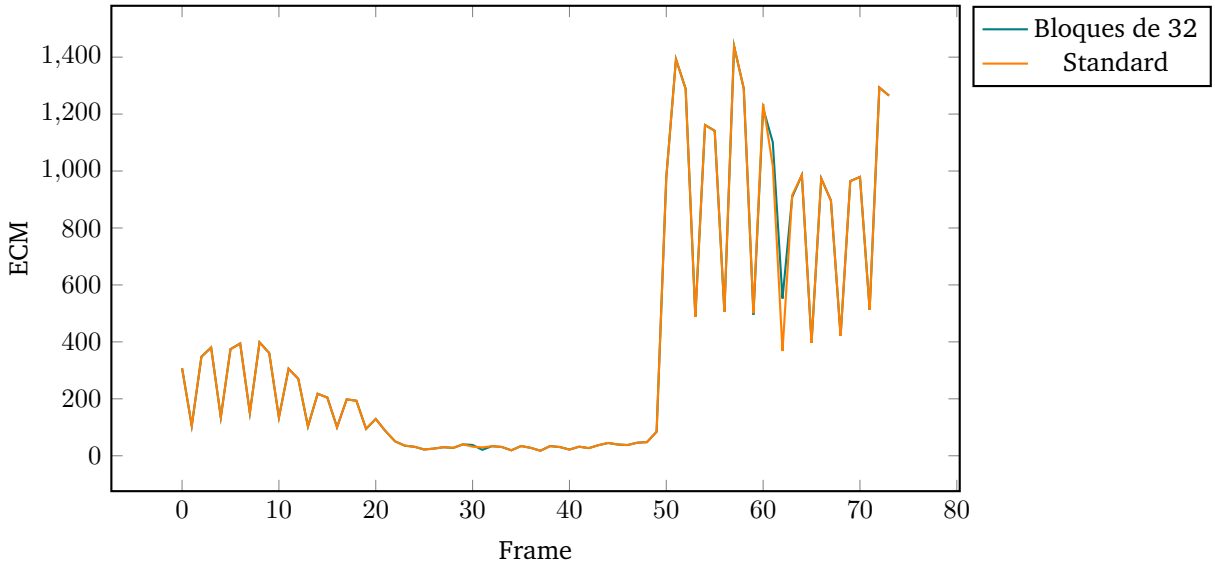


Figura 14: Igual a la Figura 13, pero mostrando solo Bloques 32 y Standard

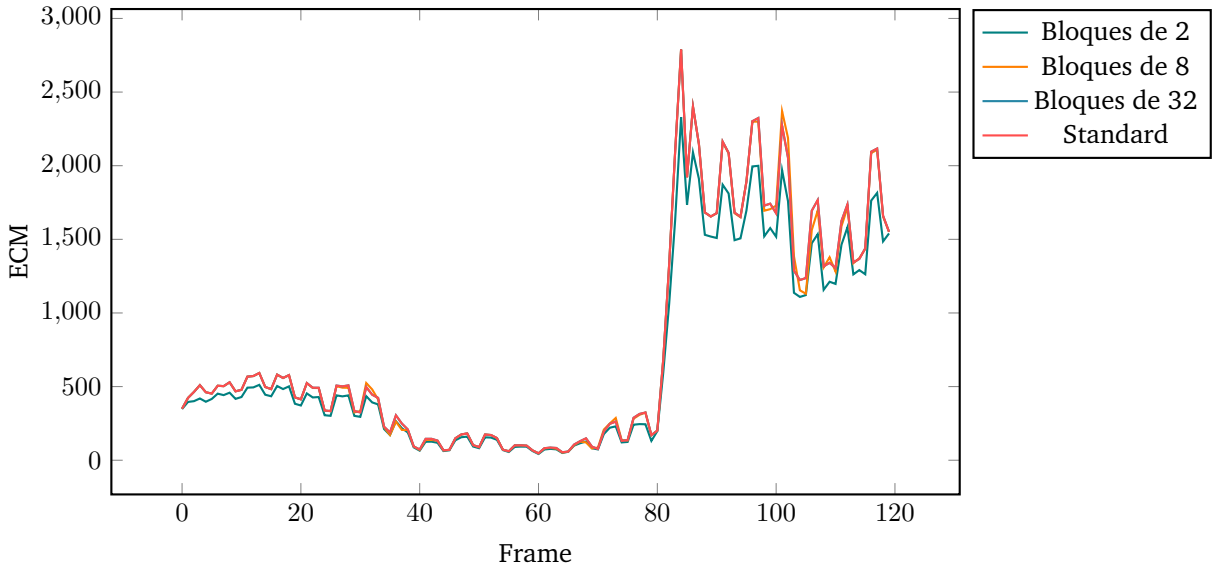


Figura 15: ECM para el video **Messi** al agregar 5 cuadros con variantes del método de Interpolación por Splines.

En base a las Figuras 13, 14 y15, podemos terminar de concluir que:

- El error del método de Interpolación por Splines usando bloques de tamaño fijo converge, al ir aumentando el tamaño del bloque, al error obtenido con la versión *standard*. Anteriormente ya habíamos confirmado un resultado parecido para diferentes familias de funciones, pero ahora podemos afirmar que esto vale también en el escenario en el cual estamos interpolando cuadros de video.
- Luego, la variante de Interpolación por Splines con tamaño fijo es similar (dependiendo del tamaño del bloque) a la variante *standard*, y por lo tanto similar a la Interpolación Fragmentaria Lineal.

4.5. Análisis cualitativos de los métodos, fenómeno de artifacts.

Los *artifacts* son errores visuales resultantes de la aplicación de los métodos. Estos errores visuales se caracterizan por romper la coherencia entre imágenes al generar distorsiones evidentes.

Para analizar los artifacts producidos por las diferentes implementaciones de Cámara Lenta, utilizamos los siguientes videos:

- **Video 1 - Skate**: 426x240, cantidad de cuadros originales: 151 , fps: 30, duración: 5s.
- **Video 2 - Messi**: 426x240, cantidad de cuadros originales: 151, fps: 30, duración: 5s.

A través de cada método de interpolación agregamos 10 cuadros entre cada par de frames de los videos. Así, nos propusimos analizar el funcionamiento de cada tipo de interpolación para verificar si hay movimientos bruscos, elementos falsos (por ejemplo cuando una persona se mueve y se duplica su pierna) u otros tipos de anomalías. Los resultados fueron los siguientes:

4.5.1. Interpolación por vecinos

- Como era de esperar no hubieron elementos falsos.
- Eso si, todos los movimientos parecieron bruscos, casi robóticos y sin ningún tipo de fluidez.
- Por ejemplo, en el video **Messi** la pelota exhibe un movimiento claramente antinatural.
- Todo esto es lógico dado que solo estamos copiando y pegando frames en el tiempo.
- Concluimos que este método no produce resultados de movimientos fluidos.

4.5.2. Interpolación lineal

- En este caso no encontramos los movimientos bruscos de interpolación por vecinos.
- Si podemos ver que hay congelamientos de imagen cada aproximadamente 1 segundo.
- Parecería haber un retroceso de calidad con respecto al método anterior.
- Detectamos imágenes falsas cuando las personas se mueven como:

4.5.3. Interpolación por Splines

- Con seguridad, este hasta ahora es el método que genera los videos más fluidos.
- Los movimientos de las cosas parecen naturales y no hay congelamientos ni transformaciones bruscas.
- Esto se lo atribuimos a que el método en sí aprovecha todos los datos del píxel a través del tiempo para producir resultados mejores que tienen que ver con el marco teórico que le dan los polinomio interpoladores de Lagrange.
- Eso si, siguen habiendo imágenes falsas a causa del movimiento y no son menores pero si ligeramente mejores que en la interpolación lineal.
- Por ejemplo, en la instancia **Skate**, los resultados son del estilo:

4.6. Comparación

Como pudimos ver, cada método introduce diferentes artifacts. En el caso de la Interpolación lineal, llama la atención que el resultado parece peor con respecto al de Vecinos. Esto nos indica que, si bien el error matemático de un método puede ser peor, las irregularidades visuales que introduce Interpolación lineal son peores para una persona usuario desde su punto de vista subjetivo. Podemos concluir que, debido a las particularidades que introduce cada método, parecería que la interpolación mediante Splines genera los mejores resultados.

5. Conclusión

En este trabajo pudimos no solo modelar el problema planteado, sino apreciar y aprovechar las propiedades del mismo para así resolverlo con los métodos estudiados observando también las características de ellos.

Así mismo, cabe destacar que al realizar operaciones con aritmética finita, tanto para la solución de los sistemas como para los valores evaluados en cada interpolador, no podemos garantizar que los resultados obtenidos sean exactos, pero dado que realizamos varias instancias de prueba con distintas metodologías, pudimos ver que los valores que obtuvimos eran coherentes a su contexto.

Todos los métodos de interpolación realizados en este trabajo cumplieron con la tarea de generar en mayor o menor medida el efecto de cámara lenta que buscábamos, aunque cada uno con sus características particulares. Mediante la experimentación pusimos bajo la lupa cada interpolador utilizado para así poder compararlos entre sí.

Comenzamos nuestros experimentos probando el correcto funcionamiento de los métodos implementados. Aquí además de ver mediante las cotas de precisión que todos los métodos cumplían la tarea de interpolar las funciones predeterminadas pudimos tener una primera visión sobre cómo se comportaba cada interpolador. El más básico, por vecinos, efectivamente demostró tener la mayor cota de error distanciándose ampliamente del resto, mientras que la interpolación lineal obtuvo resultados similares a los obtenidos con splines. En este experimento en particular, la interpolación mediante splines probó ser la de menor error, superando a la lineal para funciones cuadráticas y cúbicas, donde a su vez se pudo observar su relación con la interpolación por splines de a bloques, que con bloques más grandes su cota se aproximaba a la del spline standard.

En la siguiente prueba, donde lo que se analizó fue el error producido al eliminar cuadros del video original e intentar recrearlos con nuestros interpoladores, los resultados fueron similares al experimento anterior. Con ambas fórmulas de cálculo de error (ECM y PSNR) se pudo ver cómo vecinos era el método que peor se comportaba mientras que lineal y splines estuvieron emparejados a lo largo de todas las instancias de prueba realizadas.

Finalmente, la búsqueda de artifacts en el video producido, de anomalías y calidad final a nivel visual para el usuario resultó ser el punto determinante para decidir cuál de los sistemas propuestos mejor se desempeña en la tarea de proveer un efecto de cámara lenta. Los resultados acá dieron un giro completo con respecto a las conclusiones que veníamos realizando respecto a las métricas estudiadas. Para empezar, la interpolación de a vecinos dio mejores resultados que la lineal en términos de cantidad de artifacts y calidad general del video producido. El método por splines fue el que mejor llegó a producir la sensación de cámara lenta, con algunas anomalías pero no lo suficientemente significantes como las vistas en la interpolación lineal. Esto creemos que se podría explicar con el hecho de que las métricas seleccionadas para decidir qué interpolador era mejor estaban demasiado atadas a que interpolaran correctamente sin contemplar cómo resultaba visualmente esta interpolación.

Por último, podemos mencionar como posibles desarrollos a futuro la búsqueda de métricas que se centren más en el aspecto visual de los interpoladores, distintas formas de interpolación o directamente métodos completamente distintos, como lo sería considerar la aplicación de cuadrados mínimos, para seguir comparando y buscando que se generen los menores artifacts posibles.

6. Referencias

[1] Richard L. Burden and J. Douglas Faires. *Numerical Analysis*, chapter 3, pages 144–163. Richard Stratton, 9th edition, 2011.

7. Enunciado

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:

$$\texttt{ECM}(F, \bar{F}) = \frac{1}{mn} \sum_{i=1}^{m} \sum_{j=1}^{n} |F_{k_{ij}} - \bar{F}_{k_{ij}}|^2 \tag{3}$$

y

$$\texttt{PSNR}(F, \bar{F}) = 10 \log_{10} \left(\frac{255^2}{\texttt{ECM}(F, \bar{F})} \right). \tag{4}$$

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.

A. Código C++

video.cpp

```
#include <video.h>

Video::Video(){
  this->numero_frames = 0;
  this->numero_frames_out = 0;
  this->ancho = 0;
  this->alto = 0;
  this->fps = 0;
  this->cuadros_nuevos = 0;
}

Video::Video(const char* entrada, int cuadrosNuevos){
  ifstream archivo_entrada;
  archivo_entrada.open(entrada);

  if(archivo_entrada.good()){
    char sep;
    archivo_entrada >> this->numero_frames;
    archivo_entrada >> this->alto;
    archivo_entrada >> sep;
    archivo_entrada >> this->ancho;
    archivo_entrada >> this->fps;
    this->cuadros_nuevos = cuadrosNuevos;
    this->numero_frames_out = this->numero_frames + (this->numero_frames
        - 1)*this->cuadros_nuevos;
    this->frames = vector<vector<vector<double> > >(this->ancho, vector<
       vector<double> >(this->alto, vector<double>(this->numero_frames,
       0)));
    this->frames_out = vector<vector<vector<double> > >(this->ancho,
       vector<vector<double> >(this->alto, vector<double>(this->
       numero_frames_out, 0)));

    for(int i = 0; i < this->numero_frames_out ; i += this->
       cuadros_nuevos + 1){
      for(int y = 0; y < this->alto; y++){
        for(int x = 0; x < this->ancho - 1; x++){
          archivo_entrada >> this->frames_out[x][y][i];
          this->frames[x][y][i/(this->cuadros_nuevos + 1)] = this->
             frames_out[x][y][i];
          archivo_entrada >> sep;
        }
        archivo_entrada >> this->frames_out[this->ancho - 1][y][i];
        this->frames[this->ancho - 1][y][i/(this->cuadros_nuevos + 1)] =
            this->frames_out[this->ancho - 1][y][i];
      }
    }
    archivo_entrada.close();
  }
  else{
```

```
    cout << "Error al leer el archivo " << entrada << endl;
  }
}

vector<vector<vector<double> > > Video::obtenerFramesOriginales() {
  return frames;
}
vector<vector<vector<double> > > Video::obtenerFramesCalculados() {
  return frames_out;
}

void Video::guardar(const char* salida){
  ofstream archivo_salida;
  archivo_salida.open(salida);

  archivo_salida << this->numero_frames_out << endl;
  archivo_salida << this->alto << "," << this->ancho << endl;
  archivo_salida << this->fps << endl;

  for(int i = 0; i < this->numero_frames_out; i++){
    for(int y = 0; y < this->alto; y++){
      for(int x = 0; x < this->ancho - 1; x++){
        archivo_salida << lround(this->frames_out[x][y][i]) << ",";
      }
      archivo_salida << lround(this->frames_out[this->ancho - 1][y][i])
          << endl;
    }
  }
  archivo_salida.close();
}

void Video::cambiarTamanioBloques(int tamanio_bloques) {
  this->tamanio_bloques = tamanio_bloques;
}

void Video::aplicarCamaraLenta(MetodoInterpolacion metodo){
  switch(metodo){
    case VECINOS:
    cout << "Aplicando interpolación por vecinos" << endl;
      interpolarVecinos();
      break;
    case LINEAL:
      cout << "Aplicando interpolación lineal" << endl;
      interpolarLineal();
      break;
    case SPLINES:
      cout << "Aplicando interpolación cúbica" << endl;
      interpolarSplines();
      break;
    case MULTI_SPLINES:
      cout << "Aplicando interpolación cúbica de a tramos. Tamaño
          bloques: " << tamanio_bloques << endl;
      interpolarMultiSplines(tamanio_bloques -1);
      break;
```

```
  }
}

void Video::interpolarSplines(){
  Spline spline_xy (this->numero_frames);

  double spline_x;
  int spline_y;
  for(int x = 0; x < this->ancho; x++){
    for(int y = 0; y < this->alto; y++){
      spline_xy.recalcular(this->frames[x][y]);
      for(int i = 1; i < this->numero_frames_out; i += this->
          cuadros_nuevos + 1){
        for(int n = 0; n < this->cuadros_nuevos; n++){
          spline_x = (i - 1)/(this->cuadros_nuevos + 1) + float(n + 1)/(
              this->cuadros_nuevos + 1);
          spline_y = round(spline_xy.evaluar(spline_x));
          if(spline_y < 0){
            spline_y = 0;
          }
          else if(spline_y > 255){
            spline_y = 255;
          }
          this->frames_out[x][y][i + n] = spline_y;
        }
      }
    }
  }
}

void Video::interpolarMultiSplines(int longitud_tramo){
  MultiSpline multi_spline_xy (this->numero_frames, longitud_tramo);

  double multi_spline_x;
  int multi_spline_y;
  for(int x = 0; x < this->ancho; x++){
    for(int y = 0; y < this->alto; y++){
      multi_spline_xy.recalcular(this->frames[x][y]);
      for(int i = 1; i < this->numero_frames_out; i += this->
          cuadros_nuevos + 1){
        for(int n = 0; n < this->cuadros_nuevos; n++){
          multi_spline_x = (i - 1)/(this->cuadros_nuevos + 1) + float(n
              + 1)/(this->cuadros_nuevos + 1);
          multi_spline_y = round(multi_spline_xy.evaluar(multi_spline_x)
              );
          if(multi_spline_y < 0){
            multi_spline_y = 0;
          }
          else if(multi_spline_y > 255){
            multi_spline_y = 255;
          }
          this->frames_out[x][y][i + n] = multi_spline_y;
        }
```

```
      }
    }
  }
}

void Video::interpolarLineal(){
  for(int x = 0; x < this->ancho; x++){
    for(int y = 0; y < this->alto; y++){
      InterpolacionLineal lineal(this->frames[x][y], this->
          cuadros_nuevos);
      int count = 0;
      for (int i = 0; i < this->numero_frames - 1; i++) {
        double aux = double(1)/double(this->cuadros_nuevos + 1);
        for (int k = 0; k < this->cuadros_nuevos + 1; k++) {
          double pixel = lineal.evaluar(double(i) + double(k)*aux);
          if (pixel < 0) {
            pixel = 0;
          } else if (pixel > 255) {
            pixel = 255;
          }
          this->frames_out[x][y][count] = pixel;
          count++;
        }
      }
    }
  }
}

void Video::interpolarVecinos() {
  for(int x = 0; x < this->ancho; x++){
    for(int y = 0; y < this->alto; y++){
      InterpolacionVecinos vecinos(this->frames[x][y], this->
          cuadros_nuevos);
      int count = 0;
      for (int i = 0; i < this->numero_frames - 1; i++) {
        double aux = double(1)/double(this->cuadros_nuevos + 1);
        for (int k = 0; k < this->cuadros_nuevos + 1; k++) {
          this->frames_out[x][y][count] = vecinos.evaluar(double(i) +
              double(k)*aux);
          count++;
        }
      }
    }
  }
}
```

interpolador.h

```
#ifndef INTERPOLADOR_H_
#define INTERPOLADOR_H_

class Interpolador{
  public:
    virtual double evaluar(double x) =0;
```

```
};

#endif // INTERPOLADOR_H_INCLUDED
```

spline.cpp

```
#include <spline.h>

Spline::Spline(){
}

Spline::Spline(int n){
  generarSistema(n);
}

Spline::Spline(const vector<double> &y){
  generarSistema(y.size());
  generarSpline(y);
}

void Spline::generarSistema(int n){
  vector<vector<double> > sistema(n, vector<double>(2, 0));

  // Primera y última fila
  sistema[0] = {1, 0};
  sistema[n-1] = {0, 1};

  for(int i = 1; i < n-1; i++){
    sistema[i] = {1, 4};
  }

  // Factorización LU
  for(int i = 1; i < n - 1; i++){
    double coef = sistema[i + 1][0]/sistema[i][1];
    sistema[i + 1][0] = coef; // Guardo coef de mi L
    sistema[i + 1][1] -= coef;
  }

  this->factorizacionLU = sistema;
}

void Spline::generarSpline(const vector<double> &y){
  int n = y.size();
  this->a = y;
  // Primero necesito calcular mis coeficientes "c"

  // Resuelvo triangular inferior (Lx = b)
  // b = 3*(a[i + 1] - 2*a[i] + a[i - 1])
  vector<double> x(n, 0);
  for(int i = 1 ; i < n - 1; i++){
    x[i] = 3*(this->a[i + 1] - 2*this->a[i] + this->a[i - 1]);
    x[i] -= x[i - 1]*this->factorizacionLU[i][0];
  }
```

```
  // Resuelvo triangular superior (Uc = x)
  this->c = vector<double>(n, 0);
  for(int i = n - 2; i > 0; i--){
    this->c[i] = x[i];
    this->c[i] -= this->c[i + 1];
    this->c[i] /= factorizacionLU[i][1];
  }

  // Calculo mis coeficientes "b" y "d"
  this->b = vector<double>(n, 0);
  this->d = vector<double>(n, 0);
  for(int i = 0; i < n - 1; i++){
    this->b[i] = this->a[i + 1] - this->a[i] - (2*this->c[i] + this->c[i
        + 1])/3;
    this->d[i] = (c[i + 1] - c[i])/3;
  }
}

void Spline::recalcular(const vector<double> &y){
  generarSpline(y);
}

double Spline::evaluar(double x){
  int xj = (int)floor(x);
  return this->a[xj] + this->b[xj]*(x - xj) + this->c[xj]*(x - xj)*(x -
     xj) + this->d[xj]*(x - xj)*(x - xj)*(x - xj);
}
```

lineal.h

```
#include <lineal.h>

InterpolacionLineal::InterpolacionLineal(){
}

InterpolacionLineal::InterpolacionLineal(const vector<double> &y, int
    valores_a_agregar){
  recalcular(y, valores_a_agregar);
}

void InterpolacionLineal::recalcular(const vector<double> &y, int
    valores_a_agregar){
  this->cant_datos_originales = y.size();
  this->valores_a_agregar = valores_a_agregar;

  datos_generados.clear();

  for(unsigned int i = 0; i < y.size() - 1; i++){
    datos_generados.push_back(y[i]);

    double dif_dividida_cero = y[i];
    double dif_dividida_uno = (y[i+1] - y[i]) / double(valores_a_agregar
       +1);
```

```
    for(int k = 0; k < valores_a_agregar; k++){
      double pixel = dif_dividida_cero + dif_dividida_uno*double(k + 1);
      datos_generados.push_back(pixel);
    }
  }
  datos_generados.push_back(y[y.size()-1]);
}

double InterpolacionLineal::evaluar(double x){
  double aux = x * double(valores_a_agregar+1);
  int xj = (int)floor(aux);
  //cout << x << " -> " << xj << endl;
  return datos_generados[xj];
}
```

vecinos.cpp

```
#include <vecinos.h>

InterpolacionVecinos::InterpolacionVecinos(){
}

InterpolacionVecinos::InterpolacionVecinos(const vector<double> &y, int
    valores_a_agregar){
  recalcular(y, valores_a_agregar);
}

void InterpolacionVecinos::recalcular(const vector<double> &y, int
    valores_a_agregar){
  this->cant_datos_originales = y.size();
  this->valores_a_agregar = valores_a_agregar;

  datos_generados.clear();
  for(unsigned int i = 0; i < y.size() - 1; i++){
    datos_generados.push_back(y[i]);
    for(int k = 0; k < valores_a_agregar/2; k++){
      datos_generados.push_back(y[i]);
    }
    for(int k = valores_a_agregar/2; k < valores_a_agregar; k++){
      datos_generados.push_back(y[i + 1]);
    }
  }
  datos_generados.push_back(y[y.size()-1]);
}

double InterpolacionVecinos::evaluar(double x){
  double aux = x * double(valores_a_agregar+1);
  int xj = (int)floor(aux);
  // cout << x << " -> " << xj << endl;
  return datos_generados[xj];
}
```

multi_spline.cpp

```
#include <multi_spline.h>

MultiSpline::MultiSpline(){
}

MultiSpline::MultiSpline(int n, int tramo){
  generarSistema(n, tramo);
}

MultiSpline::MultiSpline(const vector<double>&y, int tramo){
  generarSistema(y.size(), tramo);
  generarMultiSpline(y);
}

void MultiSpline::recalcular(const vector<double> &y){
  generarMultiSpline(y);
}

void MultiSpline::generarSistema(int n, int tramo){
  this->numero_tramos = (n - 1)/tramo;
  this->longitud_tramo = tramo;
  this->longitud_tramo_menor = (n - 1)%tramo;
  if(this->longitud_tramo_menor != 0){
    this->numero_tramos++;
    this->tramos = vector<Spline> (this->numero_tramos, Spline(this->
        longitud_tramo + 1));
    this->tramos[this->numero_tramos - 1] = Spline(this->
        longitud_tramo_menor + 1);
  }
  else{
    this->tramos = vector<Spline> (this->numero_tramos, Spline(this->
        longitud_tramo + 1));
  }
}

void MultiSpline::generarMultiSpline(const vector<double> &y){
  vector<double>::const_iterator it = y.begin();
  for(int i = 0; i < this->numero_tramos - 1; i++){
    this->tramos[i].recalcular(vector<double>(it, it + this->
        longitud_tramo + 1));
    it += this->longitud_tramo;
  }

  if(this->longitud_tramo_menor != 0){
    this->tramos[this->numero_tramos - 1].recalcular(vector<double>(it,
        it + this->longitud_tramo_menor + 1));
  }
  else{
    this->tramos[this->numero_tramos - 1].recalcular(vector<double>(it,
        it + this->longitud_tramo + 1));
  }
}

double MultiSpline::evaluar(double x){
```

```
  int xj = (int)floor(x);
  double result = 0;

  if(this->longitud_tramo_menor != 0 && xj >= (this->numero_tramos - 1)*
     this->longitud_tramo){
    xj = (this->numero_tramos - 1)*this->longitud_tramo;
    result = this->tramos[this->numero_tramos - 1].evaluar(x - xj);
  }
  else{
    if(xj == this->numero_tramos*this->longitud_tramo){
      xj = this->longitud_tramo;
      result = this->tramos[this->numero_tramos - 1].evaluar(xj);
    }
    else{
      xj /= this->longitud_tramo;
      result = this->tramos[xj].evaluar(x - xj*this->longitud_tramo);
    }
  }
  return result;
}
```

test_utils.h

```
#ifndef TEST_UTILS_H_
#define TEST_UTILS_H_

#define DELTA 0.0001
#define DEBUG false

#include "mini_test.h"

#include <string>
#include <sstream>
#include <iostream>
#include <math.h>
#include <ctime>
#include <stdlib.h>
#include <cstdio>
#include <chrono>
#include <random>
#include <iomanip>

using namespace std;

namespace utils {

  static void assert_precision(double valor, double esperado, double
   precision){
    ASSERT(abs(valor - esperado) < precision);
  }

  static void assert_precision(double valor, double esperado){
    assert_precision(valor, esperado, DELTA);
  }
```

```
static void assert_interpolacion(Interpolador* interpolador, vector<
 double> esperados, double intervalo, double precision){
  double xj = 0;
  for(unsigned int i = 0; i < esperados.size(); i++){
    if(DEBUG){
      cout << "P(" << xj << ")␣=␣" << setprecision(15) << interpolador
 ->evaluar(xj) << ",␣Expected(" << xj <<")␣=␣" << esperados[i] << endl
 ;
    }
    // Si el intervalo equivale a un punto interpolado, el
 interpolador TIENE que
    // darme el "mismo" valor que la función original
    if(abs(xj - i) < DELTA){
      assert_precision(interpolador->evaluar(xj), esperados[i]);
    } else{
      assert_precision(interpolador->evaluar(xj), esperados[i],
 precision);
    }
    xj += intervalo;
  }
}

static void assert_interpolacion(Interpolador* interpolador, vector<
 double> esperados, double intervalo){
  assert_interpolacion(interpolador, esperados, intervalo, DELTA);
}

static double frame_error_cuadratico_medio(const vector<vector<double>
  > &output, const vector<vector<double> > &real) {
  ASSERT(output.size() != 0 && output[0].size() != 0);
  ASSERT(real.size() != 0 && real[0].size() != 0);
  ASSERT(output.size() == real.size() && output[0].size() == real[0].
 size());

  int ancho = output.size();
  int alto = output[0].size();
  unsigned long sum = 0;
  for(int x = 0; x < ancho; x++){
    for(int y = 0; y < alto; y++){
      sum += pow(real[x][y] - output[x][y], 2);
    }
  }
  return double(sum)/(double(ancho)*double(alto));
}

static double frame_peak_to_signal_noise_ratio(const vector<vector<
 double> > &output, const vector<vector<double> > &real) {
  ASSERT(output.size() != 0 && output[0].size() != 0);
  ASSERT(real.size() != 0 && real[0].size() != 0);
  ASSERT(output.size() == real.size() && output[0].size() == real[0].
 size());

  double aux = double(pow(255,2));
```

```
    double ecm = frame_error_cuadratico_medio(output, real);
    if (ecm < DELTA) {
      // el frame output es igual al frame real
      return 0;
    } else {
      return 10*log10(aux/ecm);
    }
  }

  static double video_max_error_cuadratico_medio(const vector<vector<
   vector<double> > > &output, const vector<vector<vector<double> > > &
   real) {
    ASSERT(output.size() != 0 && output[0].size() != 0 && output[0][0].
   size() != 0);
    ASSERT(real.size() != 0 && real[0].size() != 0 && real[0][0].size()
   != 0);
    ASSERT(output.size() == real.size() && output[0].size() == real[0].
   size() && output[0][0].size() <= real[0][0].size());

    int ancho = output.size();
    int alto = output[0].size();
    int frames = output[0][0].size();
    double max_error = 0;

    vector<vector<double> > frame_output(ancho, vector<double>(alto, 0))
   ;
    vector<vector<double> > frame_real(ancho, vector<double>(alto, 0));
    for (int k = 0; k < frames; k++) {
      for(int x = 0; x < ancho; x++){
        for(int y = 0; y < alto; y++){
          frame_output[x][y] = output[x][y][k];
          frame_real[x][y] = real[x][y][k];
        }
      }
      double error = frame_error_cuadratico_medio(frame_output,
   frame_real);
      if (error > max_error) {
        max_error = error;
      }
    }

    return max_error;
  }

  static double video_max_peak_to_signal_noise_ratio(const vector<vector
   <vector<double> > > &output, const vector<vector<vector<double> > > &
   real) {
    ASSERT(output.size() != 0 && output[0].size() != 0 && output[0][0].
   size() != 0);
    ASSERT(real.size() != 0 && real[0].size() != 0 && real[0][0].size()
   != 0);
    ASSERT(output.size() == real.size() && output[0].size() == real[0].
   size() && output[0][0].size() <= real[0][0].size());
```

```
    int ancho = output.size();
    int alto = output[0].size();
    int frames = output[0][0].size();
    double max_error = 0;

    vector<vector<double> > frame_output(ancho, vector<double>(alto, 0))
   ;
    vector<vector<double> > frame_real(ancho, vector<double>(alto, 0));
    for (int k = 0; k < frames; k++) {
      for(int x = 0; x < ancho; x++){
        for(int y = 0; y < alto; y++){
          frame_output[x][y] = output[x][y][k];
          frame_real[x][y] = real[x][y][k];
        }
      }
      double error = frame_peak_to_signal_noise_ratio(frame_output,
   frame_real);
      if (error > max_error) {
        max_error = error;
      }
    }

    return max_error;
  }

  static double video_prom_error_cuadratico_medio(const vector<vector<
   vector<double> > > &output, const vector<vector<vector<double> > > &
   real) {
    ASSERT(output.size() != 0 && output[0].size() != 0 && output[0][0].
   size() != 0);
    ASSERT(real.size() != 0 && real[0].size() != 0 && real[0][0].size()
   != 0);
    ASSERT(output.size() == real.size() && output[0].size() == real[0].
   size() && output[0][0].size() <= real[0][0].size());

    int ancho = output.size();
    int alto = output[0].size();
    int frames = output[0][0].size();
    int frames_iguales = 0;
    double sum_error = 0;

    vector<vector<double> > frame_output(ancho, vector<double>(alto, 0))
   ;
    vector<vector<double> > frame_real(ancho, vector<double>(alto, 0));
    for (int k = 0; k < frames; k++) {
      for(int x = 0; x < ancho; x++){
        for(int y = 0; y < alto; y++){
          frame_output[x][y] = output[x][y][k];
          frame_real[x][y] = real[x][y][k];
        }
      }
      double error = frame_error_cuadratico_medio(frame_output,
   frame_real);
      if (error < DELTA) {
```

```
        // el frame output es igual al frame real
        frames_iguales++;
      } else {
        sum_error += error;
      }
    }

    cout << "Hay " << frames_iguales << " frames iguales de " << frames
   << ", o: " << setprecision(4) << double(frames_iguales)/double(frames
   ) << "%" << endl;

    return sum_error/double(frames - frames_iguales);
  }

  static double video_prom_peak_to_signal_noise_ratio(const vector<
   vector<vector<double> > > &output, const vector<vector<vector<double>
    > > &real) {
    ASSERT(output.size() != 0 && output[0].size() != 0 && output[0][0].
   size() != 0);
    ASSERT(real.size() != 0 && real[0].size() != 0 && real[0][0].size()
   != 0);
    ASSERT(output.size() == real.size() && output[0].size() == real[0].
   size() && output[0][0].size() <= real[0][0].size());

    int ancho = output.size();
    int alto = output[0].size();
    int frames = output[0][0].size();
    int frames_iguales = 0;
    double sum_error = 0;

    vector<vector<double> > frame_output(ancho, vector<double>(alto, 0))
   ;
    vector<vector<double> > frame_real(ancho, vector<double>(alto, 0));
    for (int k = 0; k < frames; k++) {
      for(int x = 0; x < ancho; x++){
        for(int y = 0; y < alto; y++){
          frame_output[x][y] = output[x][y][k];
          frame_real[x][y] = real[x][y][k];
        }
      }
      double error = frame_peak_to_signal_noise_ratio(frame_output,
   frame_real);
      if (error < DELTA) {
        // el frame output es igual al frame real
        frames_iguales++;
      } else {
        sum_error += error;
      }
    }

    cout << "Hay " << frames_iguales << " frames iguales de " << frames
   << ", o: " << setprecision(4) << double(frames_iguales)/double(frames
   ) << "%" << endl;
```

```
  return sum_error/double(frames - frames_iguales);
}

static void error_cuadratico_medio_per_frame(const vector<vector<
 vector<double> > > &output, const vector<vector<vector<double> > > &
 real, vector<double> &resultados) {
  /*
  ASSERT(output.size() != 0 && output[0].size() != 0 && output[0][0].
 size() != 0);
  ASSERT(real.size() != 0 && real[0].size() != 0 && real[0][0].size()
 != 0);
  ASSERT(output.size() == real.size() && output[0].size() == real[0].
 size() && output[0][0].size() <= real[0][0].size());
  */
  int ancho = output.size();
  int alto = output[0].size();
  int frames = output[0][0].size();
  //int frames_iguales = 0;
  //double sum_error = 0;

  vector<vector<double> > frame_output(ancho, vector<double>(alto, 0))
 ;
  vector<vector<double> > frame_real(ancho, vector<double>(alto, 0));
  for (int k = 0; k < frames; k++) {
    for(int x = 0; x < ancho; x++){
      for(int y = 0; y < alto; y++){
        frame_output[x][y] = output[x][y][k];
        frame_real[x][y] = real[x][y][k];
      }
    }
    double error = frame_error_cuadratico_medio(frame_output,
 frame_real);
    if (error < DELTA) {
      // el frame output es igual al frame real
      //frames_iguales++;
    } else {
      //sum_error += error;
      resultados[k] = error;
    }
  }

  //cout << "Hay " << frames_iguales << " frames iguales de " <<
 frames << ", o: " << setprecision(4) << double(frames_iguales)/double
 (frames) << "%" << endl;

  //return sum_error/double(frames - frames_iguales);
}

static void peak_to_signal_noise_per_frame(const vector<vector<vector<
 double> > > &output, const vector<vector<vector<double> > > &real,
 vector<double> &resultados) {
  ASSERT(output.size() != 0 && output[0].size() != 0 && output[0][0].
 size() != 0);
  ASSERT(real.size() != 0 && real[0].size() != 0 && real[0][0].size()
```

```
  != 0);
   ASSERT(output.size() == real.size() && output[0].size() == real[0].
  size() && output[0][0].size() <= real[0][0].size());

   int ancho = output.size();
   int alto = output[0].size();
   int frames = output[0][0].size();
   //int frames_iguales = 0;
   //double sum_error = 0;

   vector<vector<double> > frame_output(ancho, vector<double>(alto, 0))
  ;
   vector<vector<double> > frame_real(ancho, vector<double>(alto, 0));
   for (int k = 0; k < frames; k++) {
     for(int x = 0; x < ancho; x++){
       for(int y = 0; y < alto; y++){
         frame_output[x][y] = output[x][y][k];
         frame_real[x][y] = real[x][y][k];
       }
     }
     double error = frame_peak_to_signal_noise_ratio(frame_output,
   frame_real);
     if (error < DELTA) {
       // el frame output es igual al frame real
       //frames_iguales++;
     } else {
       //sum_error += error;
       resultados[k] = error;
     }

   }

   //cout << "Hay " << frames_iguales << " frames iguales de " <<
  frames << ", o: " << setprecision(4) << double(frames_iguales)/double
  (frames) << "%" << endl;

   //return sum_error/double(frames - frames_iguales);
}

static void video_a_texto(const char* videofile, const char* textfile,
   int salto = 1) {
   cout << "Convirtiendo␣video␣" << videofile << "␣a␣texto␣" <<
  textfile << "␣con␣salto␣" << salto << endl;
   char command[1024];
   sprintf(command, "python␣tools/videoToTextfile.py␣%s␣%s␣%d␣>>␣/dev/
  null",
     videofile, textfile, salto);
   if(system(command)) { cout << "videoToTextfile␣failed" << endl; };
}

static void texto_a_video(const char* textfile, const char* videofile)
   {
   cout << "Convirtiendo␣texto␣" << textfile << "␣a␣video␣" <<
  videofile << endl;
```

```
    char command[1024];
    sprintf(command, "python␣tools/textfileToVideo.py␣%s␣%s␣>>␣/dev/null
   ",
      textfile, videofile);
    if(system(command)) { cout << "textfileToVideo␣failed" << endl; };
  }

  enum Funcion : int {F_CONSTANTE = 0, F_LINEAL = 1, F_CUADRATICA = 2,
   F_CUBICA = 3};

  static int random_in_range(int min, int max) {
    srand (time(NULL));
    return min + (rand() % (max - min + 1));
  }

  static vector<double> generarEsperados(Funcion funcion, double rango,
   double incremento) {
    vector<double> ret;
    if (funcion == F_CONSTANTE) {
      int a = random_in_range(1,10);
      for (double i = 0; i < rango; i+= incremento) {
        ret.push_back(a);
      }
    } else if (funcion == F_LINEAL) {
      int a = random_in_range(1,10);
      int b = random_in_range(1,10);
      for (double i = 0; i < rango; i+= incremento) {
        ret.push_back(a + b*i);
      }

    } else if (funcion == F_CUADRATICA) {
      int a = random_in_range(1,10);
      int b = random_in_range(1,10);
      int c = random_in_range(1,10);
      for (double i = 0; i < rango; i+= incremento) {
        ret.push_back(a + b*i + c*i*i);
      }
    } else if (funcion == F_CUBICA) {
      int a = random_in_range(1,10);
      int b = random_in_range(1,10);
      int c = random_in_range(1,10);
      int d = random_in_range(1,10);
      for (double i = 0; i < rango; i+= incremento) {
        ret.push_back(a + b*i + c*i*i + d*i*i*i);
      }
    }
    return ret;
  }

  static void test_interpolacion_funcion(MetodoInterpolacion metodo,
   Funcion funcion, double rango, double incremento, double precision =
   DELTA, int tamanio_bloques = 2) {
    vector<double> esperados(generarEsperados(funcion, rango, incremento
   ));
```

```
    vector<double> y;
    int salto = (int)(double(1)/incremento);
    for (unsigned int i = 0; i < esperados.size(); i+= salto) {
      y.push_back(esperados[i]);
    }

    vector<double>::const_iterator first = esperados.begin();
    vector<double>::const_iterator last = esperados.begin() + rango*
   salto - salto + 1;

    if (metodo == VECINOS) {
      InterpolacionVecinos vecinos(y, salto-1);
      assert_interpolacion(&vecinos, vector<double>(first, last),
   incremento, precision);
    } else if (metodo == LINEAL) {
      InterpolacionLineal lineal(y, salto-1);
      assert_interpolacion(&lineal, vector<double>(first, last),
   incremento, precision);
    } else if (metodo == SPLINES) {
      Spline spline(y);
      assert_interpolacion(&spline, vector<double>(first, last),
   incremento, precision);
    } else if (metodo == MULTI_SPLINES) {
      MultiSpline multi_spline(y, tamanio_bloques-1);
      assert_interpolacion(&multi_spline, vector<double>(first, last),
   incremento, precision);
    }
  }
}

#endif // TEST_UTILS_H_INCLUDED
```