

Laboratorio de Métodos Numéricos

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

Trabajo Práctico Número 2

Un juego de niños

Integrante	LU	Correo electrónico
Ciruelos Rodríguez, Gonzalo	063/14	gonzalo.ciruelos@gmail.com
Costa, Manuel José Joaquín	035/14	manuc94@hotmail.com
Gatti, Mathias Nicolás	477/14	mathigatti@gmail.com

asdf

key1 key2 key3 key4

Índice

1. Introducción teórica	3
2. Desarrollo	4
2.1. Convenciones	4
2.2. Métodos numéricos usados	4
2.2.1. Nearest neighbour	4
2.2.2. Interpolación lineal fragmentaria	4
2.2.3. Splines cúbicos	5
2.3. Estructuración del código	5
2.3.1. Nearest neighbour	6
2.3.2. Interpolación lineal fragmentaria	6
2.3.3. Splines cúbicos	6
2.4. Experimentación	7
3. Resultados y discusión	8
3.1. Análisis de rendimiento	8
3.1.1. Complejidades de los métodos	8
3.1.2. Cantidad de cuadros a agregar	9
3.1.3. Tamaño de bloque	11
3.1.4. Resolución	12
3.2. Análisis cuantitativo del error	13
3.2.1. <code>darthvader</code>	16
3.2.2. <code>ff6</code>	16
3.2.3. <code>motocross</code>	16
3.2.4. <code>penal</code>	17
3.2.5. Análisis del error a lo largo del tiempo	17
3.3. Análisis cualitativo de los métodos	19
3.3.1. Video de control	19
3.3.2. Movimiento rápido de cámara	20
3.3.3. Movimiento rápido de objetos	21
3.3.4. Cambio de Cámara	21
4. Conclusiones	22
5. Apéndices	23
5.0.5. Videos de Referencia	24

1. Introducción teórica

El objetivo del presente informe es resolver un problema práctico mediante el modelado matemático del mismo. Este problema consiste en generar videos en cámara lenta dados los videos originales, utilizando métodos de interpolación numérica.

Por lo tanto, vamos a tener que colocar más cuadros entre cada par de cuadros consecutivos del video original. Para esto, vamos a pensar un pixel del video a lo largo de todos los frames.

$$p_{ij}(f)$$

Donde el pixel (ij-ésimo) de un video depende del frame. Luego, la idea va a ser interpolar esta función para obtener los potenciales valores intermedios.

Entonces, por ejemplo, si tenemos un video de 1x1 píxeles, y los valores

$$p_{11}(0), p_{11}(1), \dots, p_{11}(n)$$

Si nos piden colocar 2 nuevos frames entre cada frame viejo, vamos a estar interesados en los valores

$$\begin{aligned} & p_{11}(0), p_{11}\left(\frac{1}{3}\right), p_{11}\left(\frac{2}{3}\right), p_{11}(1), p_{11}\left(\frac{4}{3}\right), p_{11}\left(\frac{5}{3}\right), p_{11}(2), \dots, \\ & p_{11}(n-1), p_{11}\left(n-\frac{2}{3}\right), p_{11}\left(n-\frac{1}{3}\right), p_{11}(n) \end{aligned}$$

Entonces la idea es fijar un algoritmo de interpolación y obtener todos esos valores, para luego poder rearmar el video.

Estos métodos, además de permitirnos realizar cámaras lentas, también permiten realizar interpolación para otros fines. Por ejemplo, al streamear un video, podríamos bien no recibir todos los cuadros del video, y que el reproductor los interpole acordemente para generar un video fluido. Otro ejemplo de uso es una cámara – de mala calidad – que no puede filmar a 24 cuadros por segundo, pero que sin embargo con estos algoritmos, a partir de videos de menor framerate, podemos generar videos fluidos.

2. Desarrollo

2.1. Convenciones

2.2. Métodos numéricos usados

Como dijimos en la introducción, nuestro objetivo será, dada una lista de valores que toma un pixel a lo largo de diferentes frames, obtener una función que los interpole y luego usarla para obtener valores intermedios.

Dados,

$$p_0, p_1, p_2, \dots, p_n$$

Queremos obtener

$$p_{ij}(f)$$

Tal que

$$p_{ij}(0) = p_0, p_{ij}(1) = p_1, p_{ij}(2) = p_2, \dots, p_{ij}(n) = p_n$$

Y el resto de los valores intermedios se obtienen usando p_{ij} . Entonces, si nos piden colocar k cuadros entre 2 cuadros existentes (supongamos cuadro 0 y cuadro 1 por simplicidad), debemos computar los siguientes valores:

$$p_{ij}\left(\frac{1}{k+1}\right), p_{ij}\left(\frac{2}{k+1}\right), p_{ij}\left(\frac{3}{k+1}\right), \dots, p_{ij}\left(\frac{k}{k+1}\right)$$

Y análogamente para el resto de los cuadros.

Ahora pasemos a ver los métodos implementados y analizados en este trabajo.

2.2.1. Nearest neighbour

El primer método, llamado vecino más cercano o *nearest neighbour* en inglés, se basa en interpolar un punto por el valor del punto más cercano conocido. Formalmente, si $(x_1, y_1), \dots, (x_n, y_n)$ son nuestros puntos de dato, la función que interpola sería:

$$\begin{aligned} f(x) &= f(\arg \min_i |x - x_i|) \\ f(x_i) &= y_i \end{aligned}$$

En el caso particular de este problema, es más simple aún. Si tenemos que colocar k cuadros entre 2 cuadros consecutivos c_1 y c_2 , simplemente podemos poner $\frac{k}{2}$ cuadros con el valor de c_1 y luego $\frac{k}{2}$ cuadros con el valor de c_2 .

En el caso de que k sea impar, simplemente tenemos un valor en el medio al que podemos darle cualquiera de los 2 valores, es una decisión de diseño. En el caso de este trabajo, ese valor es c_2

2.2.2. Interpolación lineal fragmentaria

La interpolación lineal (de a trozos) es la interpolación polinomial de a trozos más simple. Consiste en unir una serie de puntos

$$(x_0, f(x_0)), (x_1, f(x_1)), (x_2, f(x_2)), \dots, (x_n, f(x_n))$$

por líneas rectas. Una desventaja de este método, como analizaremos en la experimentación, es que no necesariamente la función resultante va a ser diferenciable en los x_i , algo generalmente deseable, sobre todo en este caso en que la no diferenciabilidad se vería reflejada en cambios bruscos en el video.

En general, dados los puntos $(x_i, f(x_i)), (x_{i+1}, f(x_{i+1}))$, si queremos interpolarlos con una recta obtenemos

$$l_i(x) = f(x_i) + \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}(x - x_i)$$

Entonces, el algoritmo se basa en esa fórmula, evaluándola en los puntos que se nombraron anteriormente.

2.2.3. Splines cúbicos

Los splines cúbicos son una interpolación polinómica (cúbica) de a trozos, que se basa en pedirle condiciones fuertes a los polinomios cúbicos que interpolan cada trozo de la función. En el caso del spline natural, si le llamamos S_i al i ésimo spline cúbico,

1. $S_i(x_i) = S_i(x_{i+1})$, es decir, que la función resultante sea continua en todo punto.
2. $S'_i(x_i) = S'_i(x_{i+1})$, es decir, que la función resultante sea derivable en todo punto.
3. $S''_i(x_i) = S''_i(x_{i+1})$, es decir, que la función resultante sea segundo derivable en todo punto.
4. $S''_0(x_0) = S''_{n-1}(x_n) = 0$, porque esta es la condición del spline natural.

Este método obviamente permite obtener como resultado una función más apropiada, a priori, para nuestro problema, dado que la derivabilidad nos garantiza que las transiciones van a ser más suaves que con la interpolación lineal.

En el caso de este trabajo, se requirió que la cantidad de cuadros que se tienen en cuenta para hacer el spline sea fija y a elección del usuario. Analizaremos las implicaciones de esto en la experimentación.

Para una explicación más profunda del método, se puede consultar [BF11]. Además, la implementación de splines de este trabajo también está basada en [BF11], posee algunas correcciones y está modificada para permitir realizar la interpolación de a bloques de tamaño fijo.

2.3. Estructuración del código

Para el modelado del problema utilizamos una interfaz muy simple. El archivo `tp-main.cpp` se ocupa de leer el todo el input, llamar a la rutina interpoladora correspondiente para cada pixel, y luego preparar el resultado para finalmente escribirlo en el archivo de output.

Lo que hacemos es tener una variable `frames` de tipo `std::vector<std::vector<unsigned int>>`, en la que el vector `frames[k]` representa los valores del pixel k en cada frame (pensamos al video como una tira de pixeles en lugar de una matriz de pixeles).

Entonces, simplemente, para cada k entre 0 y $ancho * alto$, tenemos un vector $frames[k]$, que podemos pensar como una función $k \mapsto frames[k]$; y esta función será la que interpolemos, como explicamos anteriormente.

Ahora pasaremos a describir brevemente los métodos utilizados.

2.3.1. Nearest neighbour

```
vector<unsigned int> nn(vector<unsigned int> valores, int cuadros)
```

- valores será, como dijimos antes, $frames[k]$, para algun k .
- cuadros serán la cantidad de valores nuevos a colocar entre dos valores consecutivos de valores.

El código es muy simple, y respeta lo explicado anteriormente cuando se describió el método

2.3.2. Interpolación lineal fragmentaria

```
vector<unsigned int> lineal(vector<unsigned int> valores, int cuadros)
```

- valores será $frames[k]$, para algun k .
- cuadros serán la cantidad de valores nuevos a colocar entre dos valores consecutivos de valores.

Como dijimos anteriormente, la interpolación lineal fragmentaria se basa en, dados los puntos $(a, f(a)), (b, f(b))$, interpolar entre ellos con

$$\frac{f(b) - f(a)}{b - a}(x - a) + f(a)$$

En este caso, la formula para un k dado sería,

$$\frac{valores[k + 1] - valores[k]}{k + 1 - k}(x - k) + valores[k] = (valores[k + 1] - valores[k])(x - k) + valores[k]$$

Entonces, hacemos como explicamos anteriormente, y en el caso de que $cuadros = c$, basta con tomar los siguientes valores para $x - k$:

$$\frac{1}{c + 1}, \frac{2}{c + 1}, \dots, \frac{c}{c + 1}$$

2.3.3. Splines cúbicos

```
vector<unsigned int> splines(vector<unsigned int> valores, int cuadros, int radio)
```

- valores será $frames[k]$, para algun k .
- cuadros serán la cantidad de valores nuevos a colocar entre dos valores consecutivos de valores.
- radio será la cantidad de cuadros que contendrá cada bloque. No es necesario que radio divida a la cantidad de cuadros, en ese caso el ultimo bloque simplemente tendrá menos cuadros (sí es necesario que el último bloque tenga más de 3 cuadros, para poder resolver el sistema).

La función splines simplemente lo que hace es partir al video en bloques, y para cada bloque generar el spline correspondiente e interpolar allí utilizandolo. Para llevar esto a cabo, se utilizará la función:

```
void splines_bloque(vector<double> ys, int cuadros, vector<double>* resultado)
```

Esta función se ocupará de obtener el spline que interpola a los valores de **ys** utilizando el algoritmo de [BF11] modificado para que funcione correctamente. Luego, hará **push_back** de los resultados en **resultado**.

2.4. Experimentación

La experimentación del presente trabajo se divide a grandes rasgos en tres partes:

- Primero nos ocuparemos de las mediciones de tiempos y análisis de complejidad de los algoritmos y métodos utilizados.
- Luego, nos centraremos en lo que concierne la medición del error de los resultados obtenidos a partir de los métodos, desde un punto de vista cuantitativo.
- Finalmente, nos centraremos en un análisis cualitativo de los resultados, intentando analizar subjetivamente los resultados obtenidos, especialmente buscando *artifacts* o errores que la simple medición del error numérico no nos permite percibir.

En cada parte explicaremos la metodología con la que realizamos los experimentos correspondientes, además de analizar en profundidad los resultados obtenidos.

3. Resultados y discusión

3.1. Análisis de rendimiento

A continuación realizaremos un análisis de los tiempos insumidos por los métodos, el cual estará dividido en dos partes: primero, en forma preliminar para tener una orientación de qué variables estudiar, un análisis superficial de la complejidad algorítmica de cada método; y luego, la contrastación empírica.

La PC sobre la que se tomaron los tiempos cuenta con un procesador Intel(R) Core(TM) i5-2500 @ 3.30GHz y 8GB de RAM.

3.1.1. Complejidades de los métodos

Utilizaremos la siguiente notación:

- *cuadros_originales*: la cantidad de cuadros que tiene el video original.
- *píxeles*: la cantidad de píxeles que tiene cada *frame* del video.
- *cuadros*: la cantidad de cuadros que se desea agregar entre los existentes del video original.

En el caso de *nearest neighbour* es bastante fácil darse cuenta que la complejidad es

$$O(\text{cuadros_originales} \times \text{píxeles} \times \text{cuadros})$$

pues por cada píxel de cada cuadro copiamos *cuadros* píxeles (lo que nos cuesta $O(1)$).

Para interpolación lineal pasa más o menos lo mismo, con la diferencia de que en lugar de copiar el píxel realiza unas pocas operaciones básicas que no dejan de ser $O(1)$, por lo que su complejidad es igual a la anterior.

Para analizar splines, vamos a considerar la partición en n bloques de tamaño b_1, \dots, b_n (o sea, ni siquiera suponemos que todos los bloques sean del mismo tamaño). Como por cada píxel se llama a la función **splines**, simplemente calculemos su complejidad y luego la multiplicamos por *píxeles*.

El costo de **splines** es la sumatoria de los costos de llamar a **splines_bloque** por cada bloque. La complejidad de esta última función es

$$O(b_i + b_i \times \text{cuadros}) = O(b_i \times \text{cuadros})$$

pues primero se resuelve el sistema con costo lineal ¹ sobre el tamaño del bloque, y luego por cada elemento del bloque se realizan *cuadros* iteraciones de costo $O(1)$. Entonces nos queda que la complejidad de **splines** es

$$\begin{aligned} b_1 \times \text{cuadros} + b_2 \times \text{cuadros} + \dots + b_n \times \text{cuadros} &= \text{cuadros} \times (b_1 + \dots + b_n) \\ &= \text{cuadros} \times \text{cuadros_originales} \end{aligned}$$

Como esto había que multiplicarlo por *píxeles*, la complejidad total del método termina siendo igual a la de los dos métodos anteriores, aunque en este caso las constantes que se están ignorando son más significativas.

¹la complejidad exacta del algoritmo usado para resolver el sistema tridiagonal puede encontrarse en el capítulo 6 de [BF11]

De esto deducimos entonces que las variables a analizar en la experimentación deben ser *cuadros*, *cuadros_originales* y *pixeles* (es decir, la resolución del video).

3.1.2. Cantidad de cuadros a agregar

Para esta sección consideramos variaciones con menos cuadros (para poder experimentar con mayor comodidad) de tres videos:

- ff6: Versión de 22 cuadros, resolución 400x225
- darthvader: Versión de 15 cuadros, resolución 400x225
- penal: Versión de 18 cuadros, resolución 560x315

En base a los resultados de la sección anterior, es de esperar que para todos los métodos el tiempo de cómputo sea lineal sobre la cantidad de cuadros que se desean agregar. Sin embargo, está claro que las pendientes de cada método serán distintas pues las constantes lo son. Las hipótesis son relitativamente obvias: el método de vecino más cercano será el más veloz pues no realiza ningún cálculo sino que simplemente copia píxeles; lineal será el segundo más rápido pues a diferencia del anterior se agrega el cálculo de un polinomio de grado 1. Claramente los splines serán los que más tarden pues requieren evaluar un polinomio de grado 3 por cada cuadro a agregar. A su vez, es de esperar que independientemente del tamaño de bloque considerado para splines (4, 8 o 12), el tiempo de cómputo sea el mismo. Esto es consecuencia inmediata de que la complejidad del método, como vimos en la sección anterior, no depende de la partición escogida.

A continuación presentamos los resultados. Los mismos se obtuvieron de realizar 20 iteraciones para cada método y cantidad de cuadros a agregar.

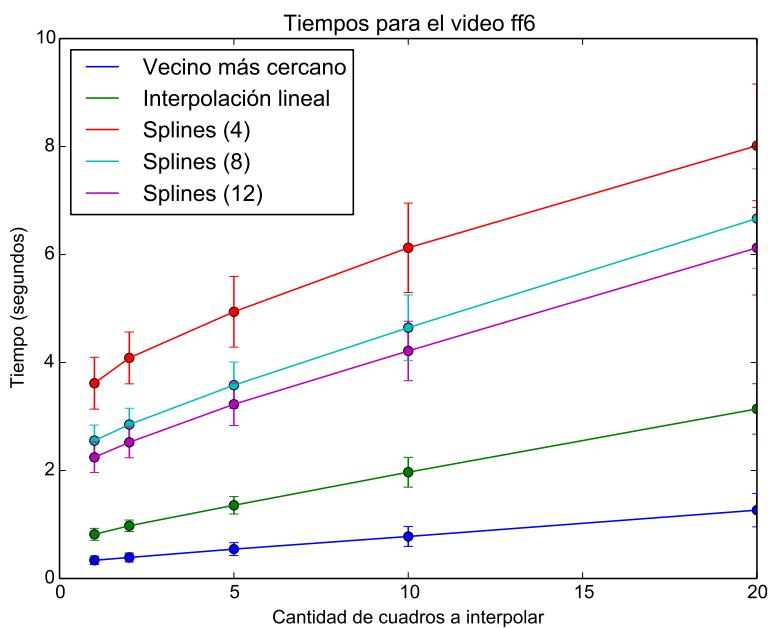


Figura 1: Tiempo que cuesta aplicar cada método sobre ff6 en función de la cantidad de cuadros que se desean agregar. Los puntos representan la media 0.20-podada de cada muestra. Las barras verticales indican la varianza.

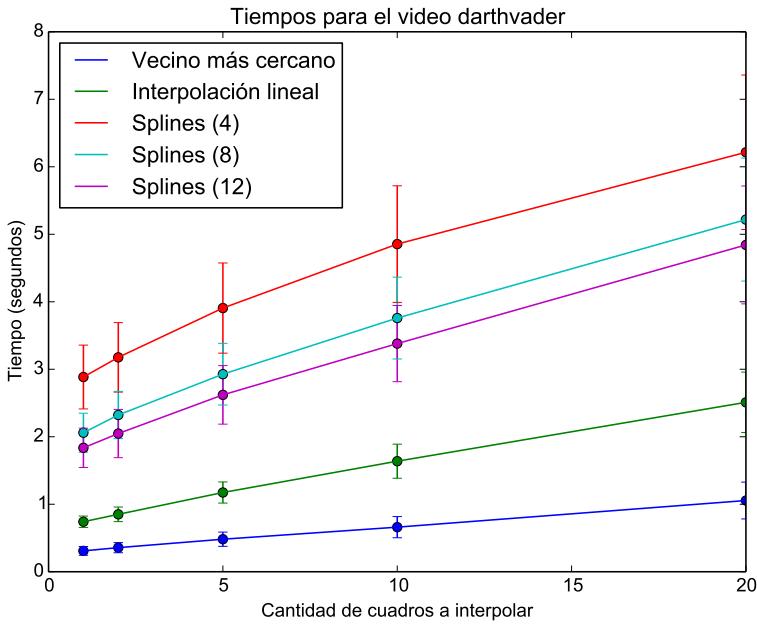


Figura 2: Tiempo que cuesta aplicar cada método sobre darthvader en función de la cantidad de cuadros que se desean agregar. Los puntos representan la media 0.20-podada de cada muestra. Las barras verticales indican la varianza.

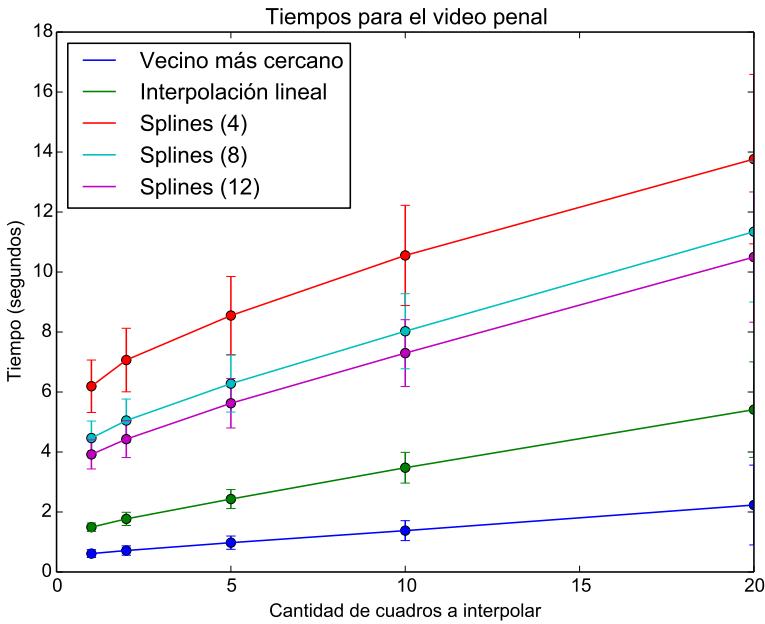


Figura 3: Tiempo que cuesta aplicar cada método sobre penal en función de la cantidad de cuadros que se desean agregar. Los puntos representan la media 0.20-podada de cada muestra. Las barras verticales indican la varianza.

Al ver los gráficos vemos que se sostienen las hipótesis respecto de los métodos de vecino más cercano e interpolación lineal, pero curiosamente las tres variantes de splines muestran tiempos distintos. No solo eso, sino que consistentemente en todos los casos a mayor tamaño de bloque, menos tarda en computar el método.

Si observamos con un poco más de detenimiento las figuras 2 y 3 vemos que las varianzas de los splines son altas y en muchos casos se superponen las tres, pero la figura 1 que muestra una varianza un poco más acotada parece señalar que la tendencia marcada en el párrafo anterior efectivamente existe. Parece razonable atribuirle gran parte de esto al hecho de que, al considerar particiones de bloques más pequeños, se llama más veces a la función `splines_bloque` y se entra más al ciclo `while` de `splines`, lo que provoca que las constantes que despreciamos al calcular la complejidad incidan más en la brecha de tiempos observada. Más específicamente, son las constantes que sumaban las que provocan la diferencia, pues al ver el gráfico vemos que lo que ocurre es un corrimiento sobre el eje de ordenadas pero no un cambio de curvatura.

3.1.3. Tamaño de bloque

Las hipótesis son escencialmente análogas al caso anterior. Ahora con la corrección de que seguramente veamos un fenómeno similar al visto antes con respecto a los tres métodos de splines.

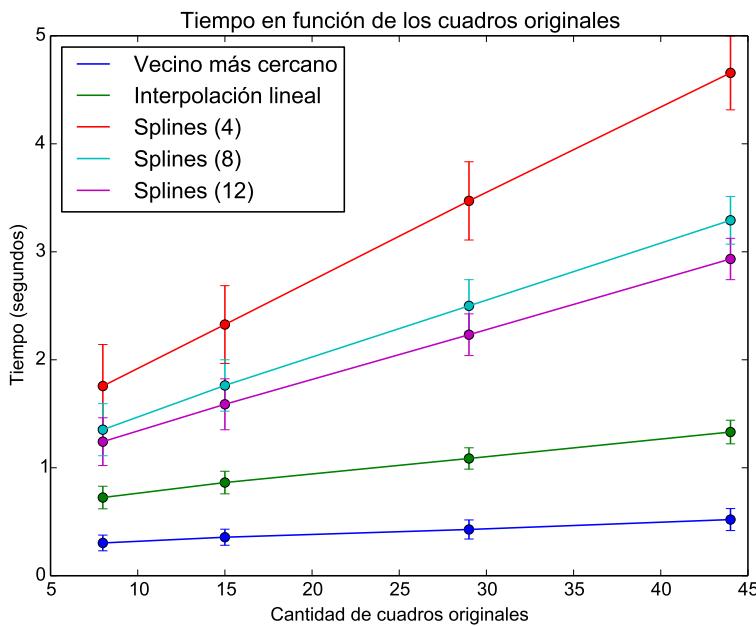


Figura 4: Tiempo que se requiere al aplicar cada método sobre un video para interpolar 2 cuadros en función de la cantidad de cuadros que poseía originalmente. Los puntos representan la media 0.20-podada de cada muestra. Las barras verticales indican la varianza.

La figura 4 efectivamente corrobora lo esperado.

Algo interesante que puede observarse en este gráfico es que, a diferencia de lo que pasaba al aumentar los cuadros a agregar, las pendientes con las que crecen las curvas de las tres variantes de splines no son iguales. Es decir, en las figuras de la 1 a la 3 puede apreciarse que la curvatura de los puntos es prácticamente igual aunque haya un desplazamiento sobre el eje de ordenadas. Sin embargo, aquí vemos que los puntos correspondientes a la versión que usa bloques de tamaño 4 están cada vez más espaciados de los otros. Incluso entre los correspondientes a *splines (8)* y *splines (12)* puede notarse esta diferencia de pendiente aunque en menor medida.

Esto resulta razonable a la luz de los hechos vistos en el punto anterior: si habíamos dicho que el tiempo del método aumentaba cuantas más veces se caía en el `while` de la función `splines` es lógico que al aumentar la variable de la cual este hecho depende directamente esta situación se potencie. Visto con un ejemplo: si dado un video le aumentáramos en 8 la cantidad de sus cuadros originales,

para *splines (8)* y *splines (12)* esto implicaría a lo sumo caer una vez más en el `while`, mientras que para *splines (4)* resulta en hacerlo dos veces más. Si lo hiciéramos en 24 cuadros, serían 2 veces para *splines (12)*, 3 para *splines (8)* y 6 para *splines (4)*. Es decir que el aumento siempre es desigual, y eso justifica la diferencia en las pendientes.

3.1.4. Resolución

Finalmente, analicemos la influencia de la resolución. Es de suponer que el comportamiento sea similar al del caso anterior, o sea, que las pendientes de crecimiento sean distintas para todos los métodos, pues la cantidad de píxeles que conforman un *frame* es lo que determina la cantidad de veces que se llame a las funciones que implementan los métodos de interpolación, por lo cual es razonable que todas las constantes propias de cada método se vean potenciadas.

En efecto, la figura 5 verifica estas suposiciones.

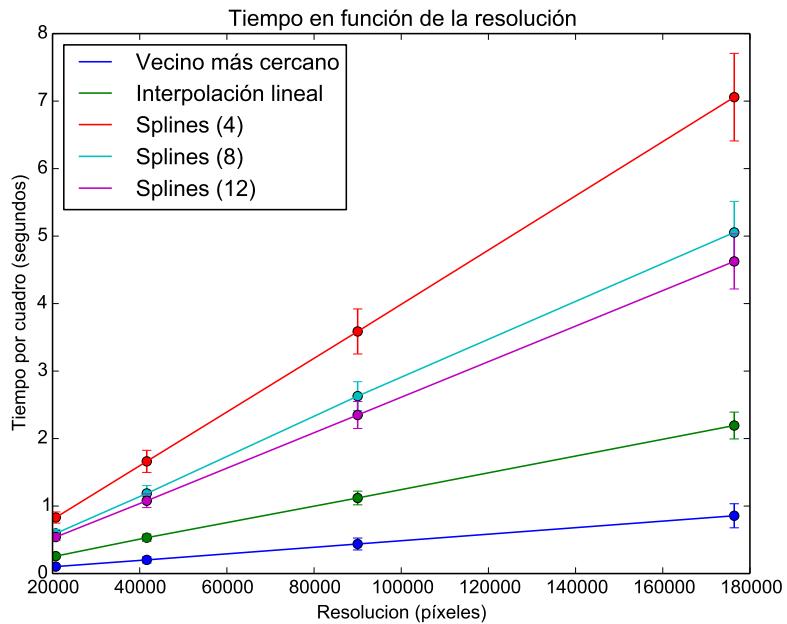


Figura 5: Tiempo que se requiere al aplicar cada método sobre un video para interpolar 2 cuadros en función de la resolución del mismo. Los puntos representan la media 0.20-podada de cada muestra. Las barras verticales indican la varianza.

3.2. Análisis cuantitativo del error

Antes de ver los resultados, hagamos un análisis de los casos de prueba que tendremos en cuenta.

- **darthvader**. El primer video contiene una cámara fija y un objeto moviéndose a una velocidad relativamente lenta, con su entorno quieto.
- **ff6**. Este es un video que contiene (a pesar de su corta duración), 6 tomas en escenarios totalmente distintos. Algunos escenarios presentan mucho movimiento, otros están prácticamente quietos. Este es un video muy interesante para analizar porque es esperable que los algoritmos de interpolación en los que importa sobre todo información local (vecinos más cercanos, interpolación lineal fragmentaria) funcionen mejor que aquellos que toman información global (splines). Analizaremos todo esto más adelante.
- **motocross**. En este video la cámara se mueve a gran velocidad, siguiendo a un objeto (una moto) que se encuentra más o menos centrada a lo largo de todo el video.
- **penal**. En este video la cámara está nuevamente fija, pero ahora hay un objeto que se mueve a velocidad medianamente rápida (pateador y arquero) y otro objeto que se mueve a una velocidad muy alta (pelota); mientras que todo el entorno se encuentra quieto.

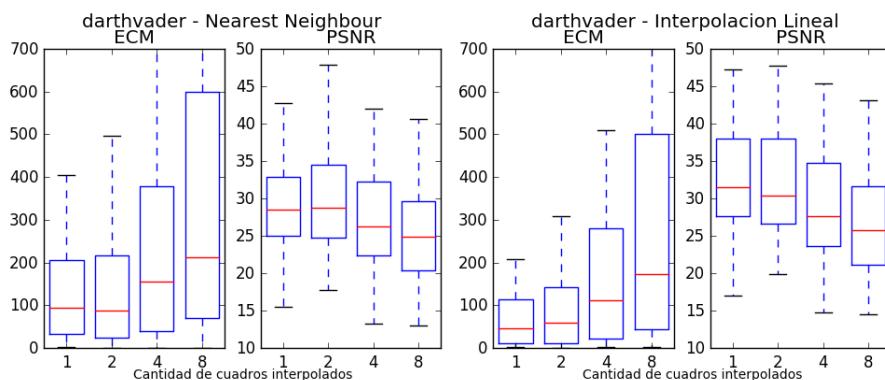
Elegimos estos videos porque creemos que representan las posibles situaciones o combinaciones que puede presentar un video de la vida real. No elegimos videos confeccionados a mano para analizar casos borde o extremos porque creemos que el análisis más interesante que se puede hacer está alrededor de casos reales, que son finalmente sobre los cuales se aplicarán estos algoritmos.

La metodología de experimentación fue la siguiente: extrajimos 1,2,4 u 8 cuadros por medio de cada video (utilizando el script `videoToTextfile.py`) y luego lo interpolamos con nuestros algoritmos. Finalmente, utilizando un script hecho por nosotros, comparamos los valores de los píxeles del video original contra los interpolados por nosotros.

Elegimos esa cantidad de cuadros porque más de 8 cuadros se torna demasiado para interpolar, dado que se pierden muchos detalles del video original y el error se torna realmente alto (ya sucede eso con 8 cuadros).

Por último, para analizar el algoritmo de Splines utilizamos bloques de 4, 8 y 12 cuadros porque, nuevamente, si los bloques eran más grandes el video comenzaba a tener importantes artifacts (ver sección 3.3) y entonces nos pareció adecuado poner 12 como el tamaño máximo de bloque a tomar (ya en algunos videos 12 es demasiado y el resultado es de mala calidad). Esto se debe a la localidad vs. globalidad de la que hablamos antes, dado que si los frames de un video cambian mucho a lo largo del tiempo, tomar en cuenta frames lejanos a la hora de interpolar es contraproducente.

Sin más aclaraciones que hacer, pasemos a ver y analizar los resultados que obtuvimos.



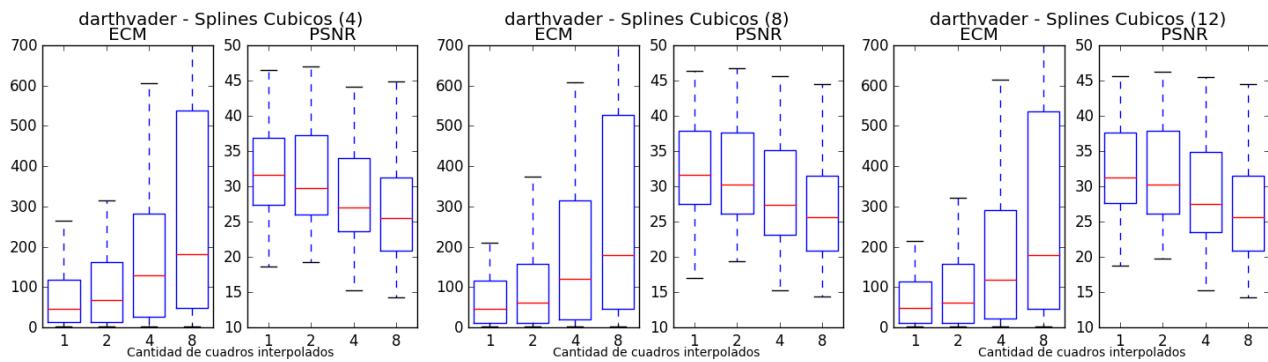


Figura 6: Resultados del cálculo de ECM y PSNR para cada algoritmo en el video *darthvader*. Para Splines cúbicos se indica además el tamaño de los bloques utilizados.

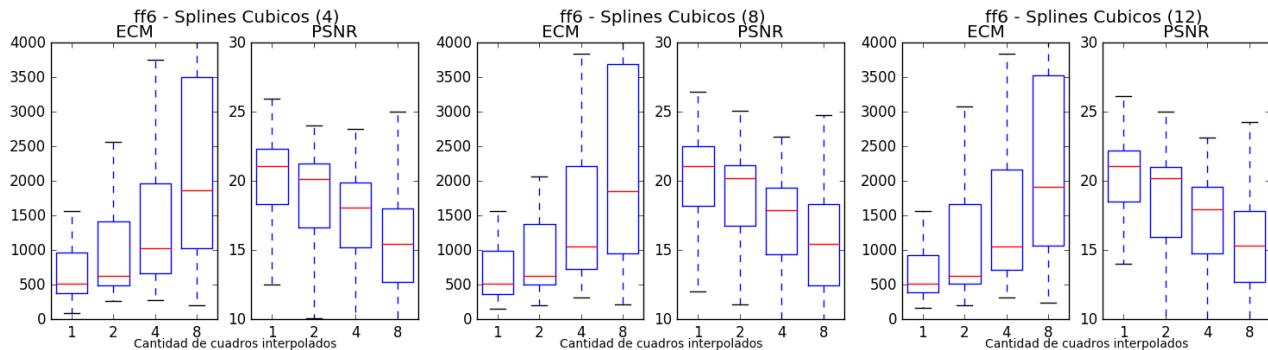
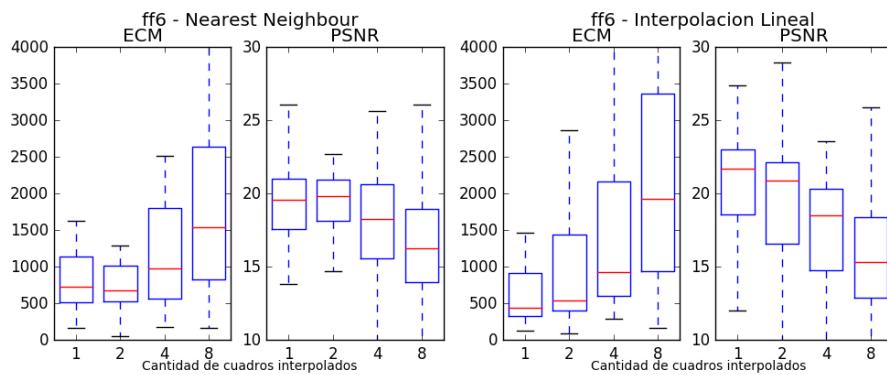
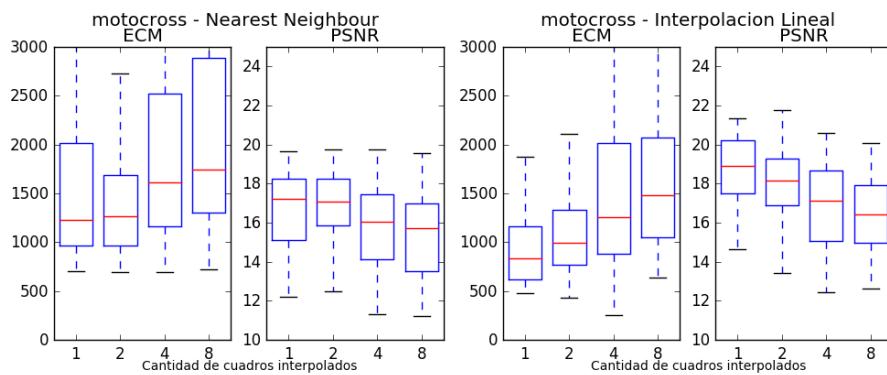


Figura 7: Resultados del cálculo de ECM y PSNR para cada algoritmo en el video *ff6*. Para Splines cúbicos se indica además el tamaño de los bloques utilizados.



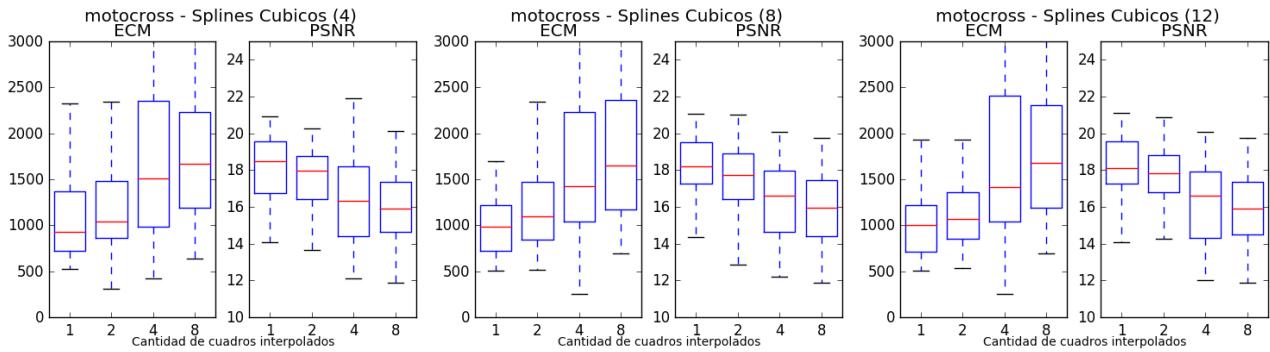


Figura 8: Resultados del cálculo de ECM y PSNR para cada algoritmo en el video **motocross**. Para Splines cúbicos se indica además el tamaño de los bloques utilizados.

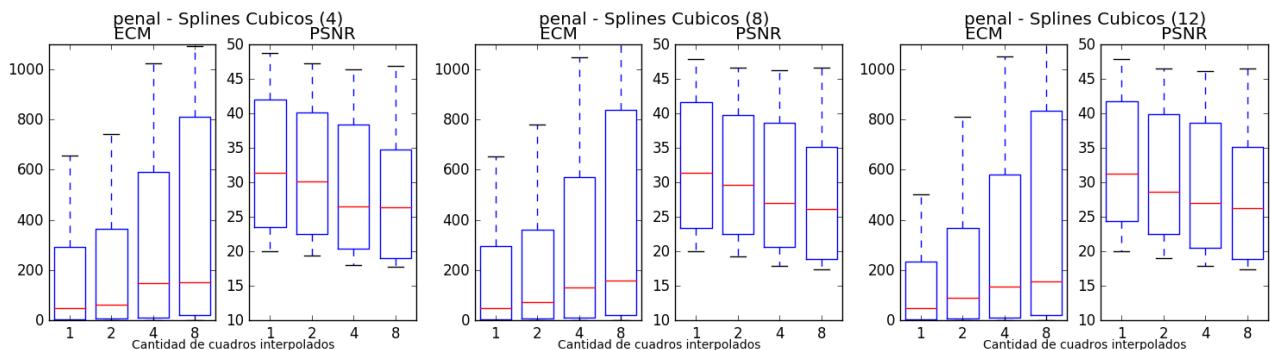
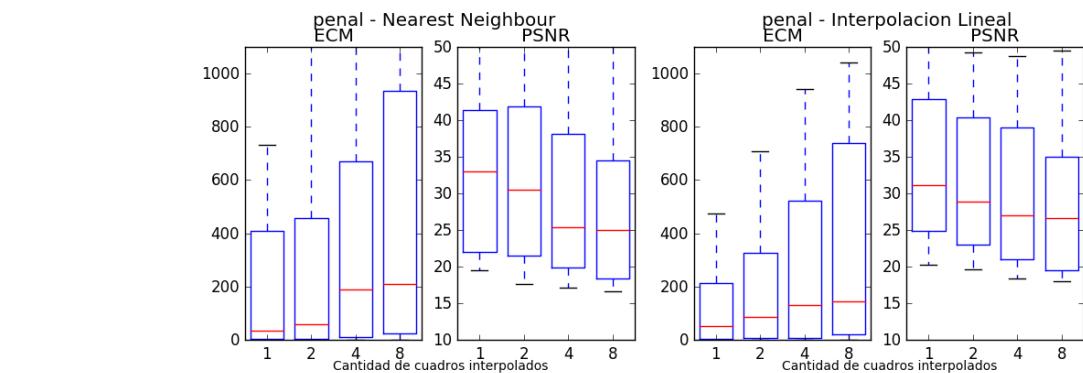


Figura 9: Resultados del cálculo de ECM y PSNR para cada algoritmo en el video **penal**. Para Splines cúbicos se indica además el tamaño de los bloques utilizados.

Antes de analizar los resultados, vale la pena explicar los gráficos que aparecen en las páginas anteriores. Cada boxplot representa los errores (en ECM y PSNR respectivamente) de todos los frames del video. Luego analizaremos dentro de cada video porqué a veces el error es más alto y porqué es más bajo, pero para este análisis en general nos bastará con los boxplots.

Además, vale la pena explicar en qué se basan el ECM y el PSNR. El ECM (Error Cuadrático Medio) se calcula como el promedio de todos los errores al cuadrado. Por lo tanto, a más alto, más error. Para este trabajo, consideraremos que un error bajo será un ECM menor a 25, y error medio será ECM menor a 400 (dado que representan diferencias de 5 y 20 respectivamente).

El PSNR (Peak Signal-to-Noise Ratio), se define como $PSNR = 10 \log_{10}(\frac{255^2}{ECM})$. Por lo tanto, a más alto, menos error tendrá la imagen. Puede pensarse como una suerte de ECM normalizado.

Usando los mismos números de antes, consideraremos que el error es bajo si el PSNR es mayor a 35 y que es medio si es mayor a 22 (pero menor a 35).

3.2.1. darthvader

Lo primero que podemos notar (muy esperable) es que el error aumenta cuando se intentan interpolar más cuadros. Esto se debe a que se intenta obtener más cantidad de cuadros a partir de una cantidad de datos menor, lo cual obviamente deteriorará la calidad del video.

Lo segundo que notamos es que, mientras el algoritmo de nearest neighbour tiene un error relativamente grande, los algoritmos de interpolación lineal y splines cúbicos (con todos los tamaños de bloque) presentan resultados muy similares. Creemos que esto se debe a la lentitud con la que transcurren los hechos en el video provoca que cualquier interpolación razonable de los valores de buenos resultados.

3.2.2. ff6

En este caso, al igual que antes y al igual que en todos los videos que analizaremos, el error aumenta cuando se intentan interpolar más cuadros.

A diferencia que en el video anterior, en este caso se nota una diferencia entre los métodos que utilizamos. Nearest neighbour, como siempre, es el peor, teniendo errores muy altos (dado que los cambios de escenarios en este video son bruscos, un frame puede no tener nada que ver con su anterior).

Luego, podemos ver que la interpolación lineal tuvo mejor performance que los splines cúbicos. La razón de eso es probablemente que, al ser un video que tiene cambios muy brutales de escenarios, es mucho mejor obtener los cuadros interpolados con la información lo más local posible, dado que cuadros lejanos tienen muy poco que ver.

Sin embargo, cuantos cuadros se tomen en el spline no afecta demasiado al error. Esto puede explicarse porque los escenarios son relativamente inmóviles, entonces la cantidad de cuadros que se tomen (mientras sean acotados) no debería afectar demasiado porque no agregan información nueva.

3.2.3. motocross

Como siempre, el error aumenta cuando se intentan interpolar más cuadros.

En este video sucede algo similar al anterior. Nearest neighbour es el peor porque el video se mueve muy rápido, un frame tiene poco que ver con el anterior. Nuevamente, la interpolación lineal tuvo mejor performance que los splines cúbicos. La razón de esto es la misma que la de antes: al ser un video que se mueve muy rápido, es mejor interpolar con información local que con información global.

Finalmente, podemos notar una diferencia con el video anterior. Podemos observar en la figura ?? como a más grande es el bloque que se toma para hacer splines cúbicos, más grande es el error. Puede verse que para un tamaño de bloque de 4 frames, el error cuadrático medio es aproximadamente 900, para un bloque de 8 frames de 950 y para uno de 12 es de 1000.

Este hecho justifica nuestra explicación de porque en el video **ff6** el tamaño de los bloques no afectó tanto, dado que en este caso tomar bloques más grandes sí agrega nueva información, que sin embargo es perjudicial para el resultado, porque como todo se mueve muy rápidamente, la información nueva altera y distorsiona los resultados.

3.2.4. penal

Este es un video muy interesante para analizar. Primero, recordemos que en todo momento la mayor parte del video se encuentra quieta, mientras que algunos objetos que ocupan poca parte de la pantalla se mueven.

Esto produce que (cuando hay que interpolar 1 o 2 cuadros) el error cuadrático medio se minimice con el algoritmo de vecino más cercano. Esto se debe a que como la mayor parte del video está quieta, al interpolar con vecinos más cercanos, nos aseguramos que el error en estos lugares sea mínimo, y en las regiones del video que se mueven, el error también será relativamente pequeño.

Además, al interpolar 1 o 2 cuadros, el error al usar splines (con cualquier tamaño de bloques) es menor que el de interpolación lineal. Una posible explicación para esto es que splines se adapta mejor que la interpolación lineal fragmentaria a los movimientos rápidos del video, manteniendo la exactitud de los pixeles que cambian poco, con lo cual puede sacarle ventaja.

Ahora bien, cuando se intentan interpolar 4 u 8 cuadros, la situación es muy distinta. Aquí el error de el algoritmo nearest neighbour sube mucho, sobre pasando al de interpolación lineal y splines cúbicos, que están prácticamente empatados.

Explicar esto no es difícil: al tener que interpolar más cuadros, nearest neighbour empieza a aumentar su error dado que repetir un cuadro muchas veces conlleva bastante error en las partes del video de alto movimiento, dado que el error se arrastra por muchos cuadros. Los algoritmos de interpolación lineal y splines, al utilizar la información mas inteligentemente, son mas *resilientes* a la falta de información, lo cual les permite sobre pasar en rendimiento (con respecto al error) al algoritmo de nearest neighbour cuando hay que interpolar muchos cuadros.

3.2.5. Análisis del error a lo largo del tiempo

Para finalizar con el análisis cuantitativo del error de los algoritmos, nos propusimos analizar como varía el error de nuestra interpolación a lo largo del tiempo para un video dado. Para ello, utilizamos el video `darthvader`, interpolando 1 cuadro entre dos cuadros consecutivos. Veamos los resultados.

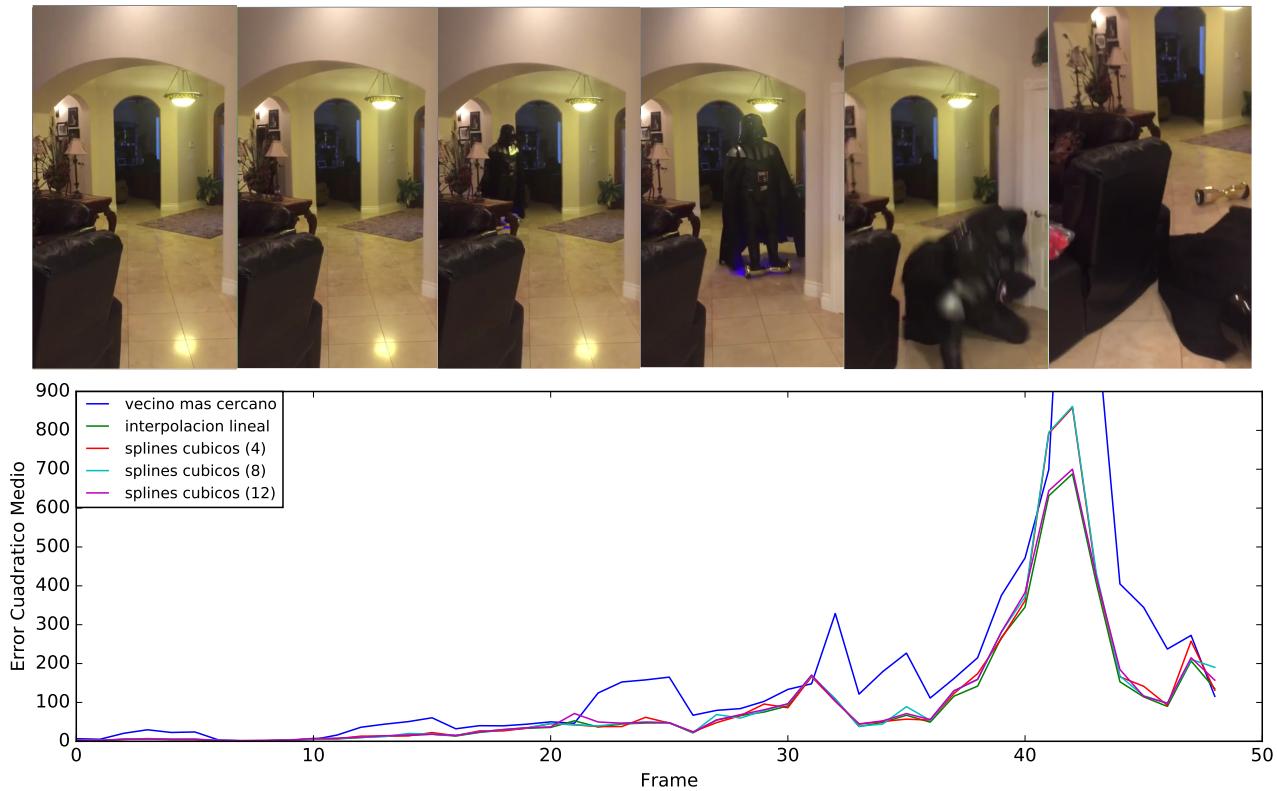


Figura 10: Error Cuadrático Medio a lo largo del tiempo para los distintos algoritmos para el video **darthvader**. Además arriba se puede ver un extracto del video en ese momento.

Como podemos observar en la figura 10, al principio del video, los frames son prácticamente iguales, entonces el error en todos los algoritmos es muy bajo, dado que interpolar es básicamente repetir el frame.

Cuando aparece en escena el objeto que se mueve, el error aumenta. Sin embargo, como la velocidad a la que se mueve es muy baja, el error sigue siendo relativamente medio-bajo.

Luego, cuando Darth Vader tropieza y se cae, el error asciende mucho. Esto se debe a que el movimiento es mucho, entonces interpolar se vuelve más difícil, y aparecen artifacts, dado que nuestros algoritmos no hacen seguimiento de objetos, sino que simplemente interpolan píxel a píxel.

Finalmente, la cámara se mueve un poco, generando error más bajo que antes, aunque sigue siendo alto, por la misma razón de la que se habló anteriormente.

3.3. Análisis cualitativo de los métodos

En esta sección nos concentraremos en las características de los métodos utilizados que escapan a nuestros análisis anteriores. Si bien logramos tener una idea de la complejidad temporal y el nivel de error que pueden tener los videos que generamos aún no vimos con suficiente claridad los efectos particulares que pueden llegar a producirse, qué tipo de errores pueden haber y cómo varían entre los distintos métodos. Esto es de lo que hablaremos en esta sección.

Un video está formado por una sucesión finita de imágenes, al reproducirlo se pasa de una imagen a la siguiente lo cual puede causar cambios en los valores de los pixels que se están visualizando. La interpolación se basará en esos pixels para calcular nuevas imágenes que se agregarán al video. En esta sección veremos de qué forma el grado de cambio entre los pixels puede afectar la calidad del video resultante tras la interpolación.

Todos los videos sobre los que trabajaremos aquí serán el resultado de aplicar alguno de los métodos de interpolación sobre los pocos videos base que describimos previamente, se detallarán siempre los métodos utilizados de forma que los experimentos realizados puedan replicarse fácilmente. Varios de ellos pueden encontrarse en el link citado en la sección del apéndice ([5.0.5](#)).

Nuestra primera hipótesis es que según la forma en que varían los pixels en el video podrían presentarse mayores dificultades y errores al intentar pasarlo a cámara lenta. Por lo tanto lo que haremos en esta sección será dividir los experimentos en cuatro tipos de video base y analizar cada caso particular con los métodos propuestos. Escogimos 4 ejemplares distintos, uno por cada categoría de video que nos pareció relevante analizar. Para cada video hicimos experimentos con cada método interpolando 1, 2, 4 y 8 cuadros. Se presentarán en esta sección los casos más representativos de nuestros hallazgos.

3.3.1. Video de control

En los próximos experimentos analizaremos videos con movimientos rápidos de objetos y cámara intentando verificar ciertas ideas sobre posibles artifacts que podrían generarse. Antes de eso utilizaremos un video de control el cual no presenta ninguna de esas características, este será un video con movimiento relajado el cual nos permitirá verificar como se comportan los métodos con videos que, según creemos, no presentarán problemas para luego compararlos con los casos más problemáticos. El caso más trivial sería un video completamente estático. Pero se ve fácilmente al analizar el funcionamiento de nuestros métodos que el resultado será un video también estático pero de mayor duración, lo cual cumpliría correctamente el resultado deseado. Un caso un poco más interesante y el cual utilizaremos como video de control es el caso en que hay cierto movimiento pero nada de gran velocidad.

Como video seleccionamos una toma de la carrera de un grupo de babosas.



Figura 11: Captura del video `babosa.avi` con el cuadruple de cuadros de lo normal calculados utilizando splines de bloque de radio 4, no se nota en ningún momento algún artifact significativo.

Al ver los resultados se corroboró que utilizando splines e interpolación lineal no había problemas y el video resultante tenía un buen efecto de cámara lenta, solo se notaron partes borrosas en las antenas, creemos que debido a que estas producen los movimientos más rápidos en el video. Con Nearest Neighbour la historia es distinta, al agregar más de 2 cuadros al video original comienza a perder mucha fluides, agregando 8 cuadros el video ya carece completamente de un movimiento natural, esta característica creemos que persistirá siempre al utilizar este método.

3.3.2. Movimiento rápido de cámara

Un caso que nos pareció relevante es el de los videos que tienen movimientos rápidos de cámara. Como ejemplar de esta situación utilizamos al video nombrado previamente *motocross*. Para este caso volvió a ocurrir con *Nearest Neighbour* que la imagen perdía fluidez a medida que se intentaban agregar más cuadros. Para *Splines* e *Interpolación Lineal* sucedió lo esperado, los movimientos rápidos del video produjeron efectos indeseados, los cuales empeoraban a medida que se agregaban más cuadros. El error puede observarse en la imagen de la izquierda.



Figura 12: A la izquierda se tiene el resultado de un cuadro calculado con Splines al interpolar de a 4 cuadros con bloques de radio 4, se observa una superposición importante entre dos imágenes. En la derecha se ve una toma del video original

Lo que sucede es que debido al gran ritmo de cambio entre cada fotograma los cuadros que se agregan al video se calculan basados en imágenes muy distintas, los métodos que utilizamos no son suficientemente buenos para predecir el cuadro adecuado y en consecuencia fallan en dar una imagen intermedia adecuada.

Otra observación que hicimos en este video fue que a simple vista no parece haber diferencia significativa entre los resultados de Splines e interpolación lineal, ambos funcionan correctamente al

agregar 1 y 2 cuadros y comienzan a fallar a un ritmo similar al comenzar a agregar más cuadros.

3.3.3. Movimiento rápido de objetos

Para este experimento utilizamos el video *penal*, este se puede ver como un caso particular del anterior. Ya que hay movimiento rápido de pixels por lo que se presentan los mismos inconvenientes pero solo sobre un pequeño conjunto de pixels, los que están en la trayectoria de la pelota y el jugador: en ese espacio es donde al utilizar splines y lineal se espera experimentar al igual que en el caso anterior un efecto de transparencia y superposición indeseable entre el fondo y el lugar donde estuvo el objeto previamente.

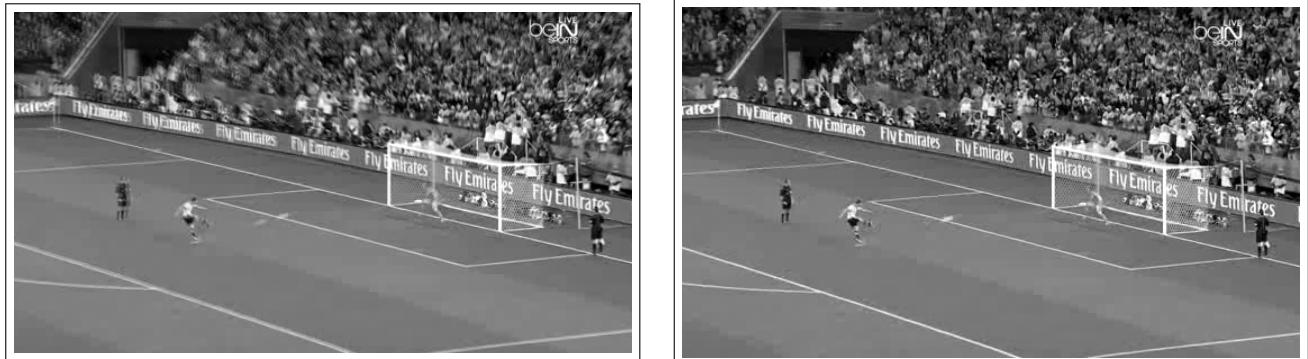


Figura 13: A la izquierda se tiene el resultado de un fotograma calculado con Splines agregando 8 cuadros con bloques de radio 4, se observa una superposición importante entre cuadros. A la derecha esta una toma del video original

Al ver los videos se corrobora lo esperado. El artifact visto en la figura aumenta cuanto mayor es la cantidad de cuadros que queremos agregar. Para *Nearest Neighbour* el efecto no existe pero la imagen tiene movimientos muy bruscos y parece más estática a medida que agregamos más cuadros.

3.3.4. Cambio de Cámara

Los cambios de cámara creemos que serán los que produzcan los mayores problemas para Splines e interpolación lineal debido a que estos producen las mayores diferencias entre cuadros contiguos, creemos que aquí en algunos casos puede llegar a ser preferible incluso *nearest neighbour* a pesar de su dinámica pobre. Para verificar nuestras ideas utilizamos el video *ff6*. También nos interesa comprobar lo charlado previamente en la sección de errores sobre qué tanto afectará el tamaño de los bloques de los Splines. Intentaremos ver si esto puede llegar a jugar un rol en que tan dramáticos sean los artifacts.



Figura 14: Distintos artifacts producidos por Splines e Interpolación lineal interpolando 4 cuadros. El del medio es Splines cúbicos interpolando 4 cuadros con 8 de radio, los otros son interpolación lineal de 8 cuadros.



Figura 15: imágenes originales.

Como se ve en las imágenes hay cuadros formados por imágenes totalmente distintas fusionadas lo cual produce un efecto indeseable. Estos problemas se ven a lo largo de todos los cambios de cámara del video. Se deduce de este experimento que no es buena idea intentar interpolar entre cambios de cámara. Una posible solución podría ser trabajar sobre cada toma por separado, hacer la interpolación deseada y luego volver a unir los subvideos obtenidos, así se evitaría generar cuadros entre tomas de lugares distintos.

Con respecto al tamaño de los bloques de Splines se pudo verificar lo charlado anteriormente en la sección de errores, con distintos tamaños de bloque (se utilizaron radios de 1, 2, 8 y 16) no se observaron cambios significativos.

Como observación final debemos decir que a simple vista no pudimos presenciar para ninguno de los casos analizados una diferencia demasiado abrupta entre interpolación lineal y Splines con distintos tamaños de bloques. Si bien en ciertas circunstancias habían mejoras con algunos de estos métodos respecto de los demás estas no superaban ampliamente a las demás y presentaban los mismos artifacts y decadencias para casos iguales de interpolación. Esta observación de seguro este relacionada con la calidad baja de los videos y la impresión del análisis a ojo por lo que quedará como trabajo a futuro la verificación y profundización de esta idea.

4. Conclusiones

5. Apéndices

5.0.5. Videos de Referencia

Los videos originales utilizados, junto con algunos de los que generamos aplicando los distintos métodos de interpolación se pueden encontrar en el siguiente link: <http://bit.ly/1HuAwnj>².

Referencias

- [BF11] R. Burden y D. Faires. *Numerical Analysis*. Brooks/Cole, 2011.

²Este link es un acortamiento de
<https://drive.google.com/drive/folders/0B5hNK09AHoDcMExwZVNQVV13Zkk>, por comodidad