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

El objetivo del presente informe es resolver un problema práctico mediante el modelado matemático del mismo. Este problema consiste en generar videos en camará 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. Entonces, 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

$$p_{11}(0), p_{11}(1), ..., p_{11}(n)$$

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

$$p_{11}(0), p_{11}\left(\tfrac{1}{3}\right), p_{11}\left(\tfrac{2}{3}\right), p_{11}(1), p_{11}\left(\tfrac{4}{3}\right), p_{11}\left(\tfrac{5}{3}\right), p_{11}(2), ...,$$
$$p_{11}(n-1), p_{11}\left(n-\tfrac{2}{3}\right), p_{11}\left(n-\tfrac{1}{3}\right), p_{11}(n)$$

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 generear 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,

$$p_0, p_1, p_2, ..., p_n$$

Queremos obtener

$$p_{ij}(f)$$

Tal que

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

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:

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

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 $(x_1, y_1), ..., (x_n, y_n)$ son nuestros puntos de dato, la función que interpola sería:

$$f(x) = f(\arg\min_i |x - x_i|)$$
$$f(x_i) = y_i$$

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 luevo $\frac{k}{2}$ 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

$$(x_0, f(x_0)), (x_1, f(x_1)), (x_2, f(x_2)), ..., (x_n, f(x_n))$$

por lineas 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 cubico,

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 mas 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 esta 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 praparar 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 funcion $k \mapsto$`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 $x - k$:

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

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 funcion:

```
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, hara **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 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 moviendose 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 estan 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. El escenario que rodea a la moto se mueve en muy rápidamente.
- `penal`. En este video la cámara esta nuevamente fija, pero ahora hay un objeto que se mueve a velocidad medianemente 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 al rededor de casos reales, que son finalmente sobre los cuales se aplicarán estos algoritmos.

La métodología 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 (lo analizaremos más adelante, cuando analicemos cualitativamente los resultados) 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 un 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.

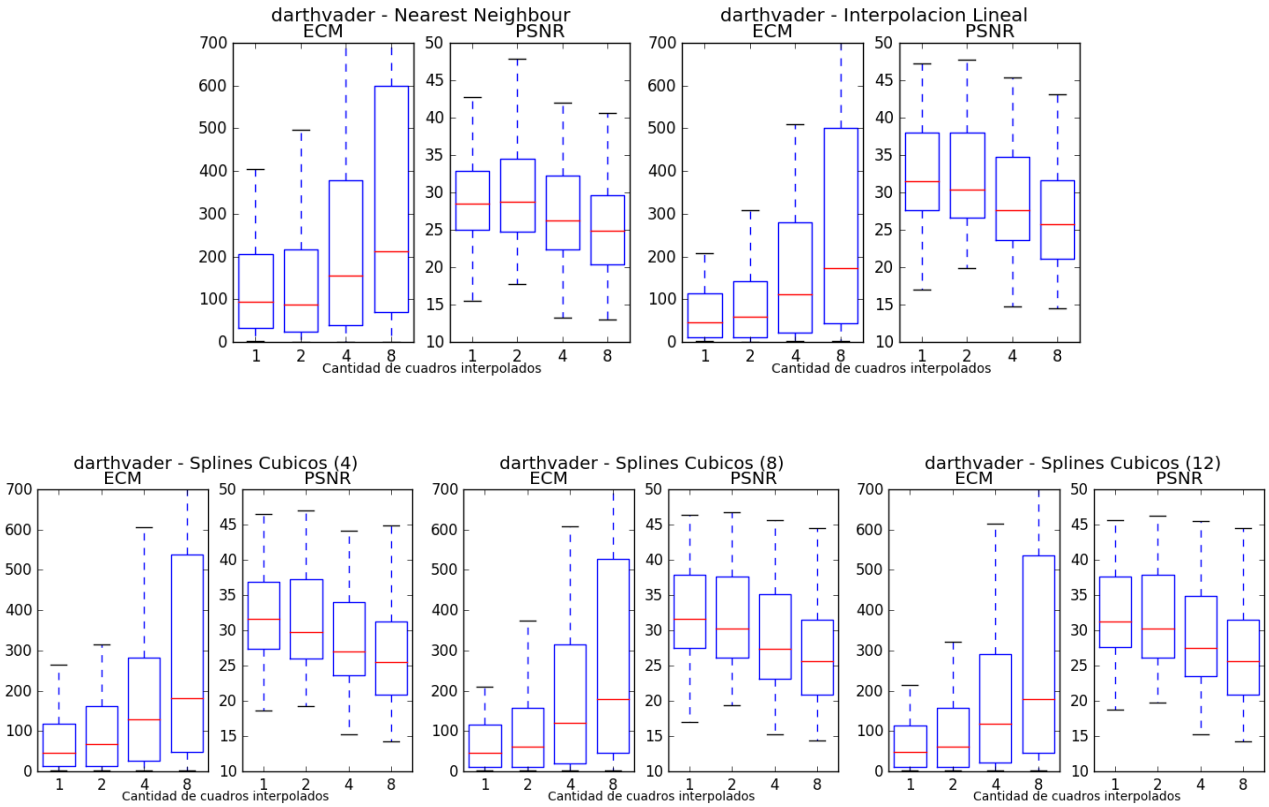

Figura 1: Resultados del cálculo de ECM y PSNR para cada algoritmo en el video `darthvader`. Para Splines cúbicos se indica además el tamaño de los bloques utilizados.

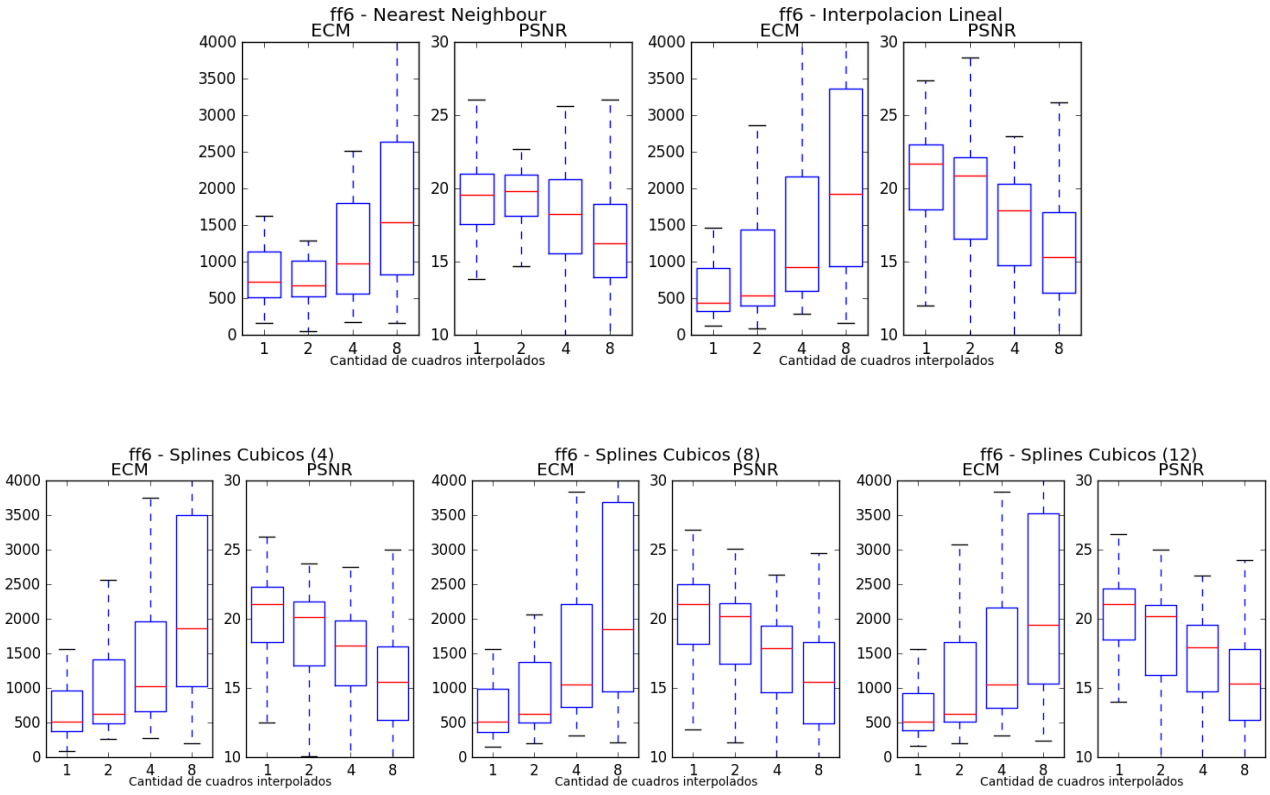

Figura 2: Resultados del cálculo de ECM y PSNR para cada algoritmo en el video `ff6`. Para Splines cúbicos se indica además el tamaño de los bloques utilizados.

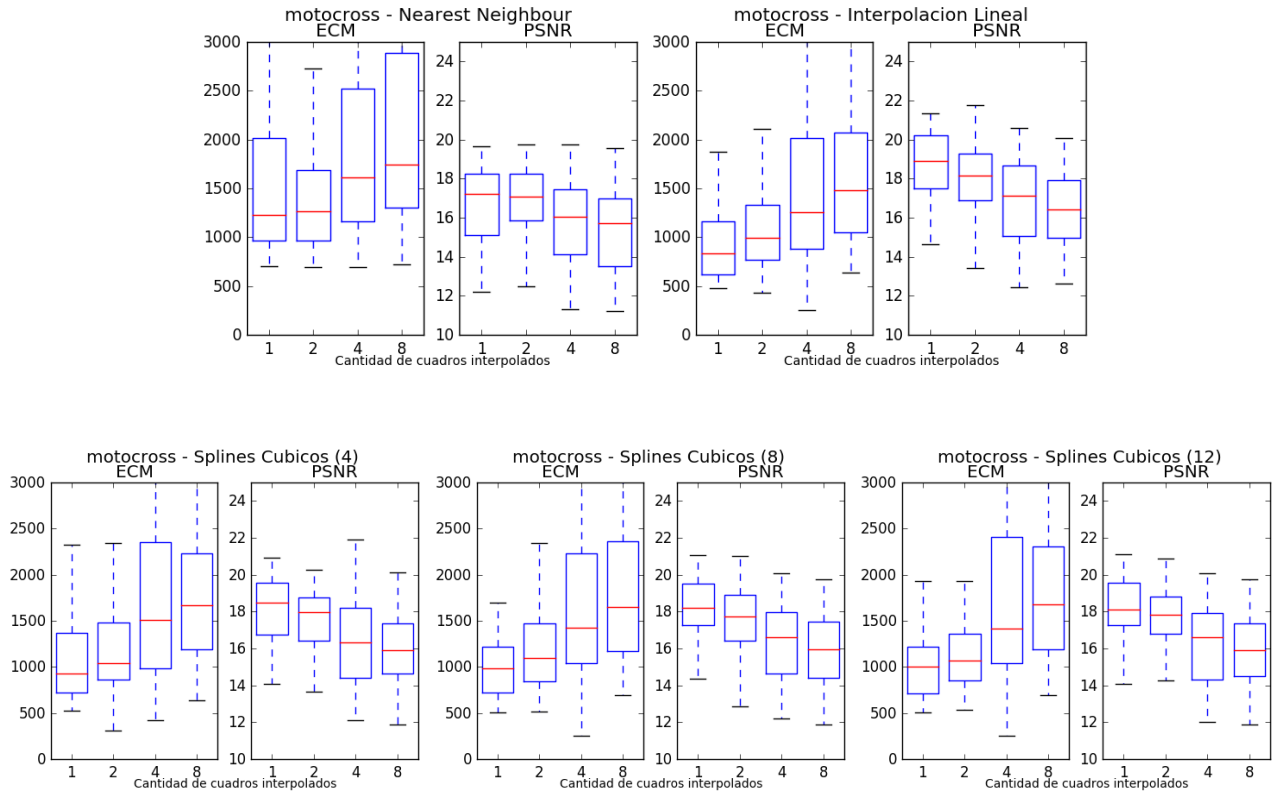

Figura 3: Resultados del cálculo de ECM y PSNR para cada algoritmo en el video `motocross`. Para Splines cúbicos se indica además el tamaño de los bloques utilizados.

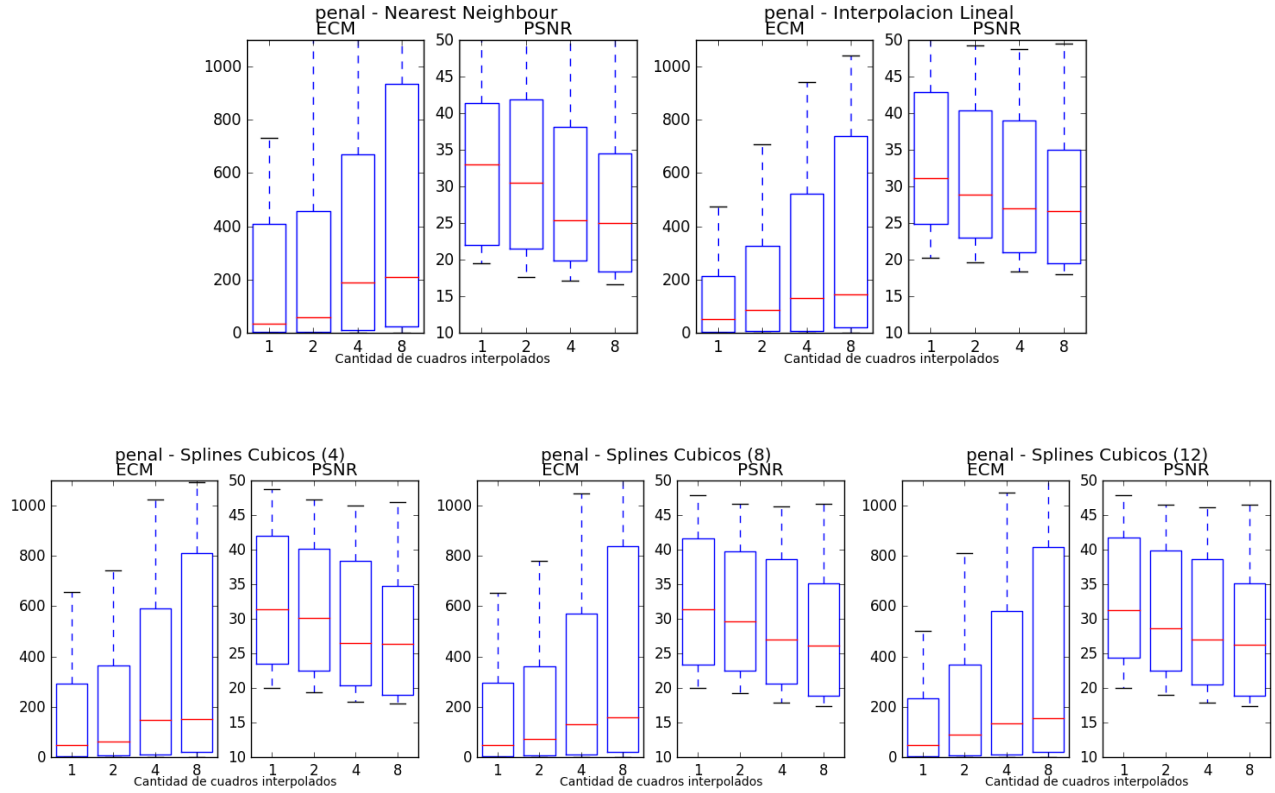

Figura 4: Resultados del cálculo de ECM y PSNR para cada algoritmo en el video `penal`. Para Splines cúbicos se indica además el tamaño de los bloques utilizados.

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 que 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, cosideraremos 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).

3.1.1. `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.

3.1.2. `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 abrutpos 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.

3.1.3. `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 rapidamente, la información nueva altera y distorsiona los resultados.

3.1.4. `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 objetios 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 esta quieta, al interpolar con vecino mas 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, sobrepasando al de interpolación lineal y splines cúbicos, que estan prácticamente empatados.

Explicar esto no es dificil: al tener que interpolar más cuadros, nearest neighbour empieza a aumentar su error dado que repetir un cuadro muchas 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 sobrepasar en rendimiento (con respecto al error) al algoritmo de nearest neighbour cuando hay que interpolar muchos cuadros.

3.1.5. 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.

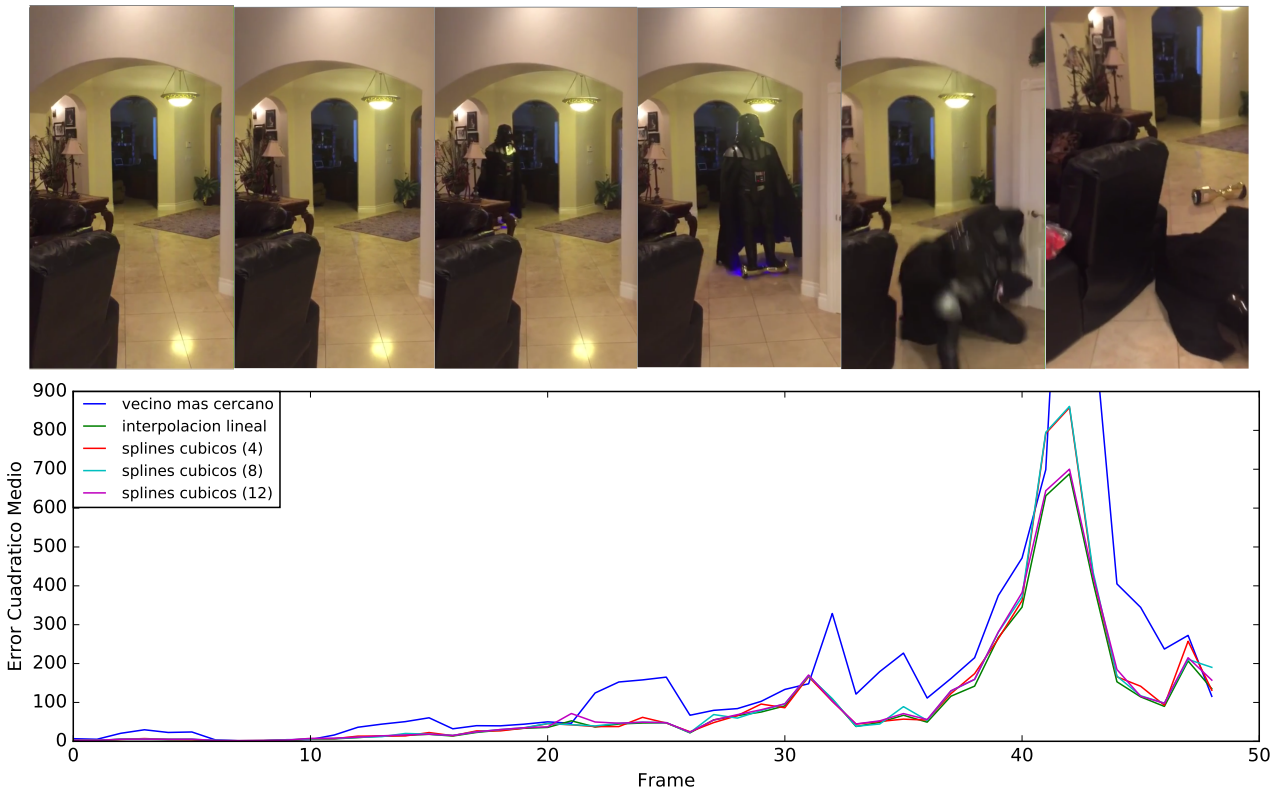

Figura 5: Error Cuadrático Medio a lo largo del tiempo para los distintos algoritmos para el video `darthvader`. Además arriba se puede ver un extracto del video en ese momento.

Como podemos observar en la figura 5, 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 dificil, y aparecen artifacts, dado que nuestros algoritmos no hacen segimiento de objetos, si no que simplemente interpolan pixel a pixel.

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.

4. Conclusiones

5. Apéndices

Referencias

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