Reinventing inter-frame compression using interpolation on pixels

Nathaniel Hamovitz
Math 104A, Atzberger, W23

due 2023-03-20

1 Abstract

We present a method of interframe-video compression based on interpolation techniques. Using `numpy` and OpenCV, we extract every 5th frame, and then calculate linear and cubic polynomials to interpolate between the saved frames. We reconstruct the videos, then calculate absolute and relative error by pixel and color channel, as well as the mean over the length of the video. Linear and cubic splines both perform comparably, yielding mean relative error of less than 0.1 across our testing. Work performed with Rianna Alers and Emma Opper.

2 Motivation

Video compression is a big problem. Lots of video is sent over the internet and sending it uncompressed would require massive amounts of data, so compression techniques are very important!

There are two paradigms of video compression techniques. Inter-frame compression is a method of compression where only some fraction of the video's frames are stored, and the in-between frames are estimated based off the available data. In intra-frame compression, on the other hand, each frame is stored, but image compression (for example run-length encoding techniques) is applied to each individual frame. Modern video compression such as MPEG and H.264 utilize both extensively. In this study, we will only consider inter-frame compression.

3 Problem Statement

Working from a source video, we will construct interpolating linear and natural cubic splines between every fifth frame to estimate the red, green, and blue values of each pixel in the intermediate frames.

4 Methods

We decided to implement a naive inter-frame video compression algorithm based on spline interpolation techniques we learned in class.

Spline interpolation can be expected to work at least passably-well for videos of natural phenomena because in that case, any motion will be continuous. If we were trying to interpolate through a scene cut, on the other hand, there is no reason to expect that the colors before the cut would be at all related to those from after, and interpolation would not work well. (Note that this is a limitation of any inter-frame compression technique.) For videos with continuous motion, though, there is some notion of smoothness in the colors

of the pixels. This satisfies the condition of the Weierstrass approximation theorem, which says that any continuous function can be approximated arbitrarily well by a polynomial. While our original functions are not continuous (in fact, each is a mapping from frame number → color value, which is piecewise discrete), the continuity of the motion displayed gives a smoothness to the data which means we should expect spline interpolation to work reasonably well.

We implemented our algorithm in Python, using the open-source libraries `numpy` and OpenCV. After reading in the video using Python bindings to the OpenCV image manipulation library, we kept every 5 frames (indices 0, 5, 10, etc) and discarded the rest.

Our method is generic over this ratio but we found that keeping every 5 afforded a good balance between space-saving (every 5 frames is 80% file size reduction) and the quality of the reconstructed video. Video data is stored as a four-dimensional array of integers, with size: frame_count x row x column x 3 (the final axis always being three long, to store red, blue, and green color values).

We constructed linear and cubic splines for each pairing of pixel and color channel; for the 240x424 pixel video of keys we used for most of our testing, this amounted to 305,280 piecewise polynomials. These splines are created by interpreting frame index as x-value and color (from 0 to 255) as y-value, and use the values from the frames we kept as their nodes. Hence our reconstructed video matches the original exactly at those frames. Then we used those splines to estimate the colors of each pixel in the frames we had previously discarded (frames index 1, 2, 3, 4, 6, etc), and displayed this new video at the same frame rate as the original.

5 Results

Figure 1: Error for reconstructions of `keys.mp4`

	linear spline	cubic spline
Mean absolute error	3.40422	3.92998
Mean relative error (255)	0.01335	0.01541
Mean relative error (true val)	0.03289	0.03594

Figure 2: Error for reconstructions of `surfing1.gif`

	linear spline	cubic spline
Mean absolute error	13.97354	14.85026
Mean relative error (255)	0.054798	0.058236
Mean relative error (true val)	n/a	n/a

Both methods yield videos where the motion is recognizable, although there are noticable stutters and visual artifacts. See Figures 3 and 4 for screenshots of the original video, the reconstruction with a linear spline and the pixel-wise error, and the reconstruction with a cubic spline and pixel-wise error. Full videos are available in this Google Drive folder.

We computed the elementwise absolute error between the 4D matrices representing the original video 'keys.mp4' and the videos reconstructed using cubic and linear splines. This absolute error array can itself be interpreted as video data and played back; then dark areas represent low error and bright areas represent high error. After taking the elementwise absolute value, we compute the mean across the entire array.

Figure 3: Screenshots from processing of `keys.mp4`, with every 5 frames kept

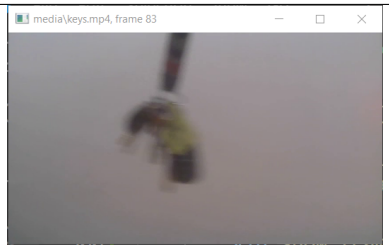
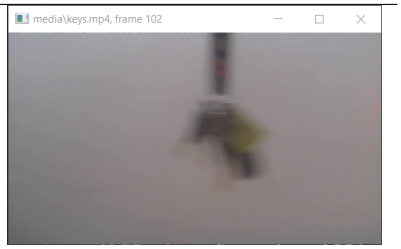
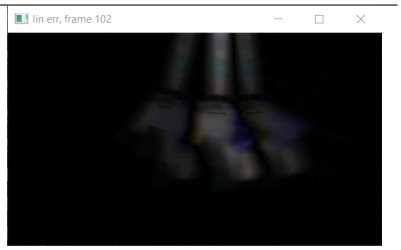
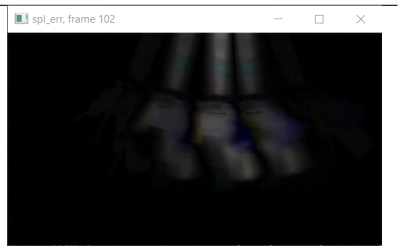
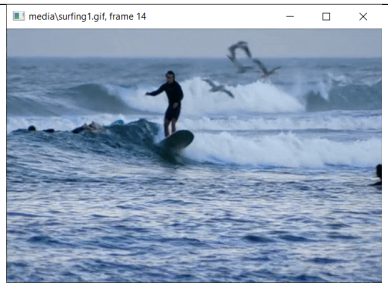
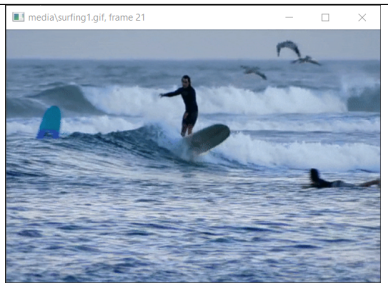
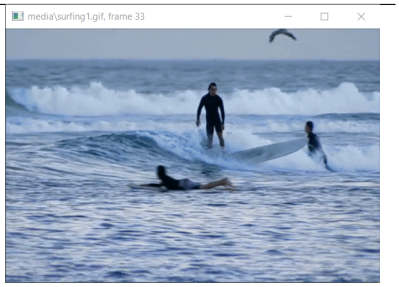
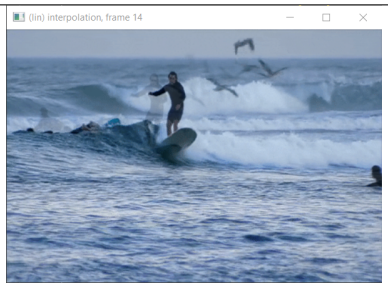
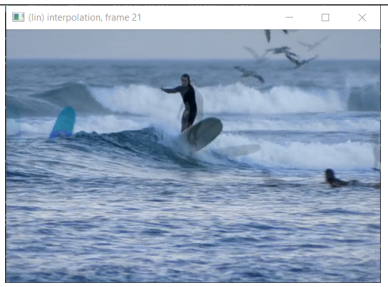
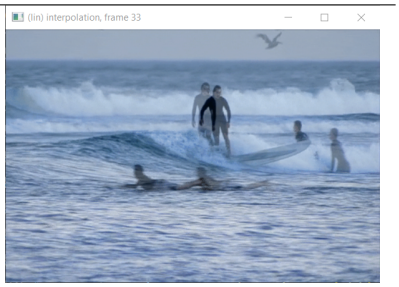
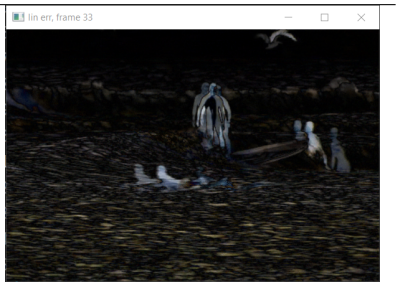
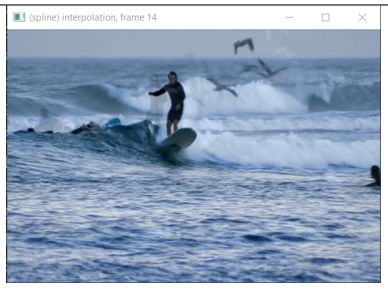
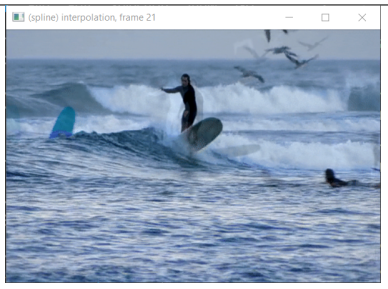
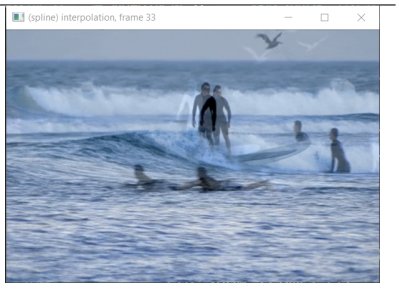
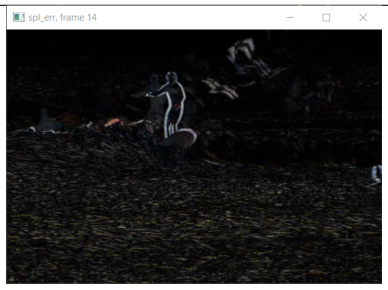
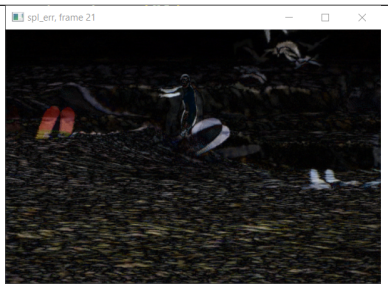
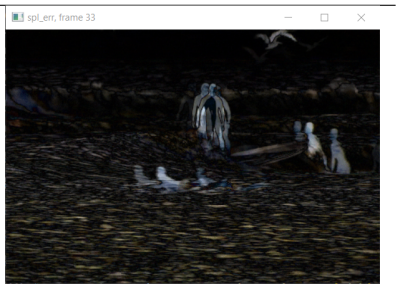
	Frame #21	Frame #83	Frame #102
Video			
Linear spline			
Linear error			
Cubic spline			
Cubic error			

Figure 4: Screenshots from processing of `surfing1.gif`, with every 5 frames kept

	Frame #14	Frame #21	Frame #33
Video			
Linear spline			
Linear error			
Cubic spline			
Cubic error			

We additionally calculate the elementwise relative error, and the mean across the entire array. Relative error is normally defined as division of absolute error by true value, but in this setting it is quite possible for the true value to be zero, rendering this definition unworkable. Another possible divisor presents itself naturally: as the highest possible value is 255, we compute another "relative error" as absolute error divided by 255. Mean error values as computed using two different test videos are shown in Figures 1 and 2 (`keys.mp4` and `surfing1.gif`, respectively).

Theoretical error bounds for natural cubic splines are fourth-order in the distance between nodes, but require C^4 continuity of the underlying function (Burden et al, 10e, 157), which we clearly cannot rely on here.