Interpolation For FPS (Frame Per Second) Modification

Mochammad Fariz Rifqi Rizqulloh 13523069[1,2]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13523069@itb.ac.id, [2]faris361707@gmail.com

***Abstract*—Frame rate, commonly known as frames per second (FPS), plays a pivotal role in how audiences perceive videos and animations. Low FPS can result in a choppy, unnatural experience, while high FPS provides fluidity and a more immersive viewing. In this paper, we explore FPS modification through interpolation methods, comparing linear interpolation, cubic Hermite spline interpolation, polynomial interpolation, and FFmpeg's minterpolate filter. Each method brings distinct trade-offs between computational efficiency and visual quality. The extracted frames from the original video are interpolated using these techniques to generate intermediate frames, achieving the desired FPS. While linear interpolation is computationally efficient, it struggles with smooth transitions. Cubic Hermite splines balance efficiency and continuity, whereas polynomial interpolation offers theoretical accuracy but becomes impractical due to high computational demands. FFmpeg's minterpolate provides superior visual quality but is computationally expensive. This paper offers a comprehensive comparison of these methods in terms of speed and output quality, providing insights into their applicability across various scenarios in video processing. By addressing computational challenges and visual fidelity, this research contributes to advancing video frame rate modification methods.**

***Keywords*—Frame rate modification, interpolation, video processing, frame synthesis, computational efficiency**

I. Introduction

Frame per second, or FPS, is the rate at which consecutive images, called frames, appear on a screen. This simple number can make a video feel smooth and natural or jagged and uncomfortable. Imagine watching a movie or playing a game where motion stutters—it pulls you out of the experience, doesn't it? That's what happens with low FPS. High FPS, on the other hand, creates fluidity, especially in fast-moving scenes. But there's a balance to strike. Too low, and the experience is choppy. Too high, and you're pushing hardware limits for little gain.

Adjusting FPS, known as FPS modification, becomes essential in ensuring a seamless viewing experience across different platforms and devices. Whether it's adapting a cinematic 24 FPS film for a 60 FPS display or breathing life into old, jerky footage, the goal remains consistent: smooth transitions that feel natural. In fields like gaming, sports broadcasting, and video streaming, this adjustment is more than a technical task—it shapes how audiences connect with visual content.

This project explores ways to tackle FPS modification. Specifically, it compares four methods: linear interpolation, polynomial interpolation, cubic spline interpolation. Each brings a unique approach while maintaining usage of same concept, Interpolation. Linear interpolation is fast and straightforward but can leave motion looking unnatural. Cubic spline enhances continuity across frames. These techniques are not just tools; they offer choices that balance speed, quality, and computational needs. For FPS modification on professional fields, like movie production, they use advanced methods like optical flow, or using other library, like FFmpeg, or with the help of AI (Artificial Inteligence). That might provide superior quality but those are computationally expensive.

Why does this matter? Videos today need to work seamlessly across devices, whether they're old movies restored for modern screens or fast-paced games with intricate graphics. Choosing the right technique for FPS modification affects how well the final product works for its audience. This project evaluates these methods to offer insights into their effectiveness and trade-offs, providing guidance for making informed decisions.

In this paper, we'll first set the stage by looking at existing work and techniques for FPS modification. Then, we'll describe how each of the four methods is implemented and analyzed. Finally, results will show how these techniques perform in practice, and we'll draw conclusions about their suitability for different types of content. Every frame matters, and this project digs into how to make them count.

.

II. Fundamental Theorem

A. Interpolation

Interpolation is a mathematical technique used to estimate the value of a function at a point within the range of a discrete set of known data points. Simply put, given a set of known point : $(x_1, y_1), (x_2, y_2), (x_3, y_3), \dots, (x_n, y_n)$, interpolation constructs a continuous function $f(x)$ that passes through these points.

For example, given these points : $(0,2), (1,4), (2,0)$,

$(3,4)$. The interpolation result may vary (there may exist more than 1 equation that satisfy the condition), for example one of the equation that satisfy the condition is $f(x) = 2 - 5x + 6x^2 - x^3$. This equation indeed satisfy the condition, as $f(0) = 2, f(1) = 4, f(2) = 0, f(3) = 4$.

B. Linear Interpolation

Linear interpolation is one of the simplest methods used to estimate unknown values between two known data points. Linear interpolation is a method of curve fitting using linear polynomials to construct new data points within the range of a discrete set of known data points. Suppose known data points is (x_1, y_1) and (x_2, y_2), then equation is $f(x) = y_1 + (\frac{x-x_1}{x_2-x_1})(y_2 - y_1)$

Suppose we have coordinate of points : $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ such that $x_1 < x_2 < \cdots < x_n$. Then the result of interpolation is a piece-wise function for each interval between two consecutive points. This is example of interpolation using linear interpolation on points $(-1,10)$, $(0,3)$, $(1,0), (2,5), (3,7), (4,7)$.

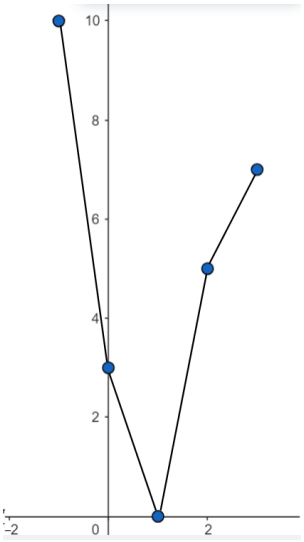

Fig 2.1 : Result of linear interpolation
Source : writer's archive

This algorithm provides very simple and very efficient approach to interpolating unknown values, as its does not need any complex processing. However, its come with a price : the result of linear interpolation may not accurate for complex data.

C. Cubic Hermite Spline Interpolation

Cubic hermite spline interpolation is a method used to estimate unknown values between two known points by fitting a cubic polynomial to the data. Unlike linear interpolation, which connects data points with straight lines, cubic hermite spline interpolation uses polynomials of degree three, allowing for smoother and more accurate approximations, especially when dealing with nonlinear data. Suppose known data points is $(x_0, f(x_0))$, $(x_1, f(x_1))$, $(x_2, f(x_2))$, …, $(x_n, f(x_n))$, where $x_0 < x_1 < x_2 < \cdots < x_n$. To evaluate data in arbitrary x, we will use information from 4 known data points (i, j, k, l), where $x_i \leq x_j < x < x_k \leq x_l$. In other words, point j is nearest point from left, point i is next nearest point from right point k is nearest point from right, and point l is next nearest point from right. After interpolation, we will be able to evaluate any points in interval $[x_j, x_k]$. i-th point and l-th points is take into consideration, because the idea of cubic hermite spline interpolation is we want to take into consideration of the slope at x_j and slope at x_k.

Consider coordinate of points $(n-1, f(n-1))$, $(n, f(n)), (n+1, f(n+1)), (n+2, f(n+2))$, where n is an integer. In addition, assume that the tangents at the endpoints are defined as the centered differences of the adjacent points, that its:

$$m_n = \frac{f(n-1) + f(n+1)}{2}$$
$$m_{n+1} = \frac{f(n) + f(n+2)}{2}$$

To evaluate the interpolated $f(x)$ for a real x, first separate x into the integer portion n and fractional portion u:

$$x = n + u$$
$$n = \lfloor x \rfloor$$
$$u = x - n = x - \lfloor x \rfloor$$
$$0 \leq u < 1$$

Then the Catmull-Rom spline is :

$$f(x) = f(n+u) = \mathrm{CINT}_u\big(f(n-1), f(n), f(n+1), f(n+2)\big)$$

$$= \begin{bmatrix} 1 & u & u^2 & u^3 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 \\ -\frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 1 & -\frac{5}{2} & 2 & -\frac{1}{2} \\ -\frac{1}{2} & \frac{3}{2} & -\frac{3}{2} & \frac{1}{2} \end{bmatrix} \begin{bmatrix} f(n-1) \\ f(n) \\ f(n+1) \\ f(n+2) \end{bmatrix}$$

$$= \frac{1}{2} \begin{bmatrix} -u^3 + 2u^2 - u \\ 3u^3 - 5u^2 + 2 \\ -3u^3 + 4u^2 + u \\ u^3 - u^2 \end{bmatrix}^T \begin{bmatrix} f(n-1) \\ f(n) \\ f(n+1) \\ f(n+2) \end{bmatrix}$$

In this case, for that interval $[n, n+1]$, the cubic polynomial formed is $f(x) =$

$$f(n) \times u^0 +$$
$$\big(-0.5f(n-1) + 0.5f(n+1)\big) \times u^1 +$$
$$(f(n-1) - 2.5f(n) + 2f(n+1) - 0.5f(n+2)) \times u^2 +$$
$$(-0.5f(n-1) + 1.5f(n) - 1.5f(n+1) + 0.5f(n+2)) \times u^3$$

Substitute $u = (x - n)$, we get $f(x) =$

$$f(n) \times (x-n)^0 +$$
$$\big(-0.5f(n-1) + 0.5f(n+1)\big) \times (x-n)^1 +$$
$$(f(n-1) - 2.5f(n) + 2f(n+1) - 0.5f(n+2)) \times (x-n)^2 +$$
$$(-0.5f(n-1) + 1.5f(n) - 1.5f(n+1) + 0.5f(n+2)) \times (x-n)^3$$

For example, if I have coordinates of points : $(-1,10)$, $(0,3), (1,0), (2,5), (3,7), (4,7)$. Then the result of the

interpolation is :

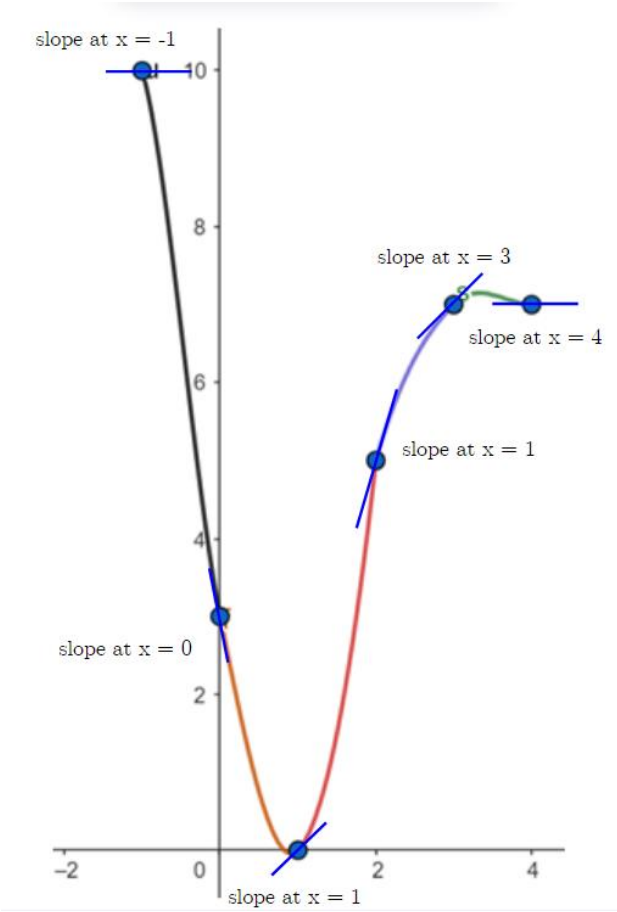

Fig 2.2 : Result of cubic hermite spline interpolation
Source : writer's archive

In that example, piece-wise equation formed is

$$P(x) = \begin{cases} 10 - 3.5(x+1) - 9.0(x+1)^2 + 5.5(x+1)^3, -1 \le x \le 0 \\ 3 - 5x + 0x^2 + 2x^3, 0 < x \le 1 \\ (x-1) + 9.5(x-1)^2 - 5.5(x-1)^3, 1 < x \le 2 \\ 5 + 3.5(x-2) - 2.0(x-2)^2 + 0.5(x-2)^3, 2 < x \le 3 \\ 7 + 1.0(x-3) - 2.0(x-3)^2 + 1.0(x-3)^3, 3 < x \le 4 \end{cases}$$

D. Cubic Spline Interpolation

Cubic spline interpolation and cubic hermite spline interpolation has very similar main idea and approach to interpolation. Both approaches using piece-wise third order polynomial, both take consideration on polynomial continuity, and both also take consideration on the polynomial slope. However, cubic hermite spline interpolation is done under one assumption the slope of a points is average of its two adjacent points :

$$m_n = \frac{f(n-1) + f(n+1)}{2}$$

That assumption make the calculations and interpolation's process much simpler. Meanwhile, the natural cubic spline interpolation work with no assumption, but it has one more consederation, the second derivative.

Consider a set of points $(x_0, f(x_0))$, $(x_1, f(x_1))$, $\dots, (x_n, f(x_n))$, where $x_0 < x_1 < \cdots < x_n$. Let P_i is the piece wise function of $f(x)$ for interval $[x_i, x_{i+1}]$. Then for all i where $0 \le i < n$ must follow these conditions :

$$P_i(x_i) = f(x_i) \dots (1)$$
$$P_i(x_{i+1}) = P_{i+1}(x_{i+1}) \dots (2)$$
$$P_i'(x_{i+1}) = P_{i+1}'(x_{i+1}) \dots (3)$$
$$P_i''(x_{i+1}) = P_{i+1}''(x_{i+1}) \dots (4)$$

Since $P_i(x)$ is a cubic polynomial, then $P_i''(x)$ is a linear polynomial. Let that value of $P_i''(x_i) = \kappa_i$. Since $P_i''(x)$ is a linear polynomial, we can easily found the value of $P_i''(x)$ where $x_i \le x \le x_{i+1}$.

$$P_i''(x) = \kappa_i \frac{x_{i+1} - x}{x_{i+1} - x_i} + \kappa_{i+1} \frac{x - x_i}{x_{i+1} - x_i} \dots (5)$$

Integrating (1), we will get the value of $P_i'(x)$.

$$P_i'(x) = -\frac{\kappa_i}{2} \frac{(x_{i+1} - x)^2}{x_{i+1} - x_i} + \frac{\kappa_{i+1}}{2} \frac{(x - x_i)^2}{x_{i+1} - x_i} + \alpha_i \dots (6)$$

Integrating (2), we will get value of $P(x)$.

$$P_i(x) = -\frac{\kappa_i}{6} \frac{(x_{i+1} - x)^3}{x_{i+1} - x_i} + \frac{\kappa_{i+1}}{6} \frac{(x - x_i)^3}{x_{i+1} - x_i} + \alpha_i x + \beta_i \dots (7)$$

We can use this information into our initial condition (1), (2), (3), and (4). In the end, we will get this equation :

$$\kappa_i(x_{i+1} - x_i) + 2\kappa_{i+1}(x_{i+2} - x_i) + \kappa_{i+2}(x_{i+2} - x_{i+1}) = 6\left[\frac{f(x_{i+2}) - f(x_{i+1})}{x_{i+2} - x_{i+1}} - \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}\right]$$

Where $i = 0, 1, 2, .., N-2$

We finally get $N-1$ linear equations with $N+1$ unknowns : κ_i, where $0 \le i \le N$. We can use any system of linear solver, such as *Gaussian Elimnation* to get the value of the unknowns. After we find the value of unknown variables, κ_i where, $0 \le i \le N$ we finally get the piece-wise function $P_i(x)$, as the coefficient of $P_i(x)$ dependant only on value of κ_i and κ_{i+1}. However, as the n number of unknown variables is higher than the number of equation, the result of linear system of linear equation might be a parametric variables. Some way to address this is using some assumptions. Those assumptions including, but not limited to :

- Make the curvature zero at the endpoints x_0 and x_n. That is, make $\kappa_0 = \kappa_n = 0$. This assumption is equivalent to assuming that $P_0(x)$ and $P_{N-1}(x)$ approach linearity at their outer extremities.
- Make the slope of $f(x)$ have a specified value at either of the boundaries.
- Make $\kappa_0 = \kappa_1$, and $\kappa_{n-1} = \kappa_n$. This assumption is

equivalent to assuming that $P_0(x)$ and $P_{N-1}(x)$ approach parabola at their outer extremities

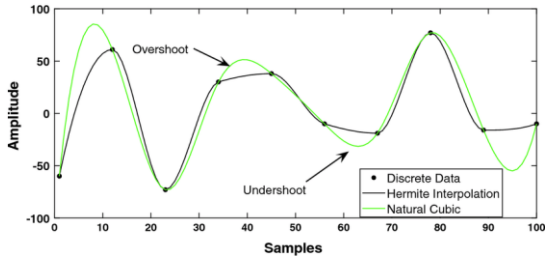

Fig 2.3. Comparison between natural cubic and hermite interpolation.
Source : [4]

E. Polynomial Interpolation

Polynomial interpolation is a method used to estimate values between known data points by fitting a polynomial that passes through these points. Given a set of data points, the goal is to find a single polynomial of the lowest possible degree that exactly matches each point. On of the key difference between cubic-spline Interpolation, is that for polynomial interpolation, the result is one continuous polynomial, whereas cubic-spline interpolation is continous piece-wise function. In general, given $n+1$ points, (x_0, y_0), (x_1, y_1), (x_2, y_2), …, (x_n, y_n), the interpolation result will be a polynom with degree of n. $f(x) = c_0 + c_1x + c_2x^2 + c_3x^3 + \cdots + c_nx^n$. The interpolation result, $f(x)$, should satisfy these condition :

- $f(x_0) = y_0$

  $c_0 + c_1x_0 + c_2x_0^2 + c_3x_0^3 + \cdots + c_nx_0^n = y_0$
- $f(x_1) = y_1$

  $c_0 + c_1x_1 + c_2x_1^2 + c_3x_1^3 + \cdots + c_nx_1^n = y_1$
- $f(x_2) = y_2$

  $c_0 + c_1x_2 + c_2x_2^2 + c_3x_2^3 + \cdots + c_nx_2^n = y_2$
- …
- $f(x_{n+1}) = y_{n+1}$

  $c_0 + c_1x_{n+1} + c_2x_{n+1}^2 + \cdots + c_nx_{n+1}^n = y_{n+1}$

Now, we need to find $c_i, 0 \le i \le n$ that satisfy all of the equation above. This problem breakdown into this following : "*given* $n+1$ *linear equation, find* $n+1$ *unknown variables*". As you can see, this quickly become a trivial system of linear equation problem. There is a lot of method to solve system of linear equation problem, one of which is using *Gauss Elimination Method.* As we solve all the coefficient, we finally can evaluate value of $f(x)$ at any points.

F. System of Linear Equation

A system of linear equations consists of multiple linear equations involving the same set of variables. For example, a system with n equations and k unknowns can be written in general form as:

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1k}x_k = b_1$$
$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2k}x_k = b_2$$
$$\cdots$$
$$a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nk}x_k = b_n$$

This system of linear equations can also be written as matrix notation:

$$Ax = b$$

Where A is the coefficient matrix, x is vector of unknowns variable, and b is vector of constant. Solving a system of linear equations can be done by various ways, including *Gaussian Elimination, LU Decomposition, Gauss-Jordan Elimination, Jacobi Method, Gauss-Seidel Method* and others.

III. Problem Analysis

Video is a series of frames, and frames is matrix of RGB values. In this paper, we let that $P_{init}(i, x, y)$ is value stored in RGB values of i-th frames of original video, at position (x, y). Similarly, we let that $P_{final}(i, x, y)$ is value stored in RGB values of i-th frames of modified video, at position (x, y). To modify the FPS, we need to generate a new set of pixel values for each frame of the modified video. The number of frames in the modified video may differ, and we must compute the values for intermediate frames.

To modify the FPS of a video, we treat each pixel's position (x, y) independently. This means that the pixel values over time at a specific location (x, y) in all frames form a series of data points that we can process separately from other pixel locations. For a given pixel position (x, y), the data points are the RGB values of that pixel across all frames in the original video. Denoting the original number of frames as N, the set of points we will interpolate is : $\begin{bmatrix} (1, P_{init}(1, x, y)), \\ (2, P_{init}(2, x, y)), \\ \cdots \\ (N, P_{init}(N, x, y)) \end{bmatrix}$.

Using an interpolation method (e.g., linear, cubic hermite spline, or polynomial), we construct a continuous function $F_{x,y}(t)$, where t is a real value for any $t \in [1, N]$. Using $F_{x,y}(t)$, we can construct $P_{final}(i, x, y)$, for all $1 \le i \le N'$, where N' is number of frames in modified video. To get consistent result, we can evalue in this following way :

- Let $\alpha = \frac{N-1}{N'-1}$
- $P_{final}(1, x, y) = F_{x,y}(1)$
- $P_{final}(2, x, y) = F_{x,y}(1 + \alpha)$
- $P_{final}(3, x, y) = F_{x,y}(1 + 2\alpha)$
- …
- $P_{final}(N', x, y) = F_{x,y}(1 + (N' - 1)\alpha)$

Using above method of evaluating make sure the difference between each frame is consistent, make the possible best result among all scenario.

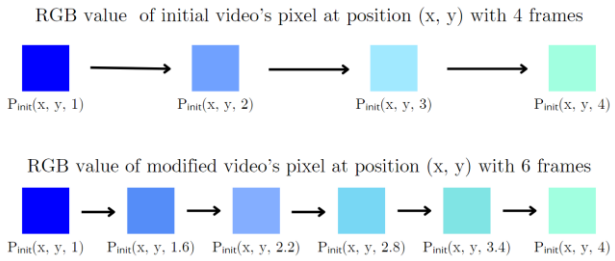

Fig 3.1. Visualization of the process.
Source : writer's archive

Interpolation is done $W \times H$ times, where one for each pixel position. After all pixel is interpolated and the value is evaluated, we can turn all the frame back into video again. Below is experiment done on video with 4 frames, changed to video with 7 frames.

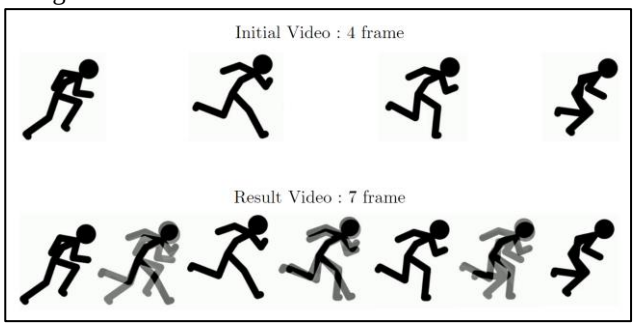

Fig 3.2. Result of FPS modification from 4 frame to 7 frame. Both of the video has the same duration, but different number of frames.
Source : writer's archive

Polynomial interpolation requires special handling due to its computational challenges. As explained earlier, we will treat each pixel position (x, y), where $1 \leq x \leq W$ and $1 \leq y \leq H$ independently. Because of that, for each pixel position, the number of data points processed for interpolation will be equal to the number of frames in the video. As the duration or FPS of the original video increases, the total number of frames also grows significantly. This becomes a problem because, in polynomial interpolation, the degree of the polynomial is equal to the number of frames minus one. For example, if we have a 15-second video with an FPS of 24, the degree of the polynomial would be:

$$n = (15 \times 24) - 1 = 359$$

Such a high degree creates serious challenges for computational speed and memory. The coefficients of the system of linear equations used to compute the polynomial grow extremely large, leading to inefficiencies.

To address this issue, I chose to split the video into smaller chunks, each consisting of 25 frames. This approach ensures that the interpolation remains accurate while significantly reducing the computational load and memory requirements. By limiting the degree of the polynomial within each chunk, the process becomes manageable and avoids excessive computation time.

IV. Implementation

In implementing FPS modification, we utilized two primary libraries: NumPy and OpenCV. OpenCV was used to handle the video processing tasks, specifically for extracting frames from the input video. By reading the video file frame by frame, OpenCV allowed us to efficiently access the raw pixel data of each frame.

The extracted frames were stored in a Python list, with each frame represented as a 3D NumPy array of shape $(height, width, 3)$, where the three channels correspond to the RGB values of each pixel. This structure ensured that the frames were organized sequentially, enabling straightforward manipulation and interpolation of pixel values. The list as a whole had a shape of $(frames, width, height, 3)$, representing the entire video as a collection of RGB matrices.

Once the interpolation process was applied, the resulting interpolated frames were computed and appended to a new list, representing the modified video. Each interpolated frame maintained the same dimensions and RGB structure as the original frames, ensuring consistency. The final list of frames for the modified video was then saved back as a new video file using OpenCV, completing the FPS modification process.

```
frames = []
cap = cv2.VideoCapture(video_path)
# Read all frames from the video
while cap.isOpened():
    ret, frame = cap.read()
    if not ret:
        break
    frames.append(frame)
cap.release()
```

Fig 4.1. Converting video into list of frames
Source : writer's archive

```
# Perform cubic interpolation for FPS modification
new_frames = []
new_frames_count = int(fps_after / fps_before * len(frames))
diff = (len(frames) - 1) / (new_frames_count - 1)

print(len(frames), new_frames_count)

for i in range(new_frames_count):
    # Calculate the interpolated index
    coord = i * diff
    lower_idx = int(np.floor(coord))
    upper_idx = min(lower_idx + 1, len(frames) - 1)
    next_upper_idx = min(lower_idx + 2, len(frames) - 1)
    prev_lower_idx = max(lower_idx - 1, 0)

    # Calculate interpolation weights
    alpha = coord - lower_idx

    # Fetch frames for cubic interpolation
    p0 = frames[prev_lower_idx].astype(np.float32)
    p1 = frames[lower_idx].astype(np.float32)
    p2 = frames[upper_idx].astype(np.float32)
    p3 = frames[next_upper_idx].astype(np.float32)

    # Perform cubic interpolation for each channel
    def cubic_hermite(t, p0, p1, p2, p3):
        return (
            (-0.5 * t + t ** 2 - 0.5 * t ** 3) * p0 +
            (1 - 2.5 * t ** 2 + 1.5 * t ** 3) * p1 +
            (0.5 * t + 2 * t ** 2 - 1.5 * t ** 3) * p2 +
            (-0.5 * t ** 2 + 0.5 * t ** 3) * p3
        )

    interpolated_frame = cubic_hermite(alpha, p0, p1, p2, p3)
    interpolated_frame = np.clip(interpolated_frame, 0, 255).astype(np.uint8)
    new_frames.append(interpolated_frame)
```

Fig 4.2. Interpolation process
Source : writer's archive

```python
def frames_to_video(frames, fps, output_path):
    height, width, layers = frames[0].shape
    fourcc = cv2.VideoWriter_fourcc(*'mp4v')
    out = cv2.VideoWriter(output_path, fourcc, fps, (width, height))

    for frame in frames:
        out.write(frame)

    out.release()
```

Fig 4.3. Converting list of frames into video
Source : writer's archive

V. Result Analysis

The speed comparison highlights the computational efficiency and result quality of different FPS modification methods for videos of varying resolutions and frame counts. Aside of the linear interpolation, cubic hermite spline interpolation, and polynomial interpolation, one more method is added into the comparison. Industrial grade FPS modification library, FFmpeg, is used to measure how efficient is this algorithm. Test is conducted with two kinds of video, simple video, and complex video. The result is shown in the following :

Simple video, consist of 48 frames, resolution is 40×100. FPS of the video is modified to 5 times fold, from 12 to 60,. The final video consist of 240 frames:

1. Linear Interpolation : 0.06 seconds
2. Cubic hermite spline interpolation : 0.07 seconds
3. FFmpeg's *minterpolate* : 0.5 seconds
4. Polynomial Interpolation : 6s

Complex video, consist of 178 frames, resolution is 720×1280. FPS of the video is modified to 2.5 times fold, from 12 to 30. The final video consist of 445 frames:

1. Linear Interpolation : 22.34 seconds
2. Cubic hermite spline interpolation : 24.11 seconds
3. FFmpeg's *minterpolate* : approximately 70 seconds
4. Polynomial Interpolation : More than 300 seconds, the computation is terminated mid-way due to excessive computational demands

This comparison show how efficient linear interpolation and cubic hermite spline interpolation, while at the same time show how inefficient polynomial interpolation is. As for FFmpeg, eventough it is significantly slower than two other interpolation method, FFmpeg's *mininterpolate* provides very high-quality motion smoothing.

For the result comparison, the result of linear interpolation and cubic hermite spline interpolation far too similar. So for this comparison, only show comparison between original video, cubic hermite spline interpolation, and FFmpeg's *mininterpolate*. When comparing the results of FPS modification using FFmpeg's *minterpolate* filter and cubic hermite spline interpolation, both methods effectively achieved the desired change in FPS, but with noticeable differences in visual output. The FFmpeg approach produced a smoother video overall, but some frames appeared deformed, likely due to motion estimation inaccuracies during interpolation. On the other hand, the bicubic hermite spline interpolation method maintained the structural integrity of all frames, with no visible deformation. However, the resulting video exhibited minor changes to the human eye, with a slight shadow or blur effect, particularly in areas with rapid motion. This suggests that while FFmpeg prioritizes smooth transitions, it may compromise frame quality in some cases, whereas bicubic hermite spline interpolation focuses on maintaining frame integrity, sometimes at the cost of a perfectly smooth experience. Below is some of the comparison. For the full comparison, you may access on this link : [Full Comparison]

Fig 5.1. Example of comparison for complex colored video
Source : writer's archive

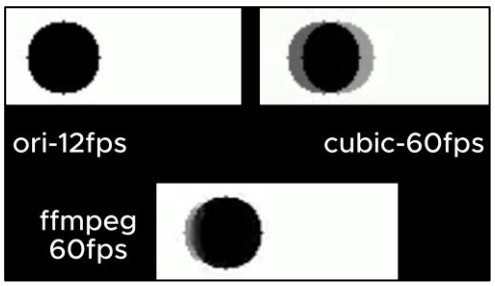

Fig 5.2. Example of comparison for simple colored video
Source : writer's archive

As shown in the figure above, the cubic hermite spline interpolation method works by "squeezing" additional frames between the original frames. As a result, the interpolated frames appear as a blend or merge of the previous and next frames, creating a smooth transition effect. However, in human eyes, the result of cubic hermite spline interpolation may appear worse.

VI. Conclusion

This research demonstrates how interpolation techniques can effectively modify a video's FPS, with each method offering unique benefits and limitations. Linear interpolation is the fastest approach but lacks smooth transitions, making it suitable for simple applications. Cubic Hermite spline interpolation provides a balance between computational efficiency and quality, producing smooth transitions without significant deformation in frames. Polynomial interpolation, although accurate in theory, is computationally prohibitive and unsuitable for practical applications. Meanwhile, FFmpeg's *minterpolate* stands out for its high-quality motion smoothing but at a significant computational cost.

The experiments reveal that cubic Hermite spline interpolation is an optimal middle ground for most scenarios, delivering consistent results with reasonable computational requirements. However, FFmpeg's approach excels in scenarios demanding the highest visual fidelity, particularly in professional video editing and restoration. The findings underscore the importance of selecting an interpolation method that aligns with the

desired trade-off between speed and quality, highlighting the versatility and limitations of these techniques. Future work could explore hybrid methods or advanced machine learning models to improve computational efficiency while maintaining high-quality outputs.

VII. ATTACHMENT

Link for github, explanation video, and google drive is stored in this link : https://linktr.ee/Ryzz17

REFERENCES

[1] S. A. Dyer and J. S. Dyer, "Cubic-spline interpolation. 1," in IEEE Instrumentation & Measurement Magazine, vol. 4, no. 1, pp. 44-46, March 2001, doi: 10.1109/5289.911175.

[2] Cristian Constantin Lalescu, Two hierarchies of spline interpolations. Practical algorithms for multivariate higher order splines, doi: 10.48550/arXiv.0905.3564

[3] Munir, Rinaldi."http://informatika.stei.itb.ac.id/~rinaldi.munir/

[4] Gibin Chacko George, et al, A Novel and Efficient Hardware Accelerator Architecture for Signal Normalization, 25 September 2019, doi:10.1007/s00034-019-01262-3

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 27 Desember 2024

Mochammad Fariz Rifqi Rizqulloh 13523069