

پایان نامه پروژه کارشناسی

Video motion magnification بزرگنمایی حرکت در ویدیو

عرفان خیراللهی - شهریور ۱۴۰۰

دوربین‌ها می‌توانند بسیاری از حرکت‌هایی که از چشم ما پنهان هستند را ثبت کنند. مثال‌هایی از این حرکت‌ها اثر حرکت خون در بدن انسان، نبض و تنفس یک کودک، لرزش‌های ریز در قطعات مکانیکی، لرزش سد آب، لرزش دوربین‌ها موقع عکاسی و لرزش پایه دوربین هستند. می‌توان با پردازش ویدیو و بزرگ‌نمایی این حرکت‌ها، آن‌ها را برای انسان قابل درک کرد. در الگوریتم طراحی‌شده در این پژوهش ابتدا با استفاده از هرم گوسی و لاپلاسی، ویدیو به باندهای فرکانس مکانی مختلف تجزیه و سپس با استفاده از تابع همبستگی، تغییرات در حوزه زمان آشکار می‌شود. در آخر با تقویت سیگنال حاصل و ترکیب آن با ویدیوی اصلی، بزرگ‌نمایی انجام می‌شود. در این پژوهش الگوریتم طراحی‌شده همبستگی با الگوریتم‌های پیشین بر اساس معیار SSIM مورد ارزیابی و مقایسه قرار گرفته است و نتایج نشان می‌دهد الگوریتم طراحی‌شده همبستگی می‌تواند در بزرگ‌نمایی‌های زیر ۲۰۰ برابر تا ۱۲۰ درصد کیفیت بهتری را ارائه کند.

کلمات کلیدی: پردازش تصویر، روش اویلری، تابع همبستگی، بزرگ‌نمایی ویدیو، هرم گوسی، معیار SSIM

Table of Contents

فصل یک مقدمه ۱.....	7
۱.۱ هدف و کاربرد.....	7
۱.۱.۱ هدف این پژوهش.....	7
۱.۱.۲ ضرورت و کاربرد این پروژه.....	7
۱.۱.۲.۱ کاربرد برای بهبود کیفیت.....	8
۱.۱.۲.۲ کاربرد برای پایداری.....	8
۱.۱.۲.۳ کاربرد برای تحقیق‌های علمی.....	9
۱.۲ تعاریف.....	10
۱.۲.۱ تصویر و ویدیوی دیجیتال.....	10
۱.۲.۲ پردازش تصویر.....	10
۱.۲.۳ پردازش ویدیو.....	11
۱.۲.۴ سیگنال در حوزه زمان.....	11
۱.۲.۵ فیلترهای مکانی.....	11
۱.۲.۶ هرم گوسی و هرم لاپلاسی.....	12
۱.۲.۷ فیلتر زمانی.....	12
فصل ۲ مروری بر ادبیات پژوهش ۲.....	13
۲.۱ دسته‌بندی.....	13
۲.۲ کارهای پیشین.....	13
۲.۳ روش‌های پیشنهادی.....	15
فصل ۳ پیاده‌سازی و آزمایش ۳.....	17
۳.۱ الگوریتم دیفرانسیلی.....	18
۳.۱.۱ روش تقویت سیگنال زمانی.....	18
۳.۱.۱.۱ بررسی ریاضی.....	18
۳.۱.۱.۲ توصیف الگوریتم.....	19
۳.۱.۱.۳ فلوچارت این الگوریتم.....	19
۳.۱.۱.۴ آزمایش روش تقویت سیگنال.....	21
۳.۱.۲ روش دیفرانسیلی بزرگنمایی تغییرات در سیگنال حوزه زمان.....	22

۳.۱.۲.۱ تحلیل ریاضی	23
۳.۱.۲.۱.۱ شبیه‌سازی کامپیوتری	23
۳.۱.۲.۲ طراحی الگوریتم دیفرانسیلی برای بزرگنمایی ویدیو	26
۳.۱.۲.۳ توصیف الگوریتم	26
۳.۱.۲.۴ پیاده‌سازی و آزمایش الگوریتم	27
۳.۱.۳ استفاده از فریم‌های قبل‌تر	28
۳.۱.۳.۱ توصیف الگوریتم	28
۳.۱.۳.۲ پیاده‌سازی و آزمایش الگوریتم	29
۳.۱.۴ استفاده از فیلتر گوسی	30
۳.۱.۴.۱ توصیف الگوریتم	30
۳.۱.۴.۲ فلوچارت الگوریتم	31
۳.۱.۴.۳ شبه‌کد الگوریتم روش دیفرانسیلی با فیلتر گوسی	33
۳.۱.۴.۴ پیاده‌سازی و آزمایش الگوریتم	34
۳.۲ الگوریتم اوپلری خطی	36
۳.۲.۱ بررسی کد متلب روش اوپلری خطی	36
۳.۲.۱.۱ تحلیل کد	36
۳.۲.۱.۲ استخراج فلوگراف سیگنال	36
۳.۲.۱.۳ استخراج بلوک دیاگرام	37
۳.۲.۲ الگوریتم استخراج شده	38
۳.۲.۳ پیاده‌سازی الگوریتم اوپلری خطی استخراج شده	39
۳.۳ الگوریتم پیشنهادی: الگوریتم همبستگی	41
۳.۳.۱ بررسی ریاضی	41
۳.۳.۱.۱ DC کسر میانگین فریم‌ها به منظور حذف	42
۳.۳.۱.۲ شکل گسسته در زمان این عبارت ریاضی	43
۳.۳.۲ آشکارسازی حرکت‌های نوسانی در ویدیو	43
۳.۳.۲.۱ توصیف الگوریتم آشکارسازی حرکت نوسانی	43
۳.۳.۲.۲ فلوچارت این الگوریتم	44
۳.۳.۲.۳ پیاده‌سازی و آزمایش الگوریتم آشکارسازی حرکت‌های نوسانی	45

۳.۳.۳ الگوریتم اولیه روش همبستگی برای بزرگنمایی حرکت	46
۳.۳.۳.۱ شکل نوشتاری این الگوریتم	46
۳.۳.۳.۲ پیاده‌سازی و تست این الگوریتم	47
۳.۳.۳.۳ محدودیت‌های روش پیشنهادی	48
۳.۳.۳.۱ از کار افتادن بزرگنمایی در صورت وجود حرکت‌های بزرگ در ویدیو	49
۳.۳.۳.۲ کاهش همبستگی در صورت انتخاب نشدن فرکانس مناسب سیگنال نمونه	50
۳.۳.۴ استفاده از پنجره و بافر برای رفع محدودیت‌های روش پیشنهادی	51
۳.۳.۴.۱ طراحی الگوریتم	51
۳.۳.۴.۲ محاسبه همبستگی در یک پنجره زمانی محدود	52
۳.۳.۴.۲.۱ بهینه‌سازی الگوریتم محاسبه همبستگی روی بافر	53
۳.۳.۴.۳ شبیه‌سازی الگوریتم محاسبه همبستگی در پنجره زمانی محدود	53
۳.۳.۴.۴ محدودیت‌های روش محاسبه همبستگی با پنجره زمانی محدود	57
۳.۳.۴.۴.۱ شبیه‌سازی و بررسی محدودیت‌های روش همبستگی روی پنجره زمانی محدود	57
۳.۳.۴.۵ پیاده‌سازی و تست الگوریتم همبستگی روی پنجره زمانی محدود	63
۳.۳.۴.۶ محدودیت‌های روش بزرگنمایی ویدیو با استفاده از الگوریتم محاسبه همبستگی روی پنجره زمانی محدود	64
۳.۳.۵ استفاده از هرم در روش همبستگی بزرگنمایی ویدیو برای رفع محدودیت نویز در بزرگنمایی	64
۳.۳.۶ الگوریتم نهایی روش همبستگی در بزرگنمایی ویدیو	66
۳.۳.۶.۱ توصیف و شبه‌کد الگوریتم نهایی همبستگی	67
۳.۳.۶.۲ بلوک دیاگرام الگوریتم نهایی همبستگی	68
۳.۳.۶.۳ پیاده‌سازی الگوریتم نهایی همبستگی	68
۳.۴ آزمایش‌ها	69
۳.۵ مقایسه الگوریتم‌ها	69
۳.۵.۱ مقایسه سرعت و حافظه مورد نیاز	69
۳.۵.۲ SSIM مقایسه کیفیت ویدیوی بزرگنمایی‌شده با معیار	72
۳.۵.۳ مقایسه تأخیر شروع و مقاوم بودن در برابر تغییرات شدید	74
۳.۶ بررسی نتایج آزمایش‌ها	75
۳.۷ اثرات نامطلوب ناشی از فشرده‌سازی در حوزه زمان و حوزه مکان	76

۳.۷.۱ بررسی در حوزه زمان.....	77
۳.۷.۲ حوزه مکان.....	77
۴ نتیجه‌گیری و پیشنهادهایی برای ادامه پژوهش.....	78
۴.۱ نتیجه‌گیری.....	79
۴.۲ پیشنهادهایی برای ادامه کار و پژوهش.....	79
۴.۲.۱.۱ حذف فرکانس‌های بالای ویدیو.....	80
۵ پیوست: ویدیوهای آزمایش.....	81
۶ پیوست: کدها.....	83
۶.۱ روش دیفرانسیلی.....	84
۶.۲ روش اوپلری خطی.....	92
۶.۳ روش همبستگی.....	105
۶.۴ نرم‌افزار تست.....	116
۶.۵ محاسبهٔ هرم‌های گوسی و لاپلاسی.....	119

۱ فصل یک مقدمه

یک صفحه کامل

۱.۱ هدف و کاربرد

۱.۱.۱ هدف این پژوهش

حرکت‌های کوچک زیادی در ویدیو وجود دارند که با چشم انسان قابل تشخیص نیستند یا به سختی قابل تشخیص هستند. این حرکت‌های کوچک می‌توانند اطلاعات سودمندی راجع به موجودیت داخل ویدیو برای ما فراهم کنند. تکنیک بزرگنمایی ویدیو می‌تواند این تغییرات کوچک را که با چشم غیر مسلح قابل دیدن نیستند برای ما آشکار کند تا بتوان با مونیتر کامپیوتری یا مشاهده آن‌ها توسط انسان اطلاعات لازم را به دست آورد و اقدام‌های لازم را انجام داد.

در این پژوهش با انجام پردازش‌های دیجیتال لازم روی ویدیوها این حرکت‌های کوچک که تشخیص آن‌ها چشم غیر مسلح سخت یا غیرممکن است^۱ را قابل تشخیص می‌کنیم

در این پایان‌نامه با استفاده از سه روش «اویلری خطی» و «همبستگی» و «دیفرانسیلی» پردازش لازم برای بزرگنمایی ویدیو انجام شده است. و ضمن مقایسه آن‌ها مزایا و محدودیت‌های هر روش بررسی شده است. که روش همبستگی روش ابداعی این پروژه است.

۱.۱.۲ ضرورت و کاربرد این پروژه

به عنوان نمونه، سه دسته از کاربردهای بزرگنمایی ویدیو شامل «بهبود کیفیت»، «پایداری سیستم‌ها» و «تحقیقات علمی» برشمرده می‌شود. به طور کلی بزرگنمایی ویدیو در زمینه‌های پزشکی، صنعت و علوم تجربی کاربردهای فراوانی دارد.

۱.۱.۲.۱ کاربرد برای بهبود کیفیت

برای مونیتور کردن علایم حیاتی بیماران در بیمارستان، مانند تنفس و نبض، در این روش به جای وصل کردن سنسور به دست یا بدن بیماران با استفاده از یک دوربین می‌توان وضعیت‌های یک یا چند نفر را به شکل هم‌زمان پایش کرد.^{۳۲}

برای بررسی ثابت بودن پایه دوربین‌ها برای تولید فیلم، می‌توان با استفاده از یک دوربین و بزرگنمایی ویدیو این کار را انجام داد. به عنوان یک راه می‌توان از دوربین دیگری برای گرفتن ویدیو از پایه دوربین استفاده کرده و تکان‌های آن را بررسی کرد. یا اینکه به عنوان راه جایگزین می‌توان ویدیویی از محلی ثابت از همان دوربین تهیه کرد. وجود تغییرات در این ویدیو نشان دهنده تکان خوردن خود دوربین است.

۱.۱.۲.۲ کاربرد برای پایداری

کاربرد دیگر این روش، بررسی پایداری سیستم‌هایی است که اتصال سنسورهای حرکتی به آن‌ها سخت، پرهزینه یا ناشدنی است.

بزرگنمایی ویدیو در افزایش طول عمر سازه‌های معماری شامل ساختمان‌ها، سدها و غیره کاربرد دارد.^۴

در اتاق‌های سرور، خصوصاً سرورهایی که از هارددیسک برای نگهداری اطلاعات استفاده می‌کنند، لرزش‌های اندک می‌تواند باعث کاهش عمر سیستم‌ها شود. با استفاده از بزرگنمایی حرکت، می‌توان وجود لرزش را در جاهای مختلف اتاق سرور و روی رک‌ها بررسی کرد. لرزش‌های ریز را تشخیص داد و در صورت وجود لرزش، منبع آن را پیدا کرده و مشکل را برطرف کرد.

در سال ۱۴۰۰ یک تانکر در یک پالایشگاه در تهران منفجر شد. بررسی ویدیوی صحنه نشان می‌دهد که ثابت‌نبودن محفظه و تشدید در لرزش باعث ایجاد ترک شده بود و در نهایت منجر به چنین اتفاقی شده بود. چرا که محفظه‌های با فشار سیال بسیار بالا بسیار حساس‌تر هستند و یک حفره کوچک می‌تواند منجر به خروج سیال با سرعت و فشاری بسیار زیاد از محفظه شده و باعث انفجار شود. با روش بزرگنمایی ویدیو، می‌توان با یک دوربین که به شکل ثابت روی یک محل مناسب نصب شده باشد، می‌توان وضعیت لرزش سیستم‌ها را به شکل دائم پایش کرد و قبل از وقوع چنین اتفاق‌هایی، از عامل آن باخبر شده و جلوی آن‌ها را گرفت.

۱.۱.۲.۳ کاربرد برای تحقیق‌های علمی

از بزرگنمایی ویدیو می‌توان در تحقیقات علمی سود برد. برای بررسی حرکت‌های کوچک در جاهایی که امکان استفاده از سنسور حرکتی وجود ندارد یا حرکت‌ها آن‌قدر کوچک هستند که سنسورها توان تشخیص آن‌ها را ندارند.

با این روش می‌توان حرکت‌های کوچک موجودات میکروسکوپی را بررسی کرد. امکان اتصال یک سنسور حرکت در چنین شرایطی وجود ندارد. در حالی که با روش بزرگنمایی ویدیو می‌توان از ویدیوی میکروسکوپی تهیه شده از این‌ها، حرکت‌های آن‌ها را بررسی کرد.

با استفاده از بزرگنمایی ویدیو می‌توان حرکت‌های داخل بدن انسان را با ویدیوهای تهیه شده مشاهده کرد.

مساله دیگر دقت در این روش است. هر چه یک دوربین بتواند بزرگنمایی مکانی بیشتری انجام دهد، با دقت بیشتری می‌توان حرکت‌ها را مشاهده کرد. دامنه بسیاری از حرکت‌ها از آستانه خطای سنسورهای حرکتی بیشتر است. در حالی که امروزه میکروسکوپ‌هایی وجود دارند که می‌توانند تصاویر زنده در ابعاد اتم‌ها را برای ما فراهم کنند. با روش بزرگنمایی ویدیو می‌توان چنین تغییراتی را نیز بررسی کرد.

همچنین در این روش نه تنها احتیاجی به اتصال فیزیکی نیست، بلکه با استفاده از تلسکوپ یا لنزهای تله‌فوتو، بررسی حرکت اجسام دور نیز ممکن است. برای مثال بررسی تغییرات جزئی در اجرام کیهانی. در چنین شرایطی موجودیت مورد نظر آن‌قدر با ما فاصله دارد که استفاده از تصاویر (در فرکانس‌های مختلف طیف امواج الکترومغناطیسی) تنها راه مشاهده آن‌هاست.

۱.۲ تعاریف

در این فصل تعدادی از اصطلاح‌های پایه‌ای به کار رفته تعریف می‌شوند.

۱.۲.۱ تصویر و ویدیوی دیجیتال

تصویر دیجیتال^۵ (digital image) بسته به رزولوشن از ماتریسی از نقاط ریز کنار هم به اسم پیکسل تشکیل شده‌است که هر پیکسل کوچک‌ترین بخش یک تصویر است. به طوری که اگر هر تصویر را با تعدادی خطوط افقی و عمودی با فاصله ثابت به مربع‌های کوچک‌تری تقسیم کنیم، نماینده روشنایی و رنگ هر مربع حاصل از برخورد نقاط با هم، پیکسل‌ها را تشکیل می‌دهند. هر چه تعداد پیکسل‌های یک تصویر بیشتر باشد، تفکیک پذیری آن بیشتر است.

هر پیکسل در تصاویر رنگی شامل اطلاعات رنگ و روشنایی است. فضا‌های مختلفی برای تعریف رنگ و روشنایی وجود دارند که از معروف‌ترین آن‌ها YUV, RGB و HSI است. هر پیکسل در یک تصویر رنگی به شکل یک سه‌تایی مرتب تعریف می‌شود. هر پیکسل در یک تصویر سیاه و سفید به شکل یک عدد که روشنایی آن را مشخص می‌کند تعریف می‌شود.

یک ویدیوی دیجیتال (digital video) مجموعه‌ای از فریم‌هاست. هر فریم یک تصویر است و ماتریسی از پیکسل‌ها است. پس ویدیو را می‌توان به شکل یک ماتریس سه بعدی از پیکسل‌ها تعریف کرد. فاصله زمانی بین فریم‌های یک ویدیو را نرخ فریم بر ثانیه ویدیو تعیین می‌کند.

۱.۲.۲ پردازش تصویر

پردازش تصویر دیجیتال (digital image processing) شاخه‌ای از پردازش سیگنال دیجیتال است که در جهت بهره‌گیری و اعمال الگوریتم‌های کامپیوتری پردازش تصویر در تصاویر دیجیتال می‌کوشد. برای هدف‌هایی مانند بهبود کیفیت تصاویر، فشرده‌سازی آن‌ها برای انتقال، استخراج اطلاعات بصری و غیره روش‌های پردازش تصویر کاربرد دارند.

۱.۲.۳ پردازش ویدیو

پردازش سیگنال‌های ویدیویی بر مبنای پردازش تصویر است. بدین معنی که در آن برای پردازش ویدیو از روش‌های پردازش تصویر روی فریم‌های ویدیو استفاده می‌شود.

در مهندسی برق، پردازش ویدیو بخشی از مبحث پردازش سیگنال است که ورودی و خروجی سیستم‌ها در آن از جنس ویدیوی دیجیتال یا تصویر دیجیتال هستند.

یک ویدیوی دیجیتال دارای اطلاعات زمانی و مکانی است. پیکسل‌های هر فریم یک ویدیو اطلاعات حوزه زمان است و فریم‌های یک ویدیو اطلاعات مکانی است. تغییراتی که در هر پیکسل با گذشت زمان ایجاد می‌شود، اطلاعات زمانی ویدیو است.

۱.۲.۴ سیگنال در حوزه زمان

temporal video signal

اگر پیکسل‌های متناظر فریم‌های یک ویدیو پشت سر هم چیده شود، یک سیگنال یک بعدی حوزه زمان (temporal video signal) حاصل می‌شود. به عبارت دیگر اگر تنها به یک نقطه از ویدیو نگاه کنیم، خواهیم دید که در گذر زمان روشنایی و رنگ آن پیکسل تغییر می‌کند. پس به تغییرات زمانی رنگ و روشنایی هر پیکسل، سیگنال حوزه زمان آن پیکسل می‌گوییم.

۱.۲.۵ فیلترهای مکانی

می‌توان فیلترهای مختلفی را روی یک فریم اعمال کرد که به آن‌ها فیلتر مکانی (Spatial filters) گفته می‌شود. از این نوع فیلترها می‌توان فیلتر گوسی یا فیلتر میانه را نام برد. این فیلترها می‌توانند برای تشخیص لبه تصاویر، نرم‌تر کردن و شارپ کردن (تیز کردن) تصاویر، کاهش نویز، جدا کردن باندهای فرکانسی و غیره به کار روند. یک فیلتر پرکاربرد پایین‌گذر، فیلتر گوسی است. که در این پایان‌نامه از آن استفاده شده است.^{۸۷۶}

۱.۲.۶ هرم گوسی و هرم لاپلاسی

هرم گوسی مجموعه‌ای از عملیات نمونه‌کاهی (downsampling) و فیلترهای مکانی است که تصویر را به باندهای فرکانسی مختلف تقسیم می‌کند.

هرم گوسی می‌تواند تصویر را به باندهای فرکانسی پایین‌گذر مختلفی تقسیم کند. در هرم لاپلاسی این باندها میان‌گذر هستند. تبدیل تصویر به هرم گوسی و لاپلاسی یک تبدیل یک‌به‌یک است یعنی با داشتن هرم لاپلاسی یک تصویر، می‌توان تصویر اصلی را بازسازی کرد.

هرم گوسی و لاپلاسی کاربردهای مختلفی در فراتفکیک‌پذیری (superresolution)، استخراج اطلاعاتی مانند لبه‌ها، بهبود کیفیت تصویر، شناخت فعالیت انسان و ترکیب تصاویر، دارد.^{۹ ۱۰ ۱۱ ۱۲ ۱۳}

۱.۲.۷ فیلتر زمانی

همانطور که اشاره شد، به توالی پیکسل‌های متناظر از فریم‌های مختلف یک ویدیو، یک سیگنال حوزه زمان گفته می‌شود. در صورتی که یک فیلتر دیجیتال روی همه سیگنال‌های زمانی اعمال شود، به این فرایند فیلتر کردن زمانی ویدیو گفته می‌شود. از کاربردهای فیلتر زمانی روی ویدیو می‌توان کاهش نویز، دنبال‌کردن حرکت و فشرده‌سازی و کد کردن ویدیو را نام برد.^{۱۴ ۱۵ ۱۶ ۱۷}

۲ فصل ۲ مروری بر ادبیات پژوهش

۲.۱ دسته‌بندی

روش‌های بزرگنمایی به طور کلی به دو دسته تقسیم می‌شوند: روش‌های اوپلری، روش‌های لاگرانژی.

در روش‌های لاگرانژی، به اجسام نگاه می‌شود و حرکت و تغییرات در آن‌ها دنبال می‌شود.

در حالی که در روش‌های اوپلری به ویدیو به عنوان یک کل نگاه می‌شود. در این روش ویدیو یک تابع در حوزه زمان و مکان تعریف می‌شود و از روش‌های پردازش سیگنال برای بررسی و پردازش آن استفاده می‌شود.

روش‌های بزرگنمایی به طور کلی به دو دسته روش‌های اوپلری و روش‌های لاگرانژی تقسیم می‌شوند. رویکرد لاگرانژی با استفاده از استخراج میدان حرکت، پیکسل‌ها را مستقیماً حرکت می‌دهد. در حالی که رویکرد اوپلری با نگاه به ویدیو به عنوان یک سیگنال سه‌بعدی در حوزه زمان و مکان، بدون دنبال کردن حرکت، از روش‌های پردازش سیگنال برای بزرگ‌نمایی حرکت استفاده می‌کند. همه این روش‌ها از سه مرحله تشکیل شده‌اند. این مراحل شامل تبدیل فریم‌ها به یک نمایه جایگزین، سپس پردازش و تغییر دادن آن نمایه و در نهایت ساخت ویدیوی بزرگ‌نمایی‌شده با بازسازی و رندر نمایه پردازش‌شده است.^{۱۸}

۲.۲ کارهای پیشین

تحلیل‌های حرکت در ویدیو به منظور اعمال پردازش‌هایی روی آن‌ها نخست در سال ۲۰۰۰ در پروژه‌ای در انجام شد.^{۱۹} این پروژه به منظور ساخت انیمیشن‌های استاپ‌موشن (stop motion) از روی تصاویر و بهبود کیفیت حرکت در آن‌ها با motion blur انجام شد.

پیش از آن پژوهش‌هایی برای یافتن حرکت در بین فریم‌های ویدیو انجام شده بود. برای مثال تحقیقی در سال ۱۹۹۰^{۲۱} با هدف «محاسبه حرکت از روی فریم‌های ویدیو» در کنفرانس بین‌المللی بینایی ماشین ارائه شد که از رایانه برای تحلیل حرکت در ویدیوهای دیجیتال استفاده می‌کرد.

ایده اصلی بزرگنمایی حرکت با استفاده از رایانه در ویدیوهای دیجیتال، اولین بار در سال ۲۰۰۵ در مؤسسه MIT مطرح شد.^{۲۱} بزرگنمایی حرکت در ویدیو در ابتدا با عنوان «میکروسکوپی برای حرکت‌های بصری» معرفی شد که هدف آن تقویت حرکت‌های کوچک در یک دنباله ویدیو بود.

با یک رویکرد لاگرانژی، تلاش این پژوهش استفاده از بینایی ماشین برای اندازه‌گیری دقیق میزان تغییر، دسته‌بندی پیکسل‌ها و سپس تقویت تغییر در آن‌ها بود. در این پژوهش برای بالا بردن دقت در این روش نیاز به دستکاری انسان بود.

طی سال‌های ۲۰۰۵ تا ۲۰۱۲ تلاش‌ها برای بهبود کیفیت بزرگنمایی در دانشگاه MIT منجر به روش جدیدی گردید که در سال ۲۰۱۲ تحت عنوان «بزرگنمایی اوپلری ویدیو» معرفی شد.^{۲۲} در روش اوپلری نگاه جدیدی به ویدیو شد و ویدیو به عنوان یک مجموعه از سیگنال‌های حوزه زمان بررسی شد. این روش به جای دسته‌بندی پیکسل‌ها با بینایی ماشین و اندازه‌گیری حرکت آن‌ها، از روش‌های پردازش سیگنال دیجیتال برای بزرگنمایی تغییرات در پیکسل‌های ویدیو بهره گرفت. روش‌های اوپلری بزرگنمایی ویدیو در ابتدا رویکردی خطی داشتند و بزرگنمایی خطی روی ویدیو انجام می‌دادند.

گاهی تغییرات اندک در ویدیو دامنه‌ای پایین‌تر از دامنه نویز دارند. بدین معنا که در صورت بزرگنمایی آن‌ها، نسبت دامنه سیگنال به دامنه نویز کمتر شده و کیفیت ویدیوی نتیجه کاهش پیدا می‌کند. برای جلوگیری از بزرگنمایی نویز استفاده از هرم گوسی و هرم لاپلاسی معرفی شد. در این روش به جای نگاه جداگانه به هر پیکسل، یک تصویر به بخش‌های مربعی کوچکی شامل تعداد مشخصی پیکسل تقسیم می‌شود و با میانگین گرفتن از پیکسل‌های موجود در این ناحیه (با استفاده از فیلتر گوسی)، نسبت سیگنال به نویز افزایش پیدا می‌کند. سپس بزرگنمایی برای این سیگنال جدید انجام می‌شود. در نتیجه کیفیت ویدیوی حاصل از نظر نویز بهبود پیدا می‌کند.

در سال ۲۰۱۳ روش اوپلری فازی برای بزرگنمایی ویدئو ابداع گردید.^{۲۳} در این روش به جای رویکرد خطی، با اعمال فیلترهای مختلف و سپس بررسی تغییرات فاز سیگنال ویدئو در حوزه فوریه و تقویت این تغییرات، بزرگنمایی انجام می‌شود. در این روش از هرم‌هایی از جمله هرم هدایت (complex steerable pyramid) و هرم ریس (Riesz pyramid) برای تجزیه ویدئو به فازها استفاده می‌شود.

در سال ۲۰۱۴ از بزرگنمایی اوپلری ویدئو به عنوان روشی برای structural modal identification استفاده شد.^{۲۴} در این روش به جای استفاده از لرزش‌سنج لیزری یا سنسور شتاب‌سنج، از دوربین با سرعت بالا (high speed camera - دوربین با نرخ فریم بالا) برای سنجش تحرک در سازه‌ها استفاده شد. نتایج دوربین، مشابه سنسورهای دیگر بود.

در سال ۲۰۱۸ و با گسترش استفاده از شبکه‌های عصبی عمیق، از این روش برای بزرگنمایی ویدئو به کار گرفته شد. در پژوهشی از این تکنیک برای بزرگنمایی ویدئو استفاده شد. در این روش با رویکرد یادگیری ماشین، از یک شبکه عصبی کانولوشنی (convolutional neural network) برای بزرگنمایی ویدئو استفاده می‌شود.^{۲۵}

۲.۳ روش‌های پیشنهادی

در این پژوهش، سه رویکرد مختلف برای بزرگنمایی ویدئو پیش گرفته می‌شود و این رویکردها به ترتیب دنبال می‌شوند.

رویکرد آغازین طراحی یک الگوریتم بر طبق مفاهیم پایه‌ای کتاب‌های مرجع دانشگاهی از جمله کتاب پردازش تصویر دیجیتال^{۲۶} است. در این رویکرد سعی بر این بوده که کم‌ترین مفاهیم جدید استفاده شود و درک الگوریتم طراحی شده برای یک دانش‌آموخته مهندسی برق آسان باشد. این الگوریتم با استفاده از برنامه‌نویسی با استفاده از زبان برنامه‌نویسی پایتون و روی سیستم عامل گنو/لینوکس پیاده‌سازی و سپس آزمایش می‌شود.

در رویکرد دوم از الگوریتم اوپلری خطی طراحی شده در مقاله وو و همکارانش^{۲۷} استفاده می‌شود. نخست این الگوریتم بررسی شده و با مفاهیم پایه پردازش سیگنال و پردازش تصویر تطبیق داده می‌شود سپس با پلتفرم مشابه رویکرد نخست پیاده‌سازی رایانه‌ای شده و سپس آزموده می‌شود.

در رویکرد سوم الگوریتم جدیدی بر پایه آموخته‌های کسب شده از دو تلاش پیشین طراحی، تکمیل و سپس کاستی‌های آن بررسی، کشف و برطرف می‌شود. پیاده‌سازی رایانه‌ای این الگوریتم نیز مشابه دو الگوریتم پیشین در زبان پایتون خواهد بود تا اجرای برای استفاده یا آزمایش آسان باشد.

۳ فصل ۳ پیاده‌سازی و آزمایش

در این فصل هر یک از سه روش توضیح داده شده در فصل قبل پیاده‌سازی و آزمایش خواهند شد.

همه پیاده‌سازی‌های انجام‌شده و برنامه‌های نوشته‌شده در محیط گنو/لینوکس توزیع KDE neon 5.22 انجام شده و از زبان پایتون نسخه ۳.۸ استفاده شده است.

الگوریتم‌ها با پارادایم تابعی^{۲۸} (functional) طراحی شده‌اند. بنابراین استفاده از برنامه برای ویدیوهای مختلف و همچنین تغییر دادن آن‌ها در صورت نیاز برای افراد مسلط به این سبک آسان خواهد بود. برنامه‌های نوشته‌شده مستقل از سیستم‌عامل هستند و در هر محیطی که از مفسر و کتابخانه‌های پایتون پشتیبانی کند قابل انجام است.

برای پیاده‌سازی در سیستم‌عامل‌های دیگر مانند ویندوز یا توزیع‌های دیگر لینوکس مانند raspbian، نیاز به نصب یک مفسر پایتون با نسخه‌ای بالاتر از ۳.۷ است (ترجیحاً cpython 3.8 زیرا تست‌های این پروژه با این ابزار انجام شده‌است). به همراه کتابخانه‌های openCV، numpy و scipy. برای اجرای برنامه و تولید خروجی نیازی به محیط گرافیکی نیست و برنامه می‌تواند بر روی یک سرور لینوکسی یا ویندوزی نیز اجرا شود.

۳.۱ الگوریتم دیفرانسیلی

در این مرحله الگوریتمی طراحی می‌شود که با استفاده از اختلاف فریم‌ها تغییرات را آشکار کند. سعی بر این است که به کمک مفاهیم موجود در کتاب مرجع پردازش تصویر الگوریتمی برای بزرگ‌نمایی حرکت طراحی شود. هدف از این رویکرد، ساده بودن درک الگوریتم برای دانشجویان مهندسی برق و معرفی آن به عنوان یک روش پایه‌ای است.

۳.۱.۱ روش تقویت سیگنال زمانی

در آغاز کار الگوریتمی برای تقویت سیگنال زمانی به منظور آشکارسازی تغییرات در ویدیو طراحی می‌شود. یعنی اساس کار این الگوریتم تقویت سیگنال دیجیتال است و هدف در طراحی این الگوریتم رسیدن به سیگنالی است که اطلاعات حرکت در آن تقویت و بزرگ‌نمایی شده است.

۳.۱.۱.۱ بررسی ریاضی

مشتق تابع، تغییرات یک تابع در حوزه زمان است. حال اگر به ازای یک تابع مشخص، تابعی مشابه داشته باشیم که مشتق آن عددی بزرگ‌تر از تابع اصلی باشد اما همان اطلاعات تابع اصلی را داشته باشد، می‌توانیم به تابعی برسیم که مشابه تابع اصلی است با این تفاوت که تغییرات حوزه زمان در آن بزرگ‌تر شده است. طبق عبارت زیر:

$$\begin{aligned}f_2' &= f_1' \times \beta \Rightarrow \int f_2' = \int f_1' \cdot \beta \Rightarrow f_2 - f_0(0) = (f_1 - f_1(0)) \cdot \beta \\ \text{let: } f_2(0) &= f_1(0) \\ f_2 &= f_1 \cdot \beta - f_1(0) \cdot (\beta - 1) \\ \Rightarrow f_2 &= f_1 \cdot \beta - C\end{aligned}$$

برای رسیدن به تابع بزرگ‌نمایی شده کافی است سیگنال حوزه زمان در عددی ضرب شده و مقدار ثابت C از آن کم شود.

با این روش، تابع حاصل مثل تابع اصلی است با این تفاوت که هر کجا از مقدار اولیه خود تغییر کرده باشد، این تغییرات بیشتر خود را نشان خواهد داد.

برای یک تصویر، این کار شبیه ضرب کردن تصویر در یک عدد برای افزایش روشنایی آن است. با این تفاوت که تصویر از مقدار اولیه خود کسر می‌شود تا نه خود روشنایی بلکه تغییرات روشنایی در آن بیشتر شود.

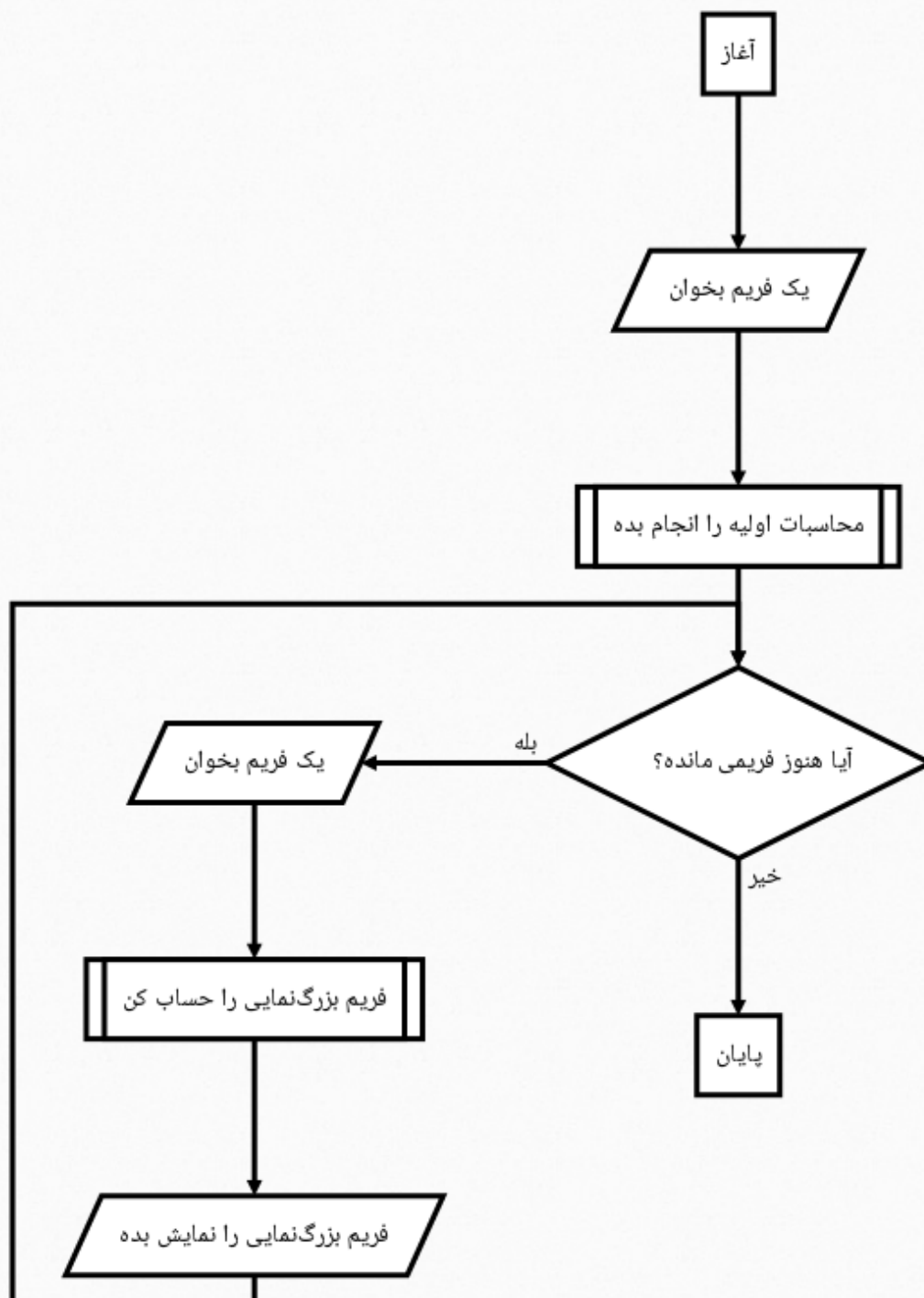
۳.۱.۱.۲ توصیف الگوریتم

توصیف این الگوریتم به صورت زیر است.

۱. محاسبه ثابت C : فریم آغازین یک ویدیو را بخوان و آن را در یکی کم‌تر از ثابت بزرگنمایی ضرب کن. حاصل را مقداری ثابت در نظر بگیر و نامش را فریم تفاوت بگذار.
۲. در ادامه بقیه ویدیو را فریم به فریم بخوان
۳. مقدار فریم را در ثابت بزرگنمایی ضرب کن.
۴. حاصل را منهای فریم تفاوت (ثابت C) کن و آن را به عنوان فریم بزرگنمایی شده نمایش بده یا ذخیره کن.

۳.۱.۱.۳ فلوچارت این الگوریتم

فلوچارت کلی پیشنهادی برای پیاده‌سازی بلادرنگ (realtime) الگوریتم‌های بزرگنمایی به شکل زیر است. بسته به الگوریتم، بخش‌های محاسبات اولیه و روش بزرگنمایی متفاوت خواهند بود.

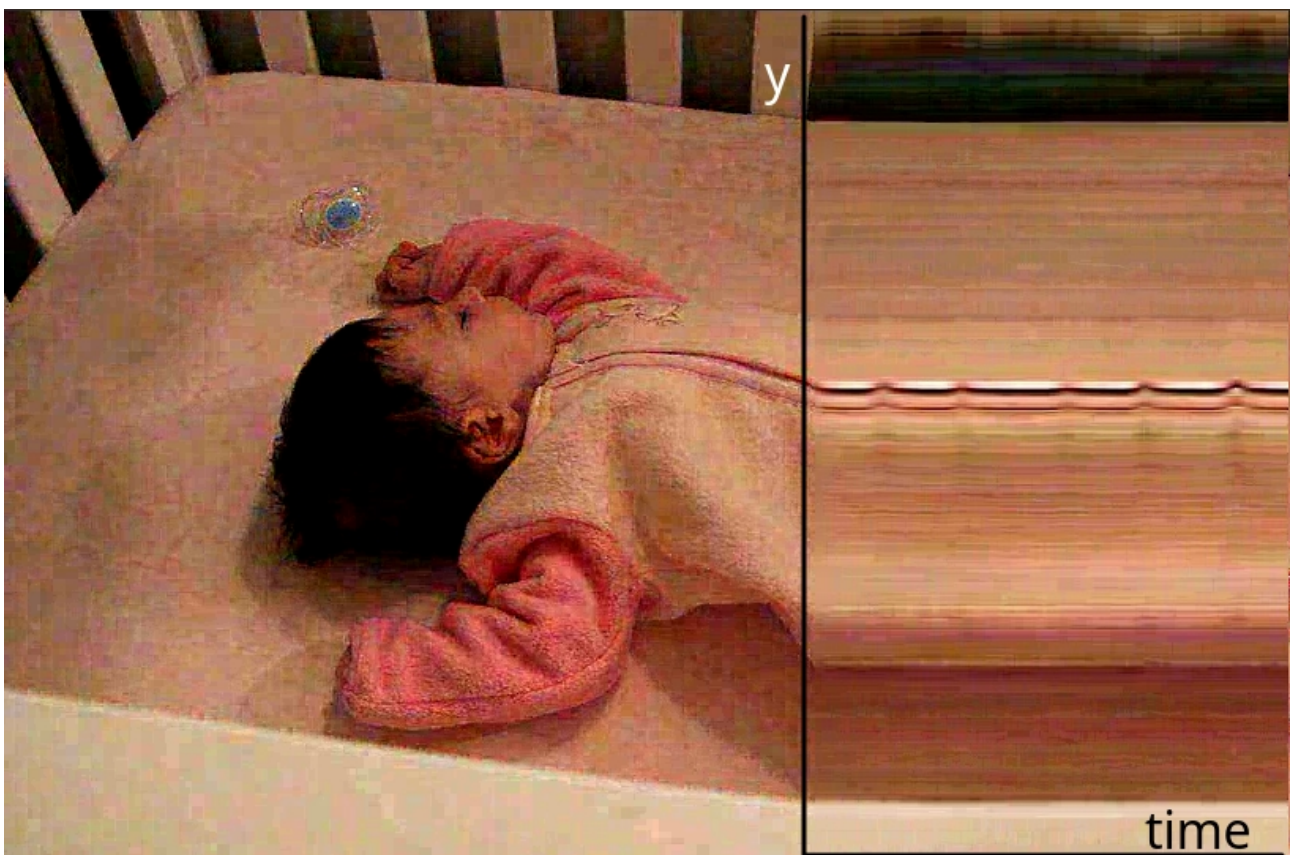


در فلوچارت الگوریتم تقویت سیگنال، محاسبات اولیه شامل محاسبه فریم تفاوت است. بخش محاسبه فریم بزرگ‌نمایی هم شامل ضرب کردن فریم خوانده‌شده در ثابت بزرگ‌نمایی و تفریق فریم تفاوت از آن است.

۳.۱.۱.۴ آزمایش روش تقویت سیگنال

این روش با پایتون پیاده‌سازی و آزمایش شد. حرکت‌هایی در ویدیوی نتیجه آشکار شد که در تصویر زیر قابل مشاهده است. در مقادیر کوچکتر بزرگنمایی، این تغییرات کم‌تر دیده می‌شد و در مقادیر بزرگ‌تر از بزرگنمایی، میزان نویز خیلی زیاد می‌شد.

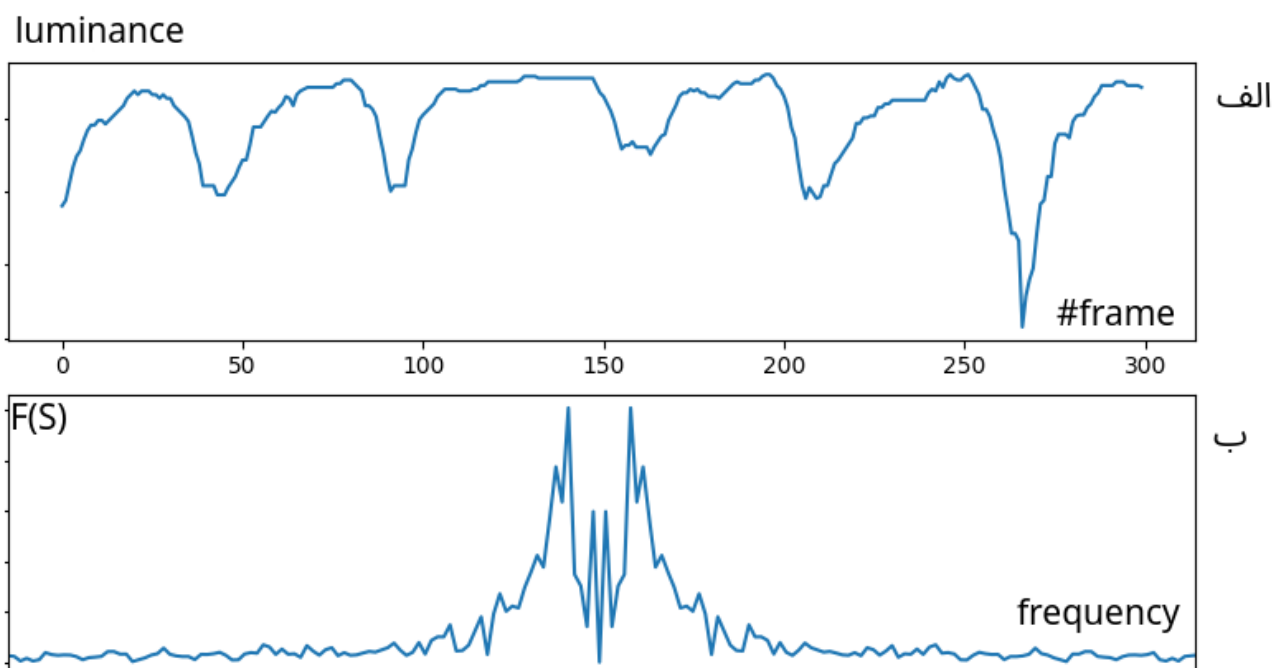
نکته دیگر اینکه این روش تنها در ویدیوهای با دوربین و چشم بسیار ثابت کار می‌کرد. چرا که در صورت تغییرات شدید در ویدیو، اختلاف هر فریم با فریم نخست آن قدر زیاد می‌شد که تفکیک تغییرات اندک دشوارتر می‌شد.



شکل 1: نمودار خط مکانی-زمان برای ویدیوی baby خروجی روش تقویت سیگنال

با این الگوریتم می‌توان سیگنال حوزه زمان بزرگ‌نمایی‌شده را برای یک پیکسل از ویدیو نمایش داد. این کار مثل یک اوسیلوسکوپ برای تغییرات تصویر عمل می‌کند و نوسان‌های سیگنال حاصل را در حوزه زمان و در حوزه فرکانس نمایش می‌دهد.

در شکل زیر یک نمونه از سیگنال حوزه زمان قابل مشاهده است.



شکل 2: الف: سیگنال روشنایی خروجی از سینه نوزاد در خروجی ویدیوی baby قله‌ها و دره‌های این منحنی نمایانگر تنفس نوزاد است.

ب: تبدیل فوریه سیگنال الف که مولفه DC آن حذف شده است.

۳.۱.۲ روش دیفرانسیلی بزرگ‌نمایی تغییرات در سیگنال حوزه زمان

مشکل اصلی روش تقویت سیگنال این است که در صورت جابه‌جایی زیاد دوربین یا جسم، کارایی خود را از دست می‌دهد. پس با مستقل کردن سیستم از فریم اولیه سعی بر رفع اشکال و بهبود روش قبلی شد. ابتدا بزرگ‌نمایی تغییرات با ریاضیات بررسی شد سپس الگوریتم مناسبی برای پیاده‌سازی آن طراحی و در نهایت آزمایش شد.

۳.۱.۲.۱ تحلیل ریاضی

الگوریتم بزرگ‌نمایی تغییرات در سیگنال حوزه زمان مشابه روش قبلی شروع می‌شود. برای تغییرات در حوزه زمان اختلاف هر فریم با فریم قبلی خود محاسبه می‌شود تا مشتق سیگنال حوزه زمان برای هر پیکسل به دست آید و سپس از آن برای بزرگ‌نمایی تغییرات استفاده می‌شود. برای اینکه بزرگ‌نمایی مستقل از فریم آغازین انجام شود، می‌توان به جای محاسبه تغییر از فریم آغازین، تغییرات محلی را استفاده کرد که رابطه ریاضی آن در زیر نشان داده شده است.

$$f_{amplified}(t) = f(t) + \alpha \cdot f'(t)$$

همچنین می‌توان به جای مشتق، از تغییرات فریم فعلی نسبت به B فریم پیشین استفاده کرد. در این صورت به غیر از تغییرات آنی، تغییراتی که در زمان طولانی‌تری اتفاق می‌افتد نیز آشکار می‌شوند. مدل ریاضی این روش به شکل زیر است.

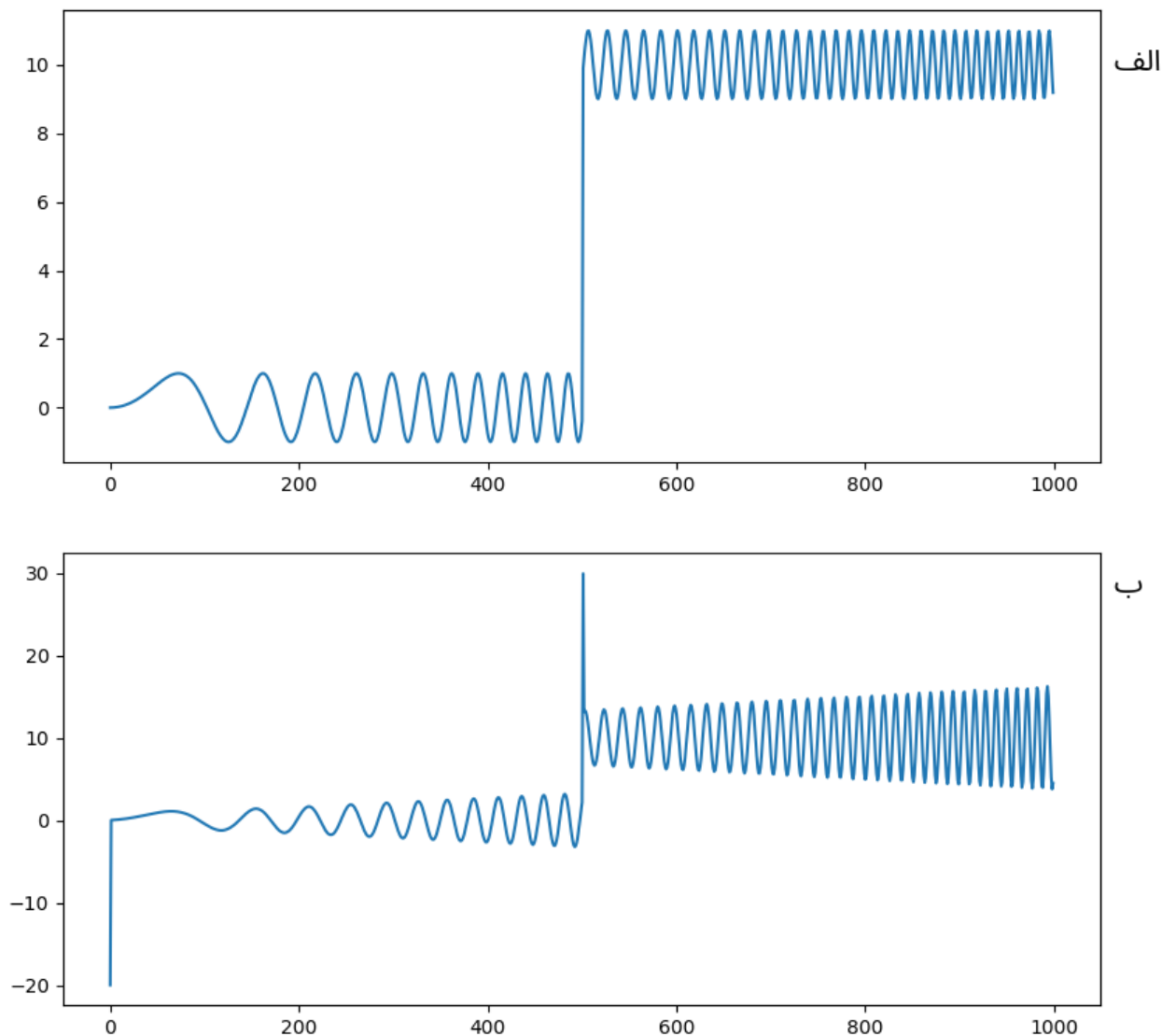
$$f_{amplified}(t) = f(t) + \alpha \cdot (f(t) - f(t - B))$$
$$f_{amplified}(t) = (\alpha + 1) \cdot f(t) - \alpha \cdot f(t - B)$$

در این رابطه، B فاصله بین دو فریم است. که مقایسه با آن انجام خواهد شد. با این روش می‌توان بدون توجه به اندازه تابع، فقط تغییرات دامنه را در تابع بیشتر کرد.

۳.۱.۲.۱.۱ شبیه‌سازی کامپیوتری

برای بررسی صحت عملکرد این روش، یک شبیه‌سازی انجام شد. برای این کار روش بزرگ‌نمایی دیفرانسیلی با استفاده از زبان برنامه‌نویسی پایتون پیاده‌سازی شد و سپس با دو آزمایش صحت عملکرد آن بررسی شد.

برای بررسی امکان بزرگ‌نمایی در حضور تغییرات شدید، نیاز است سیگنالی با تغییرات شدید به عنوان ورودی این الگوریتم داده‌شود و خروجی آن بررسی شود. در هر دو آزمایش مجموع یک سیگنال chirp (چهچه) با یک سیگنال پله به عنوان ورودی سیستم داده شد^{۲۹} که خروجی در شکل زیر قابل مشاهده است.



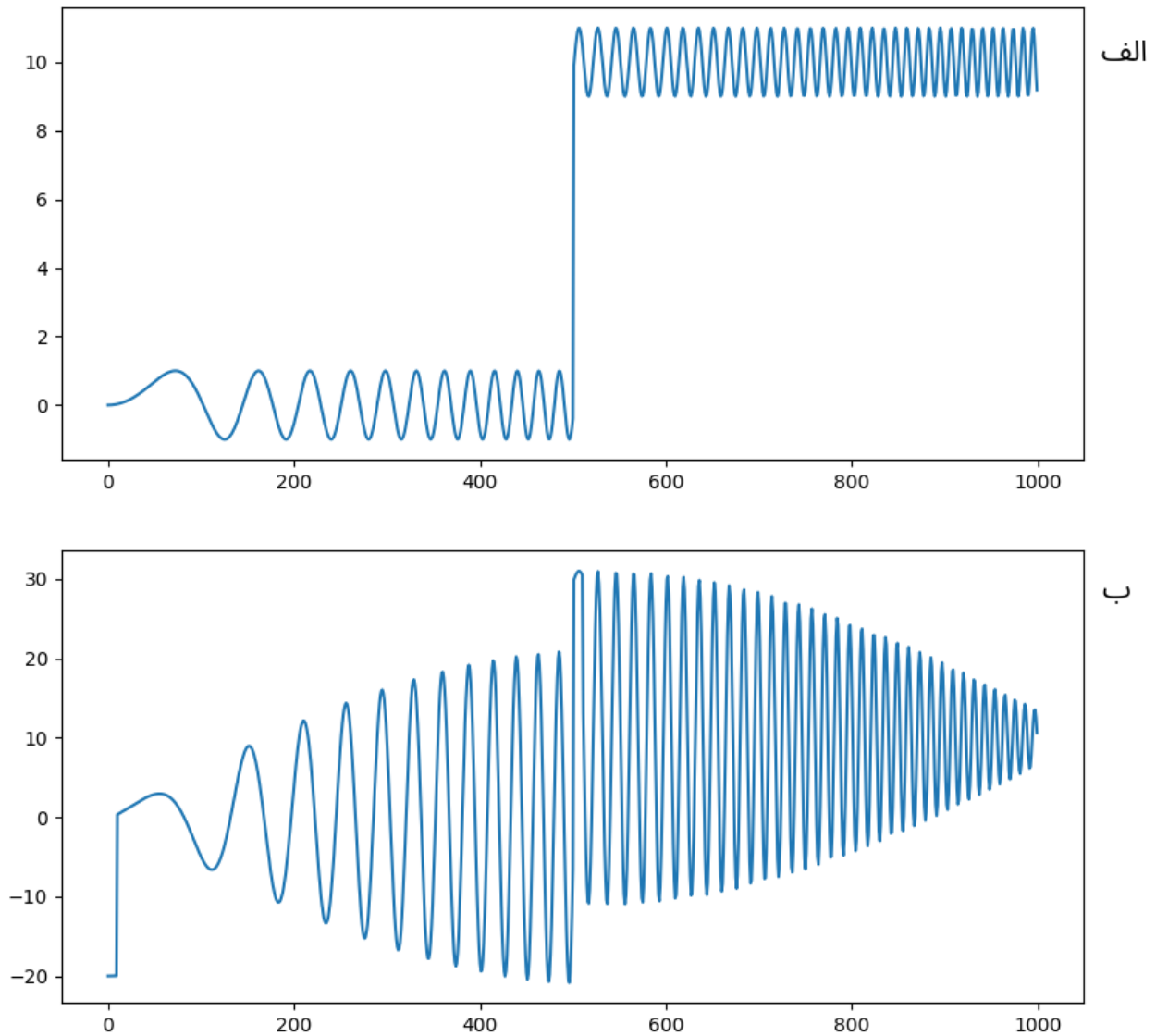
شکل 3: شبیه‌سازی روش دیفرانسیلی: مقایسه با یک فریم قبل‌تر ($B=0$)

الف: ورودی. از مجموع یک سیگنال chirp با یک سیگنال پله استفاده شده است که فرکانس ورودی از ۳ تا ۳۰ هرتز تغییر می‌کند.

ب: سیگنال بزرگ‌نمایی شده خروجی.

می‌توان مشاهده کرد که این الگوریتم در برابر تغییرات شدید دامنه ورودی مقاوم است. همچنین می‌توان دید که روش دیفرانسیلی با فرکانس‌های مختلف رفتار یکسانی ندارد یعنی بعضی فرکانس‌ها را بیشتر تقویت می‌کند.

در آزمایش دوم شبیه‌سازی با اندازه B انجام شد که خروجی در شکل زیر قابل مشاهده است.



شکل 4: شبیه‌سازی دیفرانسیلی. استفاده از شیب متوسط به جای مشتق. مقدار B یک دهم ثانیه انتخاب شده است.

الف: ورودی. از مجموع یک سیگنال chirp با یک سیگنال پله استفاده شده است. فرکانس ورودی از ۳ تا ۳۰ هرتز تغییر می‌کند.

ب: سیگنال بزرگ‌نمایی شده خروجی.

می‌توان مشاهده کرد که این الگوریتم در برابر تغییرات شدید دامنه ورودی مقاوم است. و با مقایسه این آزمایش با آزمایش قبلی می‌توان نتیجه گرفت که اندازه‌های مختلف B رفتارهای متفاوتی با فرکانس‌های مختلف تغییرات خواهند داشت.

۳.۱.۲.۲ طراحی الگوریتم دیفرانسیلی برای بزرگنمایی ویدیو

با توجه به اینکه فریم‌های ویدیو گسسته در زمان قرار می‌گیرند الگوریتم بزرگنمایی باید در حوزه زمان گسسته طراحی شود.

برای فریم‌های ویدیو مدل ریاضی الگوریتم طراحی شده به شکل زیر است:

$$f_{amplified}[N] = f[n] + \alpha \cdot (f[n] - f[n-1])$$
$$f_{amplified}[N] = (\alpha + 1) \cdot f[n] - \alpha \cdot f[n-1]$$

بر پایه این مدل ریاضی الگوریتم روش دیفرانسیلی برای بزرگنمایی ویدیو طراحی شد. در این مدل برای بزرگنمایی ویدیو تنها از یک فریم قبلی هر فریم استفاده شده است.

۳.۱.۲.۳ توصیف الگوریتم

توصیف این الگوریتم به صورت زیر است

به ازای هر فریم ویدیو:

۱. یک فریم از ویدیو را بخوان

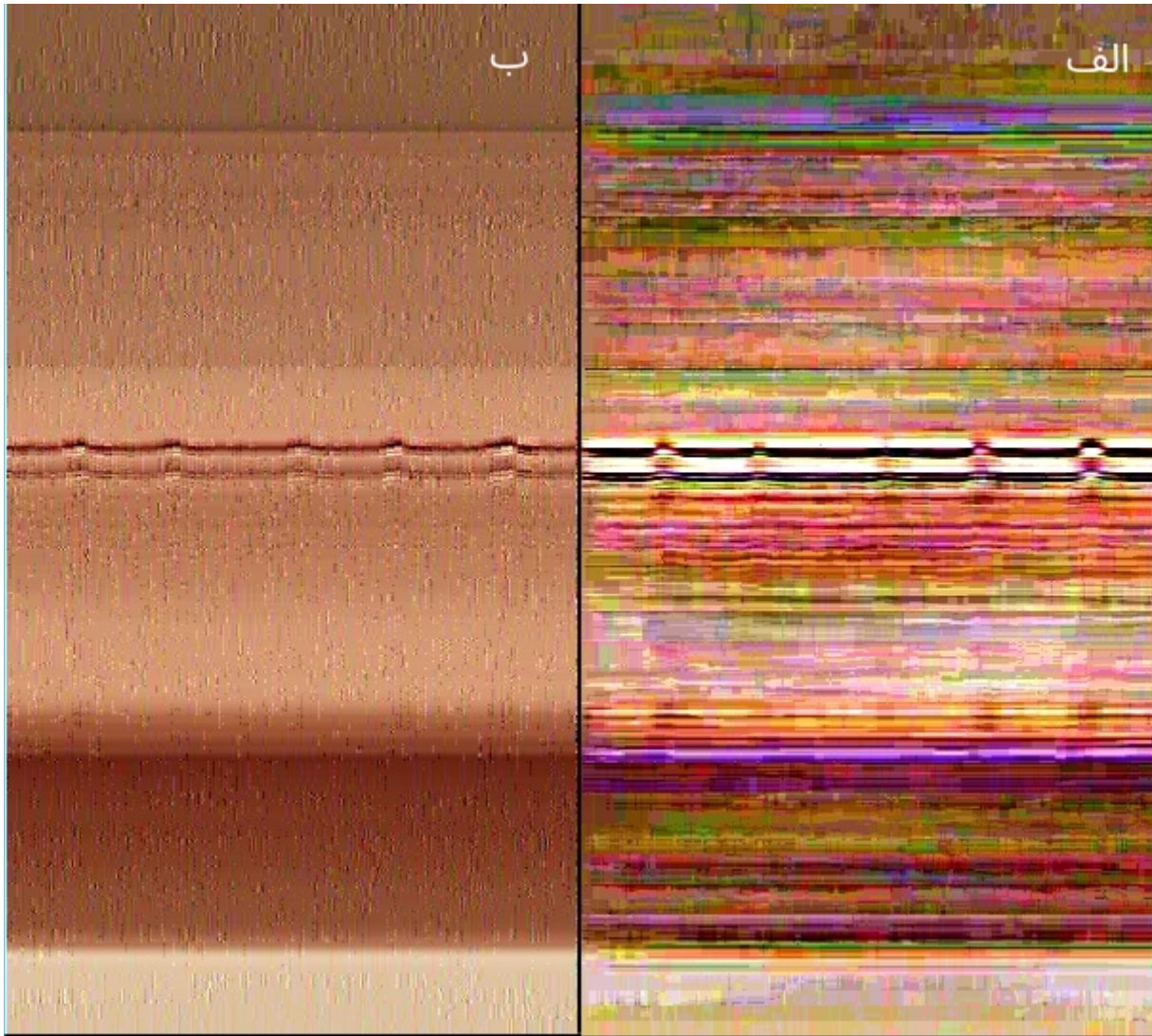
۲. اختلاف فریم خوانده شده را با فریم قبلی آن محاسبه کن.

۳. مقدار اختلاف حاصل را در ثابت بزرگنمایی ضرب کن و آن را با فریم خوانده شده جمع کن.

۴. نتیجه را به عنوان فریم خروجی نمایش بده یا ذخیره کن.

۳.۱.۲.۴ پیاده‌سازی و آزمایش الگوریتم

این الگوریتم در زبان برنامه‌نویسی پایتون پیاده‌سازی و آزمایش شد. در ابتدا روش تقویت سیگنال زمانی و سپس الگوریتم دیفرانسیلی روی ویدیوی baby آزمایش شد که نتایج آن در شکل زیر آورده شده است.



شکل 5: سیگنال‌های حوزه زمان برای مقایسه خروجی دو روش روی ویدیوی baby

محور افقی زمان است.

الف: خروجی الگوریتم تقویت سیگنال

ب: خروجی الگوریتم دیفرانسیلی

در شکل بالا نمونه‌گیری سیگنال‌های حوزه زمان از یک خط عمودی روی تصویر سینه نوزاد انجام شده است. مشاهده می‌شود که آثار مخرب در روش دیفرانسیلی کم‌تر است در حالی که در روش تقویت سیگنال جزئیات ویدیو از بین رفته و بخش‌های دارای حرکت نیز دچار آثار نامطلوب (artifact) شده‌اند.

۳.۱.۳ استفاده از فریم‌های قبل‌تر

همان‌طور که اشاره شد می‌توان به جای استفاده از فریم قبلی برای مقایسه، از فریم‌های دیگر قبلی هم استفاده کرد. رابطه ریاضی گسسته در زمان به شکل زیر است.

$$f_{amplified}[N] = f[n] + \alpha \cdot (f[n] - f[n-B])$$
$$f_{amplified}[N] = (\alpha + 1) \cdot f[n] - \alpha \cdot f[n-B]$$

که در آن B فاصله بین دو فریمی است که مقایسه می‌شوند. این روش مثل استفاده از شیب متوسط به جای شیب لحظه‌ای در ریاضیات است.

۳.۱.۳.۱ توصیف الگوریتم

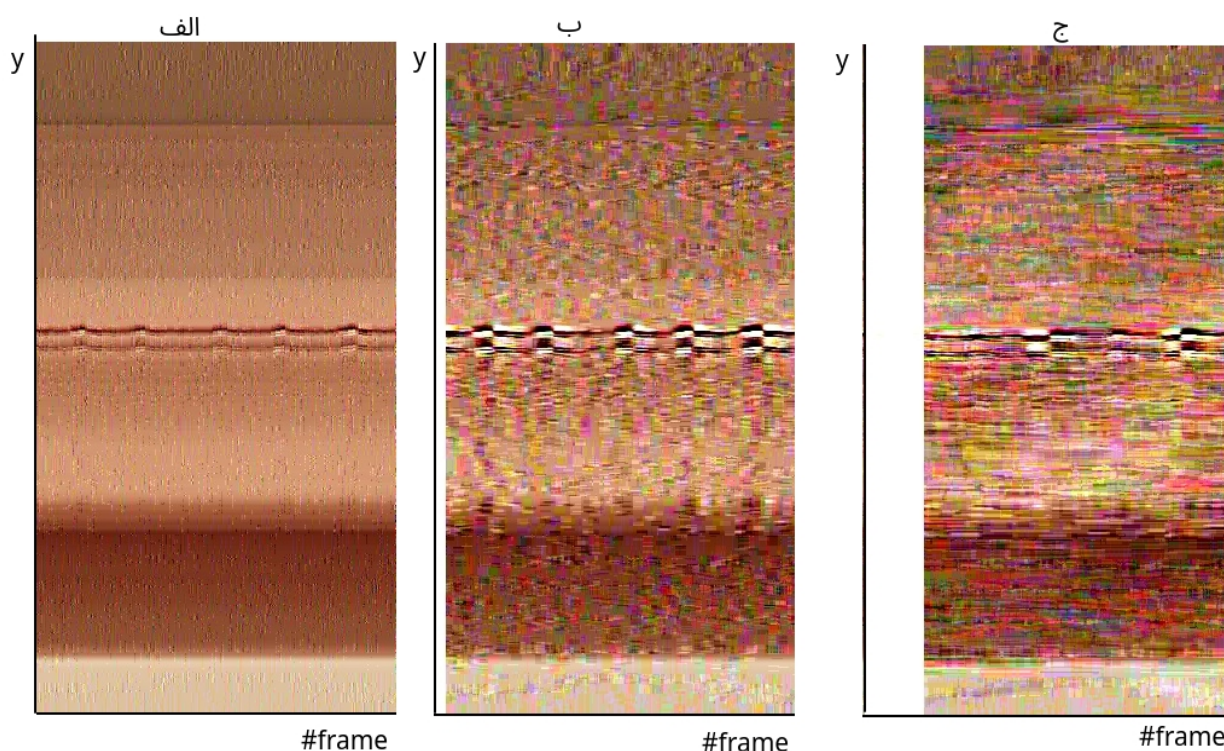
برای دسترسی به فریم‌های قبلی لازم است از یک بافر استفاده کرد.

توصیف الگوریتم به صورت زیر است:

۱. یک ویدیو را فریم به فریم بخوان و فریم‌ها را در یک بافر به اندازه B وارد کن.
۲. به ازای هر فریم خوانده شده اختلاف آن را با B فریم قبلی خود، که درواقع قدیمی‌ترین فریم داخل بافر است، حساب کن.
۳. مقدار اختلاف حاصل را در عدد بزرگنمایی ضرب کن و آن را با فریم خوانده جمع کن.
۴. نتیجه را به عنوان فریم خروجی نمایش بده یا ذخیره کن.

۳.۱.۳.۲ پیاده‌سازی و آزمایش الگوریتم

این الگوریتم در محیط پایتون پیاده‌سازی شد. در شکل زیر نمونه‌ای از سیگنال حوزه زمان خروجی الگوریتم قابل مشاهده است. با بررسی این شکل‌ها مشخص می‌شود که نتایج حاصل مطابق با تحلیل ریاضی هستند.



شکل 6: خروجی الگوریتم دیفرانسیلی با اندازه‌های بافر متفاوت روی ویدیوی baby

نمودارهای سیگنال‌های حوزه زمان. محور افقی زمان است و محور عمودی پیکسل‌های روی یک خط عمودی ترسیم‌شده بر روی سینه نوزاد در ویدیوی baby هستند.

الف: اندازه بافر برابر ۱ فریم (بدون بافر)

ب: اندازه بافر برابر ۱۰ فریم.

ج: اندازه بافر برابر ۵۰ فریم

در شکل بالا در بخش ب مشاهده می‌شود که آشکارسازی در بزرگنمایی نسبت به الف بیشتر شده است و در بخش ج مشاهده می‌شود که اطلاعات تنفس نوزاد در بعضی جاها از بین رفته است. همچنین در شکل ج مشاهده می‌شود که تعداد ۵۰ فریم از ابتدای ویدیو به خاطر خالی بودن بافر بزرگنمایی نشده است.

۳.۱.۴ استفاده از فیلتر گوسی

از نتایج آزمایش‌های انجام شده تأثیر نویزهای حوزه زمان در بزرگنمایی ویدیو ملاحظه می‌شود. نویزهایی که در هر فریم از ویدیو وجود دارند همراه با اطلاعات اصلی ویدیو بزرگنمایی شده و در نتیجه در خروجی نویز بیشتری وجود خواهد داشت. این نویزها در هر فریم با فریم بعدی متفاوت هستند و در سیگنال حوزه زمان در تمام فرکانس‌ها دیده می‌شوند.

ریفرنس برای مستقل بودن نویز از تصویر
ریفرنس برای همبستگی سیگنال‌های زمانی مجاور
سیگنال‌های زمانی کنار هم (پیکسل‌های مجاور) با هم همبستگی دارند. پس اگر آن‌ها را جمع
کنیم، نسبت سیگنال به نویز افزایش پیدا می‌کند.

برای کاهش اثر نویزهای موجود در هر فریم می‌توان از فیلتر گوسی استفاده کرد. این فیلتر پیکسل‌های روی یک ناحیه را با نسبت‌های مشخصی باهم جمع می‌کند و در نتیجه اطلاعات فرکانس‌های بالا در این تصویر از بین می‌رود. می‌توان نویزهای افزوده شده (additive noise) موجود در یک تصویر را مستقل از خود تصویر در نظر گرفت.^۳ همچنین توان فرکانسی تصاویر در فرکانس‌های بالا پایین است. در نتیجه با حذف فرکانس‌های بالای یک تصویر، نسبت سیگنال به نویز افزایش پیدا می‌کند. پس اعمال یک فیلتر گوسی باعث می‌شود یک تصویر با نویز کمتر حاصل شود. حال از این فریم می‌توان برای بزرگنمایی ویدیو استفاده کرد.

۳.۱.۴.۱ توصیف الگوریتم

الگوریتم طراحی شده برای بخش بزرگنمایی فریم به شکل زیر است.

برای بخش بزرگنمایی:

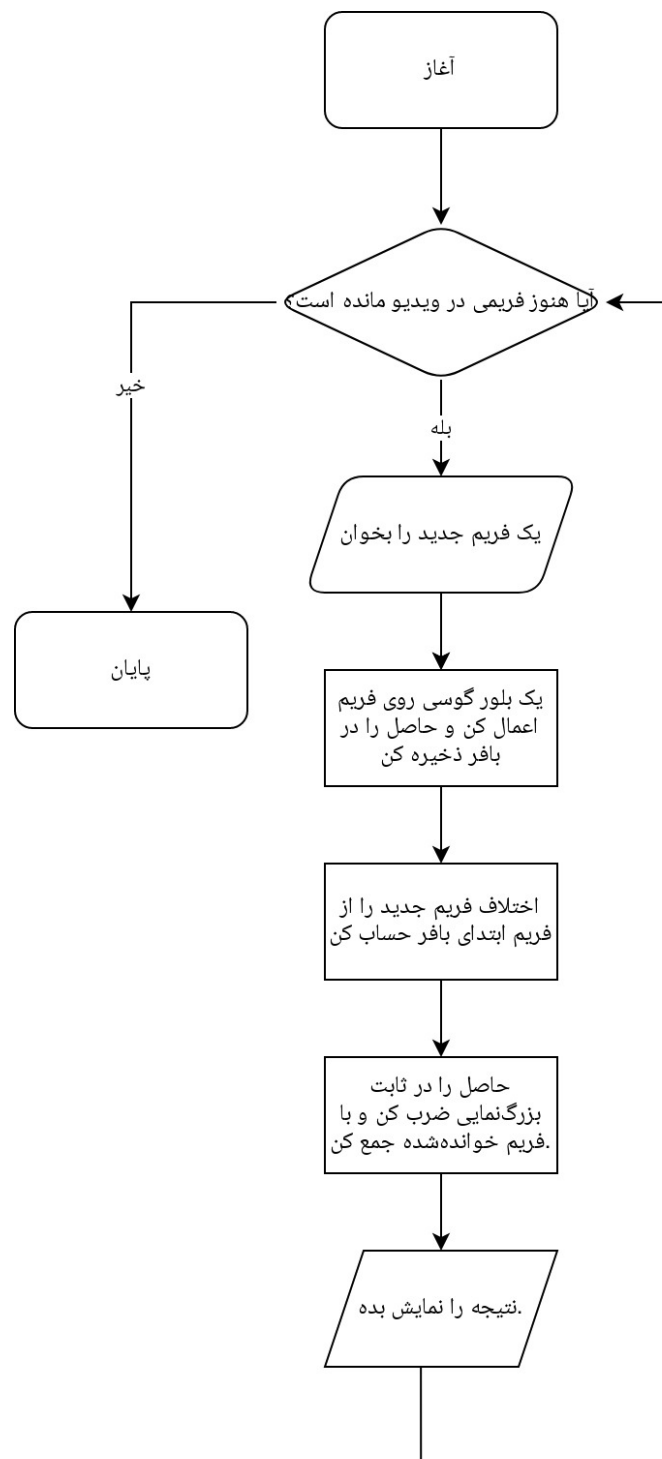
۱. فریم جدید را بخوان و یک فیلتر گوسی روی آن اعمال کن و آن را در یک بافر ذخیره کن.

۲. فریم فیلترشده را در ثابت بزرگ‌نمایی ضرب کن و نسخه فیلترشده B فریم قبل‌تر را از آن کم کن.

۳. فریم حاصل را با فریم خوانده‌شده جمع کن و نتیجه را نمایش بده.

۳.۱.۴.۲ **فلوچارت الگوریتم**

فلوچارت الگوریتم روش دیفرانسیلی با فیلتر گوسی به شکل زیر است.



شکل 7: فلوچارت الگوریتم پایانی روش دیفرانسیلی

۳.۱.۴.۳ شبکه‌کد الگوریتم روش دیفرانسیلی با فیلتر گوسی

شبکه‌کد پیشنهادی برای الگوریتم روش دیفرانسیلی با فیلتر گوسی به صورت زیر است. در این شبکه‌کد همه موارد مطرح‌شده شامل بافر و فیلتر گوسی افزوده شده و این الگوریتم به صورت بلادرنگ طراحی شده‌است.

مقادیر اولیه:

یک ویدیو، ثابت بزرگ‌نمایی، اندازه کرنل فیلتر گوسی، یک بافر که در آغاز همه مقادیر آن فریم‌های صفر است. (ماتریس صفر)

```
video_file = open_video(video_file_name)
amplification_constant = 4
kernel_size = (5, 5)
buffer[buffer_size] = 0
```

حلقه اصلی برنامه:

loop():

به ازای هر فریم از ویدیو:

۱. یک فریم بخوان.

```
frame_now = read_frame()
```

۲. یک فیلتر گوسی روی آن اعمال کن و آن را در بافر ذخیره کن.

```
filtered_frame = gaussian(frame_now, kernel_size)
buffer.push(filtered_frame)
```

۳. اختلاف فریم خوانده‌شده را از فریم آغازین بافر حساب کن و اسم این را فریم تفاوت بگذار.

```
diff_frame = frame_now - buffer[oldest]
```

۴. فریم تفاوت را در ثابت بزرگ‌نمایی ضرب کن و آن را با فریم اصلی خوانده‌شده جمع کن.

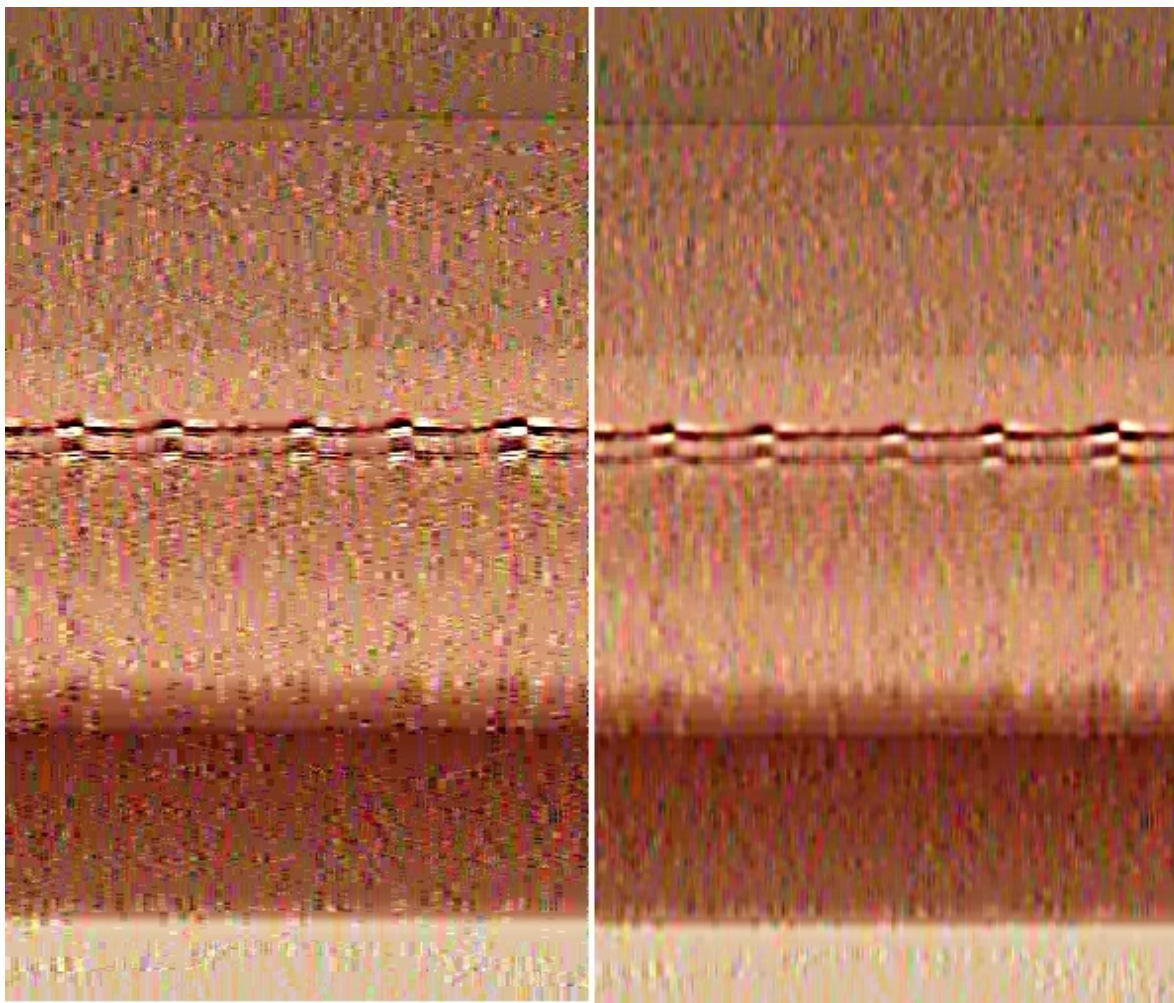
```
amp_diff_frame = diff_frame * amplification_constant
magnified_frame = frame_now + amp_diff_frame
```

۵. حاصل را نمایش بده.

```
show_frame(magnified_frame)
```

۳.۱.۴.۴ پیاده‌سازی و آزمایش الگوریتم

ویژگی اعمال فیلتر گوسی به برنامه پایتون افزوده شد و با ویدیوهای مختلف آزمایش و نتایج آن با حالت بدون فیلتر گوسی مقایسه شد. در شکل زیر یک نمونه سیگنال حوزه زمان برای خروجی ویدیوی baby قابل مشاهده است.



الف

ب

شکل 8 نمونه سیگنال حوزه زمان خروجی تست الگوریتم روی ویدیوی baby

الف: الگوریتم دیفرانسیلی بدون فیلتر گوسی

ب: الگوریتم دیفرانسیلی با فیلتر گوسی

همان‌طور که از شکل ملاحظه می‌شود افزودن فیلتر گوسی باعث کاهش نویز در خروجی می‌شود. در نهایت این الگوریتم به شکل یک کتابخانه پایتون پیاده‌سازی شد. کد این الگوریتم در پیوست ۱ قابل دسترسی است.

۳.۲ الگوریتم اویلری خطی

این الگوریتم و بلوک دیاگرام آن در مقاله اصلی^{۳۱} آورده شده است. و هدف در این پژوهش پیاده‌سازی این الگوریتم با استفاده از زبان برنامه‌نویسی پایتون به منظور مقایسه با روش‌های دیگر می‌باشد.

۳.۲.۱ بررسی کد متلب روش اویلری خطی

در این بخش، کد متلب روش اویلری خطی که در پیوست مقاله اصلی بود مورد بررسی قرار گرفت. و هدف از این کار استخراج الگوریتم با استفاده از تحلیل کد و پیاده‌سازی مجدد آن در پایتون بود. برای استخراج الگوریتم کد و درک چگونگی کارکرد آن روش مهندسی معکوس مورد استفاده قرار گرفت. استخراج الگوریتم با استفاده از تحلیل کد به روش مهندسی معکوس و با طی مراحل زیر انجام شد:

۱. مطالعه کد و تغییر اسم متغیرها

۲. استخراج فلوگراف سیگنال^{۳۲} (نمودار گذر سیگنال)

۳. استخراج بلوک دیاگرام و طراحی الگوریتم

که در ادامه به توضیح هر مرحله پرداخته خواهد شد.

۳.۲.۱.۱ تحلیل کد

در کدی که نوشته شده بود، نام‌گذاری‌ها دقیق و با توضیح کافی نبودند و حتی چند بار از یک اسم برای چیزهای مختلف استفاده شده بود. اولین مرحله تغییر اسم متغیرها از حروف ساده انگلیسی به عبارت‌های بامعنا است. با این کار برای درک برنامه دیگر نیازی به حفظ اسم متغیرها نیست و دنبال کردن ادامه کد آسانتر خواهد شد.

۳.۲.۱.۲ استخراج فلوگراف سیگنال

اگر خطوط کد برنامه را زیر هم بنویسیم و ارتباط آن‌ها را ترسیم کنیم، به یک فلوچارت خواهیم رسید. اما چیزی که نیاز داریم ارتباط بین پیاده‌سازی برنامه و بلوک دیاگرام است. استفاده از فلوگراف برای رسیدن به

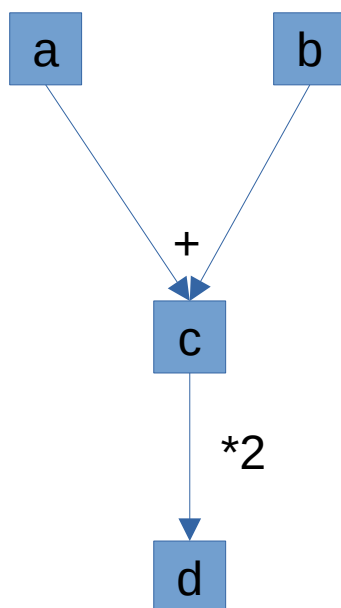
بلوک دیاگرام روشی است که در این کار انتخاب شد. برای رسیدن به فلوگراف باید جریان داده‌ها را استخراج کنیم یعنی به جای اینکه کنترل برنامه را بررسی کنیم، باید ببینیم هر داده یا سیگنال از کجا شروع و به کجا ختم می‌شود و در این مسیر چه اتفاق‌هایی برای آن می‌افتد.

هر متغیر در جایی از برنامه مقدار خود را می‌پذیرد. این مقدار می‌تواند مقداری ثابت باشد، از ورودی گرفته شده باشد یا حاصل ترکیب یک یا چند متغیر دیگر باشد که به آن منبع ورودی متغیر می‌گوییم. اگر اسم متغیرها را بنویسیم و هر کدام را با یک فلش به منبع یا منابع ورودی آن متصل کنیم، یک گراف حاصل می‌شود که فلوگراف سیگنال برنامه است.

به عنوان مثال، اگر کد ما به این شکل باشد:

```
c = a + b  
d = 2 * c
```

نمودار فلوگراف آن به این شکل خواهد بود:

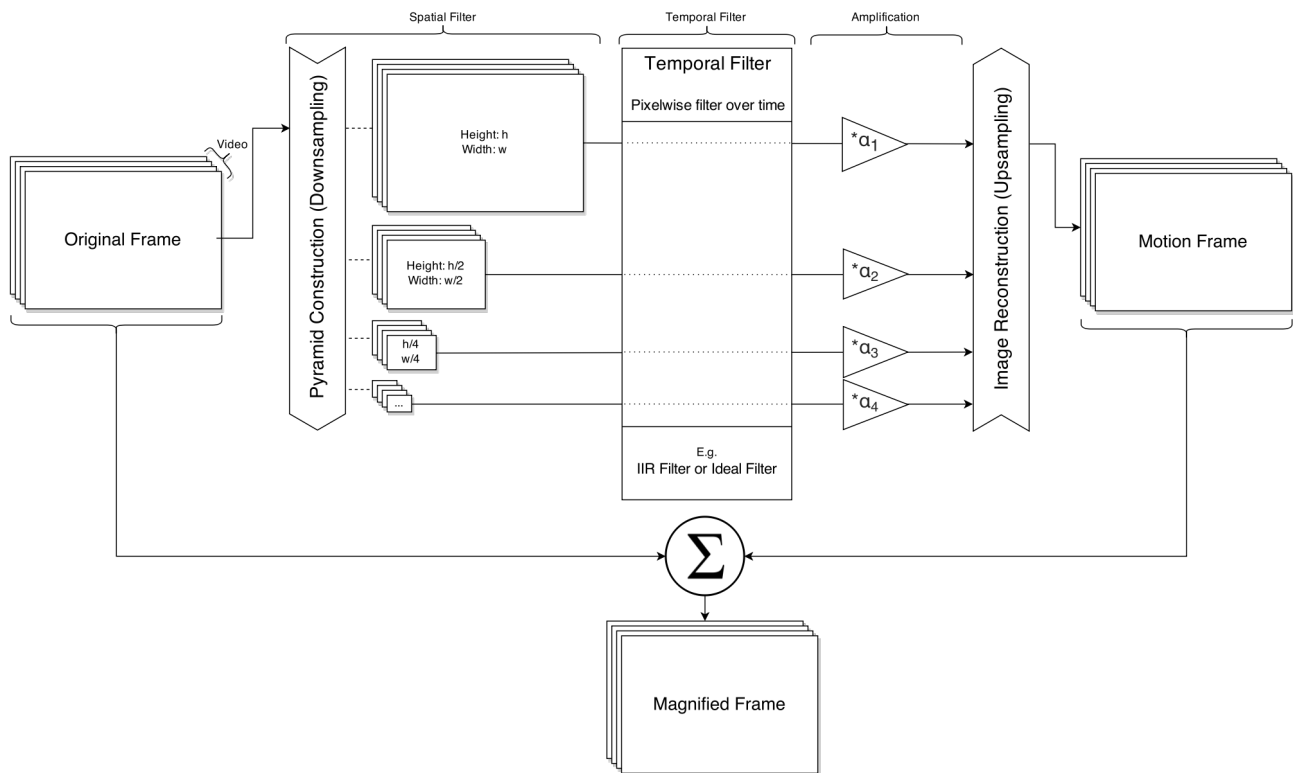


۳.۲.۱.۳ استخراج بلوک دیاگرام

در این بخش به تحلیل و درک اجزای مختلف سیستم پرداخته می‌شود. برای این منظور باید مسیر اصلی جریان سیگنال را تشخیص داده و سپس بخش‌های مختلف آن را پیدا کنیم. برای این کار باید به این نکته توجه کنیم که هر بخش از بلوک دیاگرام تنها یک کار انجام می‌دهد و آن کار را کاملاً مستقل از بقیه بخش‌ها

انجام می‌دهد. برای آسانتر شدن کار در این مرحله بهتر است محاسبات ریاضی را کم از گراف حذف کنیم تا پیچیدگی کار کمتر شود. در پایان این مرحله به یک بلوک دیاگرام کلی خواهیم رسید.

شکل زیر از مقاله مرجع^{۳۳}، بلوک دیاگرام الگوریتم اوپلری خطی را نمایش می‌دهد.



شکل 9: بلوک دیاگرام الگوریتم اوپلری خطی

۳.۲.۲ الگوریتم استخراج شده

با روشی که توضیح داده شد، الگوریتم این کد استخراج و ارتباط آن با بلوک دیاگرام ترسیم شد.

در روش اوپلری خطی، ابتدا با استفاده از هرم گوسی یا لاپلاسی فریم‌های ویدیو به باندهای فرکانس مکانی مختلف تقسیم می‌شود. سپس یک فیلتر زمانی روی هر مجموعه سیگنال‌های حوزه زمان هر طبقه هرم اعمال می‌شود (pixelwise filter over time) تا اطلاعات حرکت در فرکانس مورد نظر استخراج شود. هر یک از طبقات هرم حاصل در یک ثابت بزرگ‌نمایی ضرب شده و پس از فرو ریختن هرم، فریم حرکت تشکیل شده با فریم اصلی جمع می‌شود تا فریم بزرگ‌نمایی شده حاصل شود.

الگوریتم استخراج شده به صورت زیر است:

Step 0: define some constants for magnification, get video properties and generate butterworth lowpass and highpass filters.

Step 1: read frames from video and convert their colors from RGB into NTSC color space.

Step 2: generate Laplacian pyramid for each frame.

Step 3: apply a temporal filter on frames using the generated butterworth filter.

Step 4: amplify frequency bands by multiplying each frame into magnification constant (α)

Step 5: reconstruct frames from Laplacian pyramid and add them up to the original frames.

۳.۲.۳ پیاده‌سازی الگوریتم اوپلری خطی استخراج شده

این الگوریتم در محیط پایتون پیاده‌سازی شد. برای خواندن و نوشتن تصاویر و تبدیل آن‌ها از کتابخانه openCV استفاده شد. در این کتابخانه تصاویر به شکل ماتریس‌هایی ذخیره می‌شوند. برای کار با این ماتریس‌ها از کتابخانه numpy استفاده می‌شود. این کتابخانه توابع مختلفی برای کار با ماتریس‌ها از جمله اعمال ریاضی و عملیات‌های ماتریسی را فراهم می‌کند.

برای طراحی فیلتر حوزه زمان، مشابه مقاله اصلی ذکر شده، از فیلتر باترورث استفاده شد. فیلترهای با مرتبه‌های مختلف از نظر سرعت و میزان حافظه مصرفی و همچنین کیفیت ویدیوی خروجی آزمایش شد. مشابه مقاله مرجع، در پیاده‌سازی مقاله مرجع از فیلترهای با مرتبه‌های پایین‌تر از ۵ استفاده شد. برای طراحی فیلتر و اعمال فیلتر از کتابخانه scipy بخش signal استفاده شد. این کتابخانه توابعی را برای طراحی فیلتر و اعمال فیلترها به روش‌های مختلف بر روی بردارهای numpy فراهم می‌کند.

برای اعمال فیلتر زمانی روی ویدیو نیاز است ابتدا فریم‌های ذخیره شده ویدیو به سیگنال حوزه زمان تبدیل شوند تا روی سیگنال متناظر هر پیکسل فیلتر اعمال شود. برای این کار روشی که استفاده شد ذخیره فریم‌ها در یک ماتریس سه‌بعدی و سپس گرفتن دترمینان آن ماتریس برای رسیدن به سیگنال‌های حوزه زمان است.

temporal signal: $s_{x,y}(t)$

frame: $f_t(x, y)$

frame stack: $F[t, x, y] \rightarrow \text{transpose} \rightarrow F^T[t, x, y] = T[y, x, t] \rightarrow \text{stack of temporal signals}$

بدین ترتیب به روشی برای اعمال غیر مستقیم فیلتر زمانی روی ویدیو دست پیدا کردیم.

برای ساخت هرم گوسی از مجموعه عملیات فیلتر مکانی گوسی و نمونه‌کاهی (downsampling) روی هر

فریم ویدیو استفاده شد. سپس برای ساخت هرم لپلاسی ابتدا یک هرم گوسی ساخته شد و سپس با

نمونه‌افزایی (upsampling) و تفریق طبقات متوالی آن از هم‌دیگر هرم لپلاسی ساخته شد.

این برنامه با پارادایم تابعی پیاده‌سازی شد تا بتوان به سادگی آن را گسترش داد. تمام برنامه نیز به شکل

یک کتاب‌خانه پایتون تعریف شد تا بتوان در برنامه‌های دیگر از آن استفاده کرد.

در نهایت برنامه پیاده‌سازی شده با ویدیوهای مجموعه آزمایش مقاله مرجع آزموده شد و نتیجه مشابه را

تولید کرد.

مقایسه نتایج این روش با روش دیفرانسیلی و روش پیشنهادی که در بخش بعد مطرح خواهد شد، در

انتهای فصل ۳ قرار دارد.

۳.۳ الگوریتم پیشنهادی: الگوریتم همبستگی

در این رویکرد هدف طراحی یک الگوریتم جدید و مناسب پس از مطالعه و بررسی الگوریتم اوپلری خطی و طراحی الگوریتم دیفرانسیلی است. پایه علمی طراحی این الگوریتم مثل روش اوپلری خطی و روش دیفرانسیلی، بر مبنای پردازش سیگنال دیجیتال است. با فرض اینکه تغییرات در ویدیو به شکل یک سیگنال نوسانی سینوسی باشد، روش‌های مختلفی برای تشخیص این حرکت در یک سیگنال یک بعدی وجود دارد. یکی از این روش‌ها در پردازش سیگنال دیجیتال استفاده از تابع همبستگی است. با بررسی همبستگی این سیگنال با یک سیگنال سینوسی نمونه می‌توان حضور آن فرکانس را تشخیص داد. به این شکل که با ضرب سیگنال حوزه زمان هر پیکسل از ویدیو در یک تابع سینوسی با فرکانس مشخص و انتگرال‌گیری از حاصل ضرب و میانگین‌گیری می‌توان میزان همبستگی این دو را به دست آورد.

۳.۳.۱ بررسی ریاضی

برای محاسبه همبستگی یک تابع با تابع سینوسی نمونه آن دو را در یک سیگنال سینوسی نمونه ضرب کرده و از حاصل آن میانگین گرفته می‌شود. اگر در حالت خاص تابع مورد نظر یک تابع سینوسی باشد، برای محاسبه همبستگی باید آن را در خودش ضرب کرد و از حاصل میانگین گرفت:

نکته: همبستگی خودش تقسیم بر زمان دارد.

تابع دقیق همبستگی آورده شود

$$C = \int_0^t f \times g = \int \sin(t) \times \sin(t) dt = \int \sin^2(t) dt$$

$$C = \int \frac{1 - \cos(2at)}{2} dt = \frac{1}{2}t - \frac{1}{4a} \sin(2at)$$

$$\text{correlation} = \frac{1}{2}t - \frac{1}{4a} \sin(2at)$$

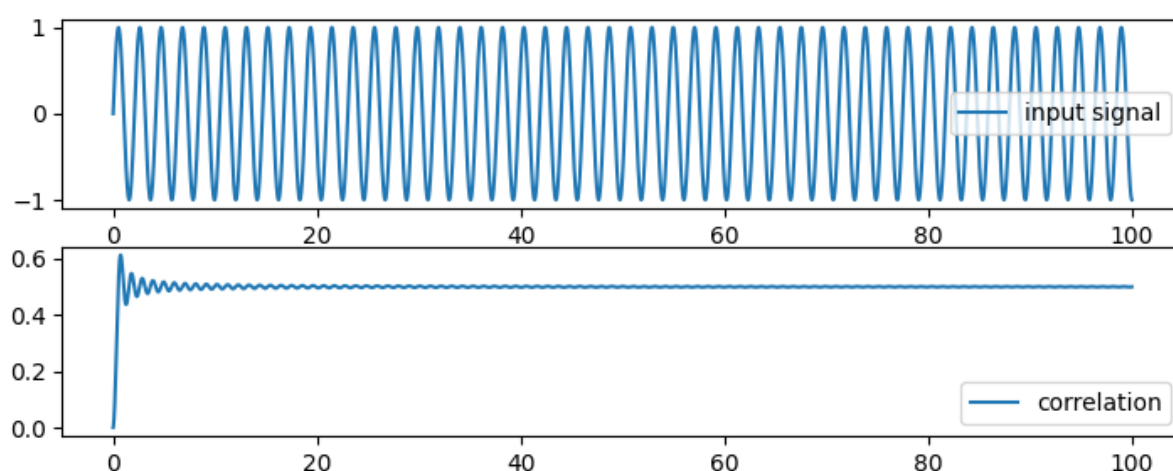
$$\text{normal correlation} = \frac{\text{correlation}}{t}$$

$$\text{normal correlation} = \frac{1}{2} - \frac{1}{4at} \sin(2at)$$

که در این حالت مقدار این تابع به عدد $\frac{1}{4}$ همگرا خواهد بود.

شبیه‌سازی کامپیوتری این عبارت ریاضی در محیط پایتون انجام شد. برای تعریف سیگنال و اعمال محاسبات ریاضی روی آن از کتابخانه numpy و برای ترسیم منحنی حاصل از کتابخانه matplotlib استفاده شد.

در شکل زیر نتیجه خروجی تابع هم‌گرایی یک سیگنال سینوسی با خودش قابل مشاهده است.



شکل 10: بالا: سیگنال سینوسی ورودی.

پایین: حاصل همبستگی سیگنال ورودی با خودش

۳.۳.۱.۱ کسر میانگین فریم‌ها به منظور حذف DC

یک سیگنال حوزه زمان اگر نوسان سینوسی داشته باشد، فرمی مانند عبارت زیر خواهد داشت. این سیگنال علاوه بر فرکانس، یک مقدار اولیه و یک دامنه حرکت دارد.

$$f(t) = f_0 + A \sin(\omega t)$$

$$f(t) = f_0 + A \sin(\omega t)$$

$$C = \int f \cdot \sin(\omega t) dt = \int (f_0 + A \sin(\omega t)) \cdot \sin(\omega t) dt$$

$$C = \int f_0 \cdot \sin(\omega t) dt + \int A \sin(\omega t) \cdot \sin(\omega t) dt$$

برای اینکه اثر مقدار DC در خروجی را حذف کنیم، باید مقدار آن از میانگین فریم‌ها حذف شود. این کار را می‌توان با تعریف سیگنالی جدید مانند عبارت ریاضی زیر انجام داد.

$$g(t) = f(t) - f_0$$

۳.۳.۱.۲ شکل گسسته در زمان این عبارت ریاضی

محاسباتی که در بخش قبل بیان شد برای توابع پیوسته بررسی شد در حالی که سیگنال حوزه زمان یک ویدیو، سیگنالی زمان-گسسته است که فرکانس نمونه‌برداری آن با نرخ فریم ویدیو برابر است.

اگر فریم شماره n را f_n بنامیم و میانگین همه فریم‌ها را f_{mean} بنامیم، همبستگی کل با فرمول زیر قابل محاسبه است.

$$C = \frac{1}{N} \times \sum_{n=0}^N (f_n - f_{mean}) \times \sin(\omega \cdot n)$$

در این عبارت ω فرکانس نمونه‌برداری و N نمایانگر تعداد فریم‌ها است. مطابق بخش‌های قبل می‌توان برای ویدیوهایی با تغییرات کوچک، f_{mean} را برابر با یعنی فریم اول ویدیو (f_0) در نظر گرفت. با استفاده از این فرمول به دست آمده می‌توان الگوریتمی مطابق با روش ریاضی بررسی‌شده را برای یک ویدیوی دیجیتال پیاده‌سازی کرد.

۳.۳.۲ آشکارسازی حرکت‌های نوسانی در ویدیو

با استفاده از محاسبات ریاضی اشاره شده در بخش قبل، الگوریتمی برای آشکارسازی حرکت‌های نوسانی در یک سیگنال ویدیو طراحی شد. ورودی این الگوریتم یک ویدیو و خروجی آن تصویری از حرکت‌های نوسانی موجود در ویدیو است.

۳.۳.۲.۱ توصیف الگوریتم آشکارسازی حرکت نوسانی

توصیف الگوریتم آشکارسازی حرکت نوسانی در ویدیو به شکل زیر است:

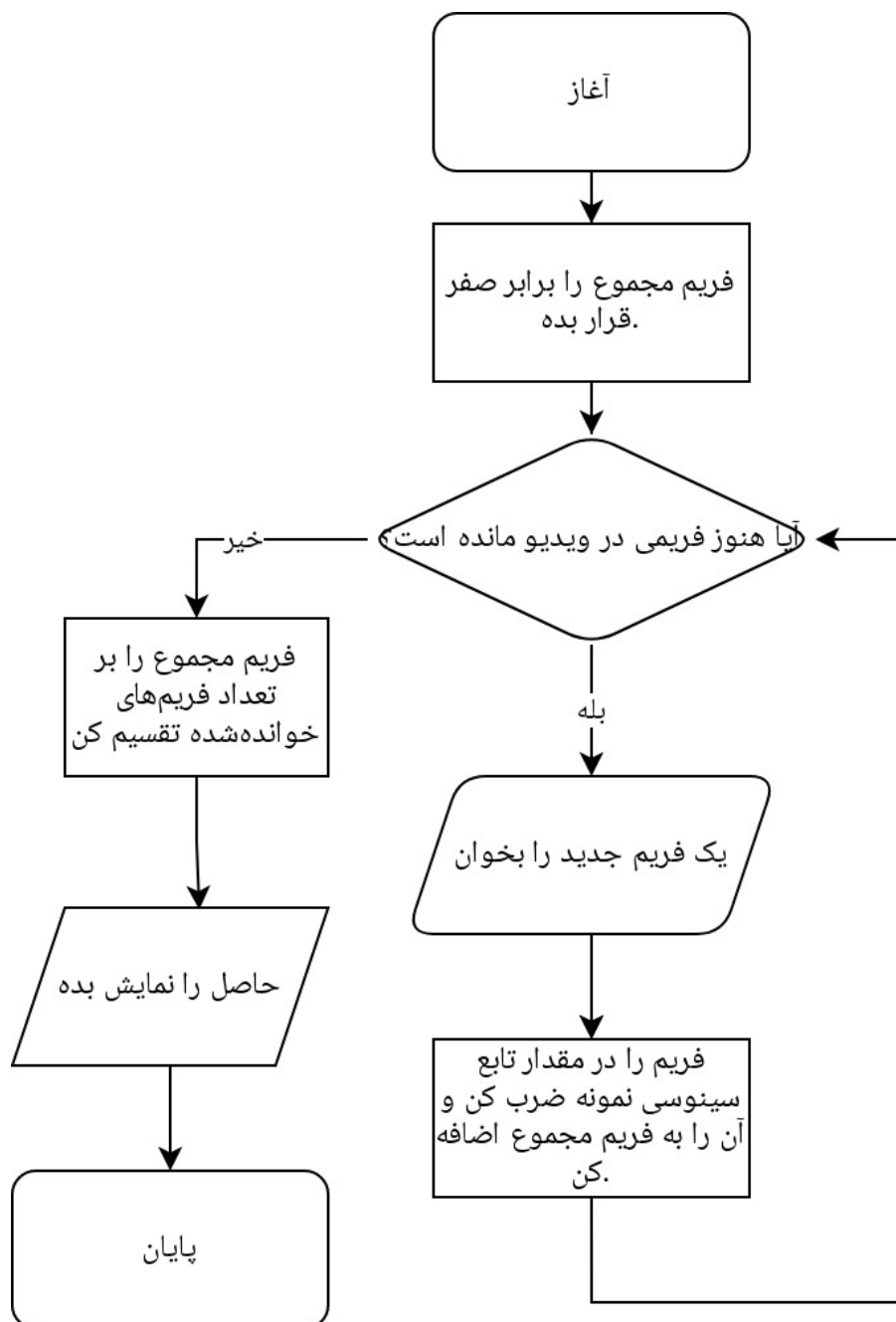
۱. یک ویدیو را انتخاب کن و فریم‌های آن را بخوان.

۲. به ازای هر پیکسل در هر فریم، آن را در مقدار سینوسی نمونه ضرب کن و با مقادیر قبلی جمع کن.

۳. بعد از آخرین فریم، همبستگی را با تقسیم کردن بر تعداد فریم‌ها عددها را محاسبه کن تا تصویر همبستگی به دست آید.

۴. این تصویر را به عنوان همبستگی ویدیو با سیگنال سینوسی نمایش بده.

۳.۳.۲.۲ فلوجارت این الگوریتم



شکل 11: فلوجارت الگوریتم آشکارسازی حرکت‌های نوسانی

۳.۳.۲.۳ پیاده‌سازی و آزمایش الگوریتم آشکارسازی حرکت‌های نوسانی

این الگوریتم در محیط پایتون پیاده‌سازی و با ویدیوی baby آزمایش شد. فرکانس تابع سینوسی نمونه استفاده شده در این آزمایش برابر یک هرتز و نتیجه آن به شکل زیر است.



شکل 12: تصویر همبستگی برای ویدیوی baby با فرکانس یک هرتز

در این شکل نقاطی از ویدیو که بیشترین همبستگی را دارند با مقادیر دامنه بیشتر یعنی رنگ‌های روشن‌تر مشخص شده است. با توجه به شکل بالا بیشترین میزان همبستگی در سینه نوزاد مشاهده می‌شود. این تصویر نشان می‌دهد که استخراج تغییرات حوزه زمان در ویدیو با روش همبستگی امکان‌پذیر است. گام بعدی، استفاده از این روش برای طراحی یک الگوریتم به منظور بزرگنمایی این حرکت‌ها است.

۳.۳.۳ الگوریتم اولیه روش همبستگی برای بزرگنمایی حرکت

ملاحظه شد که اندازه‌گیری حرکت‌های نوسانی در یک ویدیو با استفاده از تابع همبستگی قابل انجام است. برای بزرگنمایی این حرکت‌ها و پردازش یک ویدیوی جدید از روی ویدیوی ورودی نیز می‌توان از این روش استفاده کرد.

برای این منظور به جای یافتن دقیق حرکت‌ها در ویدیو، میزان همبستگی را برای ویدیو محاسبه شده و این‌ها به عنوان ضریب در سیگنال اصلی ویدیو ضرب می‌شود. در نتیجه در ویدیوی حاصل که هر کجا همبستگی وجود دارد دارای اطلاعات خواهد بود و همان پیکسل‌های اصلی ویدیو را نشان خواهد داد و هر کجایی که همبستگی وجود ندارد، مقدار پیکسل برابر صفر می‌شود. سپس می‌توان این را در یک ضریب بزرگنمایی ضرب کرده و با ویدیوی اصلی جمع کرد. بدین ترتیب ویدیوی حاصل در جاهایی که همبستگی وجود ندارد شبیه ویدیوی اصلی خواهد بود و در جاهایی که همبستگی وجود دارد، اطلاعات پیکسل‌ها بزرگنمایی می‌شوند.

۳.۳.۳.۱ شکل نوشتاری این الگوریتم

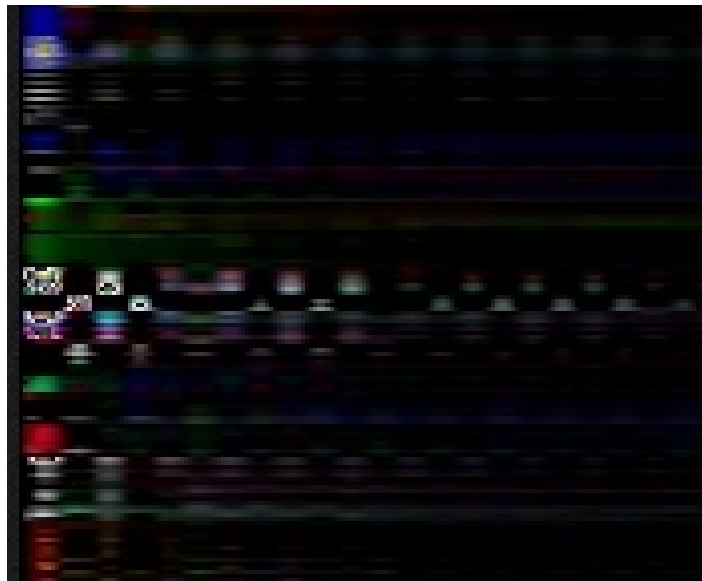
شکل نوشتاری الگوریتم اولیه بزرگنمایی ویدیو با روش محاسبه تابع همبستگی. نمایش ترتیبی الگوریتم به شکل زیر است:

۱. یک ویدیو را انتخاب کن و یکی یکی فریم‌های آن را بخوان.
۲. به روش **الگوریتم ۱** میزان همبستگی را از ابتدای ویدیو تا آخرین فریمی که به آن رسیده‌ای حساب کن تا فریم همبستگی حاصل شود.
۳. فریم همبستگی را ~~در آخرین فریمی که خوانده بودی ضرب کن~~ و آن را بزرگنمایی کن سپس با همان فریم جمع کن.
۴. این فریم را به عنوان خروجی نمایش بده.

در این الگوریتم برای محاسبه تابع همبستگی از روش مشابه الگوریتم ۱ استفاده می‌شود.

۳.۳.۳.۲ پیاده‌سازی و تست این الگوریتم

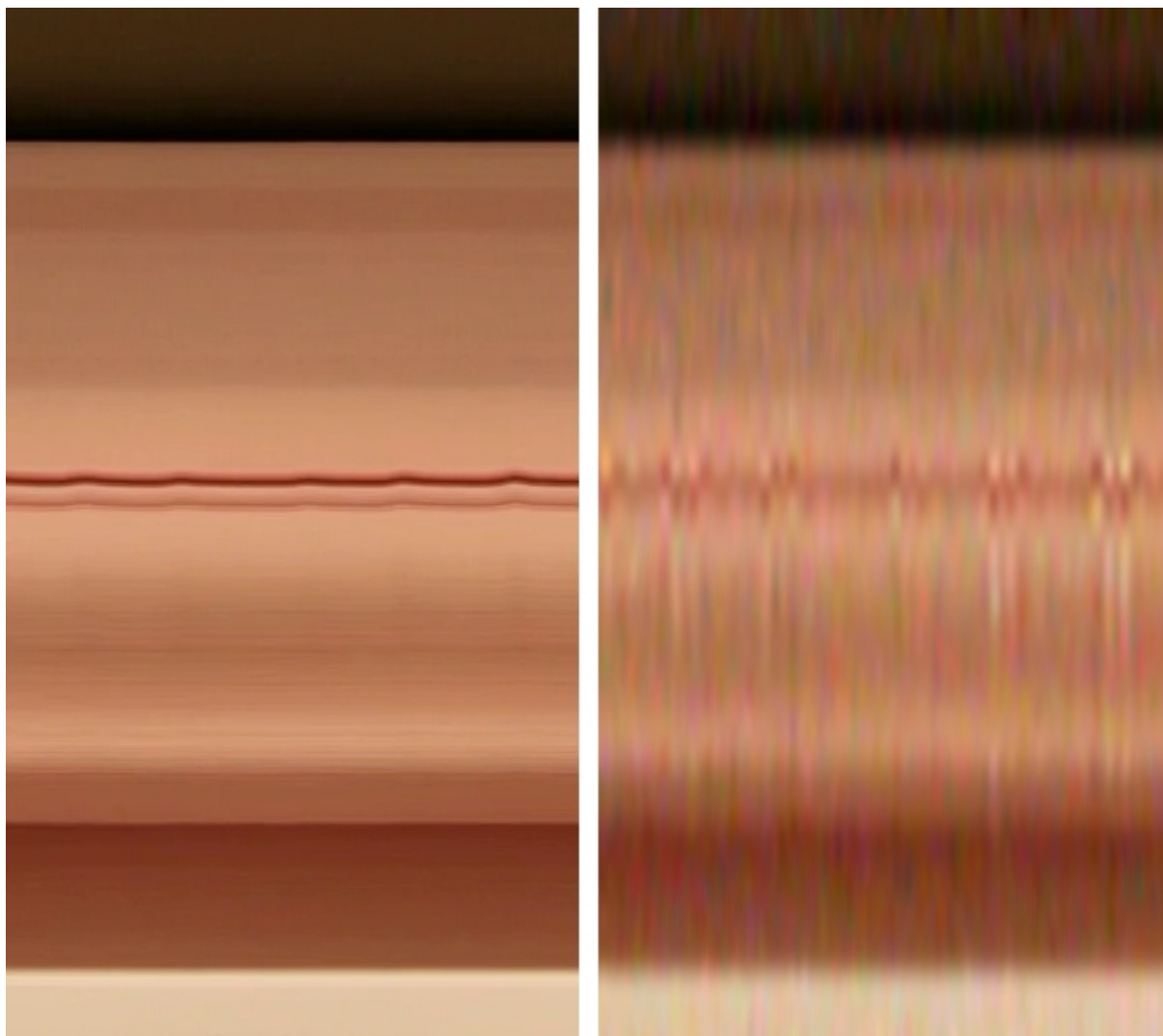
این الگوریتم در محیط پایتون پیاده‌سازی شد و نرم‌افزار حاصل زبان پایتون، روی ویدیوی baby تست شد. در نتیجه آن بزرگنمایی انجام شده و نتایج قابل مشاهده است.



شکل 13: میزان همبستگی در ویدیوی baby در گذر زمان.

خط افقی وسط شکل مربوط به سینه نوزاد است.

در شکل بالا می‌توان همبستگی با فرکانس انتخابی را در گذر زمان مشاهده کرد.



شکل 14: چپ: ویدیوی اصلی.

راست: ویدیوی خروجی.

در شکل بالا می‌تواند ویدیوی حاصل را مشاهده کرد. این شکل سیگنال حوزه زمان را برای پیکسل‌های روی یک خط عمودی از ویدیو، در جایی که سینه نوزاد در تصویر قرار دارد نشان می‌دهد.

۳.۳.۳.۳ محدودیت‌های روش پیشنهادی

الگوریتم پیشنهادی که همبستگی را برای همه فریم‌ها حساب می‌کند دو محدودیت مهم دارد:

۱. از کار افتادن بزرگنمایی در صورت وجود حرکت‌های بزرگ (تغییرات شدید) در ویدیو

۲. کاهش همبستگی در گذر زمان در صورت انتخاب غیر دقیق فرکانس سیگنال نمونه.

در ادامه به هر یک از این موارد به تفصیل پرداخته می‌شود.

۳.۳.۳.۳.۱ از کار افتادن بزرگنمایی در صورت وجود حرکت‌های بزرگ در ویدیو

همان‌طور که گفته شد محدودیت اول روش ارایه‌شده این است که با جابجا شدن موقیت دوربین و یا جابجایی جسم داخل ویدیو، برنامه دچار خطا می‌شود. دلیل آن این است که چنین حرکت‌هایی، تغییراتی بسیار بزرگ‌تر از حرکت‌های جزئی در ویدیو ایجاد می‌کنند. با توجه به این که بزرگنمایی حرکت نیازمند بیشتر شدن نسبت تغییرهای کوچک به دامنه خود ویدیو است، اگر تغییر بزرگی به ویدیو اضافه شود نسبت تغییرهای کوچک در بزرگنمایی نیز عدد کم‌تری خواهد بود. برای درک بهتر این مساله به عبارت ریاضی زیر توجه کنید.

$$f_1 = A + s_i \Rightarrow g_1 = A + \alpha \cdot s_i \Rightarrow S_o = \alpha \cdot s_i \Rightarrow P_1 = \frac{\alpha \cdot s_i}{A}$$

$$f_2 = A + b + s_i \Rightarrow g_2 = A = \alpha \cdot (b + s_i) = (A + \alpha \cdot b) + \alpha \cdot s_i \Rightarrow P_2 = \frac{\alpha \cdot s_i}{A + \alpha \cdot b}$$

$$\Rightarrow P_2 > P_1$$

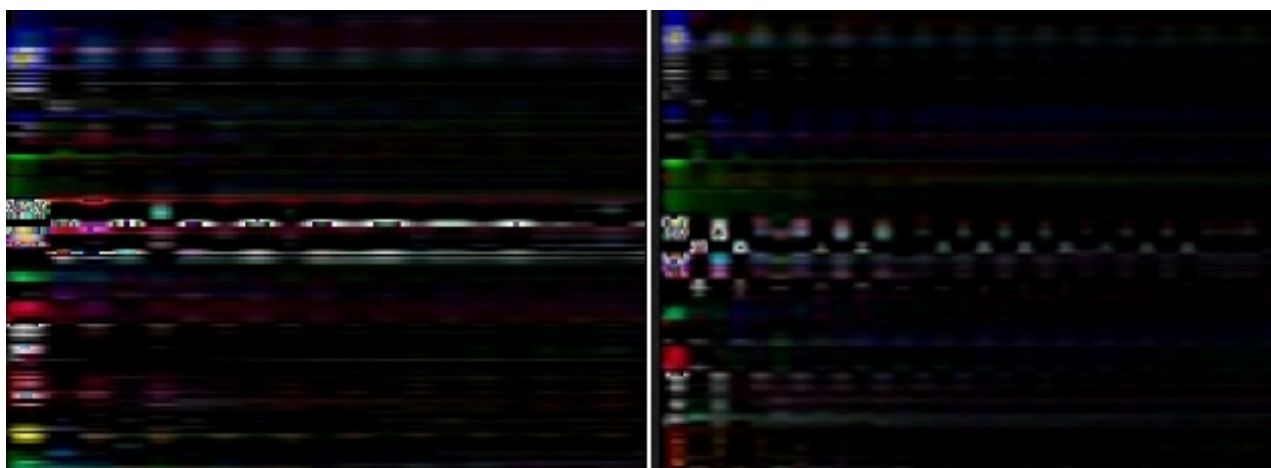
در این عبارت ریاضی S_o و s_i نمایانگر دامنه تغییرات اندک به ترتیب قبل و بعد از بزرگنمایی هستند و b نمایانگر دامنه تغییرات بزرگ است. همچنین A نمایانگر دامنه فریم میانگین ویدیو و P نمایانگر نسبت دامنه حرکت‌های کوچک به میانگین دامنه فریم در ویدیوی پردازش‌شده است. وقتی تنها یک تغییر کوچک داشته باشیم، بعد از بزرگنمایی به عبارت اول خواهیم رسید و وقتی یک حرکت بزرگ را در کنار تغییر کوچک داشته باشیم، عبارت دوم را خواهیم داشت. نسبت تغییرهای اندک در رابطه اول α برابر شده است در حالی که در رابطه دوم این نسبت کم‌تر است.

مشاهده می‌شود که در حالت دوم نسبت تغییرات کوچک به دامنه میزان کم‌تری است. حال اگر اندازه b بزرگ باشد، بعد از بزرگنمایی ممکن است دامنه یک پیکسل از بازه عددی $0 \sim 255$ خارج شود و مقدار حاصل را اشباع کند. در صورت اشباع شدن تغییرات کوچک کاملاً از بین خواهد رفت. چرا که پیکسل‌ها در سیستم ۸ بیتی مقداری صحیح و داخل این بازه به خود می‌گیرند. این باعث از بین رفتن اطلاعات تغییرات کوچک شده و بزرگنمایی از کار می‌افتد.

در صورت خارج شدن مقدار یک پیکسل از بازه، علاوه بر از بین رفتن اطلاعات، اثر نامطلوب نیز مشاهده خواهد شد.

۳.۳.۳.۳.۲ کاهش همبستگی در صورت انتخاب نشدن فرکانس مناسب سیگنال نمونه

وقتی فرکانس سیگنال نمونه از ابتدا به طور دقیق انتخاب نشده باشد، همبستگی کاهش پیدا می‌کند. چرا که روش محاسبه تابع همبستگی به گونه‌ای است که سعی می‌کند بیشترین شباهت را پیدا کند. در صورتی که فرکانس دو سیگنال به هم نزدیک باشند، حاصل همبستگی مقداری مثبت و بزرگ خواهد داشت در حالی که اگر فرکانس‌ها از هم دور باشند و تفاوت زیادی داشته باشند، باگذشت زمان تفاوت آن‌ها بیشتر ظاهر خواهد شد و مقدار همبستگی کاهش پیدا خواهد کرد.



الف

ب

شکل 15: الف: فرکانس درست انتخاب شده است.

ب: فرکانس درست انتخاب نشده است.

خطوط افقی در این شکل سیگنال‌های حوزه زمان ویدیوی baby هستند که خط روی سینه نوزاد یعنی جایی که بیشترین حرکت وجود دارد انتخاب شده است. در شکل الف همبستگی وجود دارد و آشکارسازی به درستی انجام شده است در حالی که در شکل ب با انتخاب غیر دقیق فرکانس سیگنال نمونه، با گذر زمان همبستگی کاهش پیدا می‌کند.

۳.۳.۴ استفاده از پنجره و بافر برای رفع محدودیت‌های روش پیشنهادی

همان‌طور که اشاره شد، محاسبه تابع همبستگی برای همه فریم‌های ویدیو دو محدودیت عمده دارد. برای برطرف کردن این محدودیت‌ها می‌توان به جای محاسبه میزان همبستگی از ابتدای ویدیو، این مقدار را در یک پنجره زمانی محدود حساب کرد. یعنی تعداد مشخصی از فریم‌ها جدا شده و یک ویدیوی مستقل فرض شود با این تفاوت که همبستگی با همان تابع سینوسی ویدیوی اصلی برای آن محاسبه می‌شود. نتیجه آن یک ویدیوی بزرگنمایی شده است که در صورت وجود تغییرات شدید در ویدیوی ورودی، تا زمانی که فریم‌های قبل از تغییر، در بافر قرار دارند چند فریم خطا خواهد داشت و بعد از خروج آن چند فریم از بافر، دوباره به حالت طبیعی خود باز می‌گردد و ویدیو مجدداً بزرگنمایی می‌شود.

۳.۳.۴.۱ طراحی الگوریتم

همان‌طور که گفته شد به منظور حذف محدودیت‌های اشاره شده، در الگوریتم پیشنهادی همبستگی در یک پنجره زمانی محدود محاسبه خواهد شد. برای این کار لازم است در برنامه ابتدا فریم‌ها بافر شوند تا به عنوان پنجره زمانی سیگنال‌های حوزه زمان از آن استفاده شود.

توصیف الگوریتم در ادامه آورده می‌شود.

۱. یک ویدیو را فریم به فریم بخوان

۲. فریم‌ها را در یک بافر ذخیره کن. به ازای هر فریم جدیدی که وارد می‌شود، قدیمی‌ترین فریم را از بافر حذف کن.

۳. همبستگی فریم‌های داخل بافر و سیگنال سینوسی نمونه با فرکانس دلخواه محاسبه کن تا یک فریم همبستگی حاصل شود.

۴. فریم همبستگی را در ثابت بزرگنمایی دلخواه ضرب کن و با آخرین فریمی که خوانده شده است جمع کن و آن را به عنوان فریم ویدیوی خروجی نمایش بده یا ذخیره کن.

البته نحوه تعیین فرکانس سیگنال سینوسی نمونه و مقدار ثابت بزرگنمایی بسته به محتوای ویدیوی مورد نظر برای بزرگنمایی تعیین می‌شود.

۳.۳.۴.۲ محاسبه همبستگی در یک پنجره زمانی محدود

با توجه به این که الگوریتم پیشنهادی در یک پنجره زمانی محدود اجرا می‌شود، میانگین فریم‌ها (مقدار DC) به ازای هر پنجره با دیگر متفاوت خواهد بود. پس رابطه ریاضی محاسبه تابع همبستگی روی یک پنجره به شکل زیر خواهد بود:

$$C(n, B) = \frac{1}{B} \cdot \sum_{N=n-B}^n (f[N] - f_{mean}[N]) \cdot \sin(\omega N)$$

$$f_{mean}[N] = \frac{1}{N} \cdot \sum_{n=N-B}^N f[n]$$

که در آن B اندازه بافر و $f_{mean}[N]$ میانگین فریم‌های داخل بافر است. با توجه به این که با ورود هر فریم جدید، بافر تغییر می‌کند، میانگین آن‌ها نیز تغییر می‌کند.

همچنین با فرض کوچک بودن تغییرات و با توجه به اینکه فریم‌های موجود در یک پنجره باهم همبستگی

زیادتری دارند، می‌توان فریم‌های بافر را با فریم آغازین تقریب زد: $f_{mean}[N] \approx f[N-B]$

پس می‌توان فرمول تابع همبستگی را برای کم‌هزینه‌تر شدن محاسبات این چنین بیان کرد:

$$C(n, B) = \frac{1}{B} \cdot \sum_{N=n-B}^n (f[N] - f[N-B]) \cdot \sin(\omega N)$$

یعنی برای حذف DC از تابع همبستگی نیازی به محاسبه میانگین نیست. بلکه می‌توان با کسر هر فریم از فریم آغازین پنجره، DC را با تقریب خوبی حذف کرد.

به بیان دیگر می‌توان همبستگی را چنین نیز محاسبه کرد:

$$C(n, B) = C(n) - C(n-B) - f_{n-B}$$

$$C(n) = \sum_{N=0}^n f[N] \cdot \sin(\omega N)$$

که در نتیجه محاسبه همبستگی پیچیدگی زمانی کمتری خواهد داشت.

۳.۳.۴.۲.۱ بهینه‌سازی الگوریتم محاسبه همبستگی روی بافر

اگر طول یک پنجره (معادل تعداد فریم‌های داخل بافر) برابر B باشد، الگوریتم فعلی محاسبه همبستگی از پیچیدگی زمانی از مرتبه $O(B)$ است.

محاسبه همبستگی در یک پنجره زمانی می‌تواند بهینه‌تر باشد. با یک بررسی ساده ریاضی می‌توان نتیجه گرفت که با داشتن جمع یک بافر از فریم‌های $n-B$ تا n می‌توان جمع فریم‌های $n+1-B$ تا $n+1$ را با دو عمل جمع و تفریق پیدا کرد. یعنی:

$$P(n) = f_n \cdot \sin(\omega(n))$$

$$C(n, B) = C(n) - C(n-B)$$

$$C(n+1, B) = C(n+1) - C(n+1-B)$$

$$C(n+1) = C(n) + P(n+1)$$

$$C(n+1-B) = C(n-B) + P(n+1-B)$$

$$C(n+1, B) = C(n) + P(n+1) - (C(n-B) + P(n+1-B))$$

$$C(n+1, B) - C(n, B) = P(n+1) - P(n+1-B)$$

در عبارت ریاضی بالا $P(n)$ حاصل ضرب یک فریم در مقدار سیگنال سینوسی نمونه است.

بدین ترتیب با داشتن همه مقادیر $P(n)$ مورد نیاز می‌توان $C(n)$ را از روی مقدار قبلی حساب کرد.

برای پیاده‌سازی بهتر این روش در برنامه‌نویسی، به جای نگهداری فریم‌های اصلی در بافر، مقادیر $P(n)$ در بافر نگهداری می‌شود. پس با بافری این چنین برنامه‌نویسی بهینه‌تر شده و پیچیدگی زمانی این الگوریتم به $O(1)$ کاهش پیدا می‌کند.

۳.۳.۴.۳ شبیه‌سازی الگوریتم محاسبه همبستگی در پنجره زمانی محدود

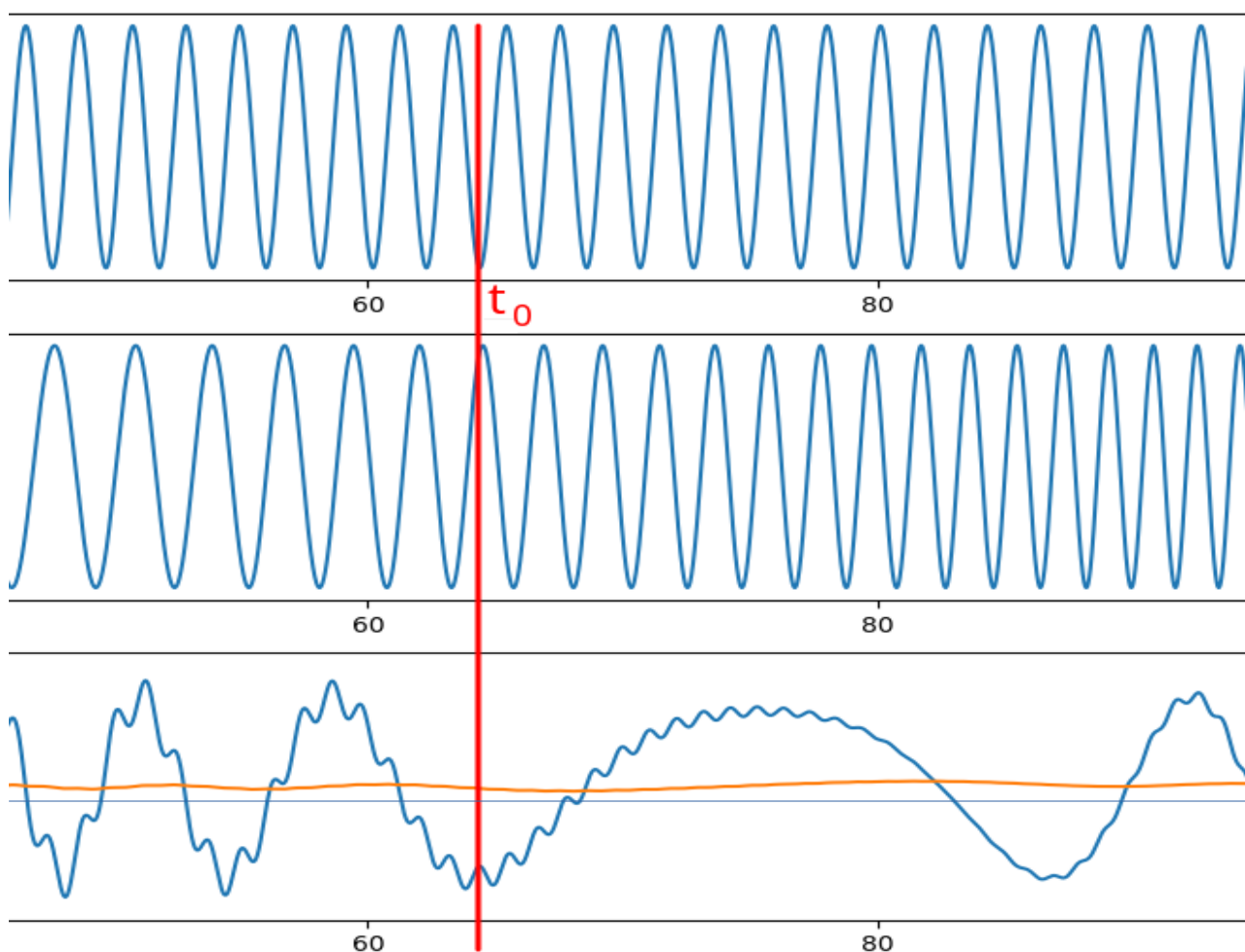
رویکرد جدید در محاسبه همبستگی باید بتواند دو محدودیت مطرح شده را برطرف کند. برای بررسی عملکرد این الگوریتم در برابر عواملی که مطرح شد، برای هر کدام یک شبیه‌سازی کامپیوتری انجام شد.

بررسی عملکرد این الگوریتم برای فرکانس‌های مختلف در ادامه آورده می‌شود.

اینجا هم ریفرنس علت chirp آورده شود.

با توجه به این که سیستم جدید طراحی شده از پنجره زمانی استفاده می‌کند، برای تست کردن پاسخ سیستم برای فرکانس‌های مختلف می‌توان از سیگنال chirp استفاده کرد. سیگنال chirp یک سیگنال سینوسی با فرکانس متغیر در حوزه زمان است. کاربرد این سیگنال برای مثال ما بررسی پاسخ سیستم به ازای فرکانس‌های مختلف بدون نیاز به تست‌های مکرر است. این سیگنال همچنین در بررسی پاسخ فرکانسی سیستم‌های مختلف کاربرد دارد و برای این کار معمولاً از chirp های خطی استفاده می‌شود و فرمول یک تابع chirp با تغییرات فرکانس خطی به شکل $\sin((f_0 + c \cdot t) \cdot t + \phi)$ است که پارامترهای آن فاز و فرکانس اولیه و نرخ افزایش فرکانس هستند.

توضیحات شکل زیر شکل



شکل 16: بالا: سیگنال سینوسی به عنوان ورودی

وسط: سیگنال chirp به عنوانه سیگنال نمونه برای محاسبه همبستگی.

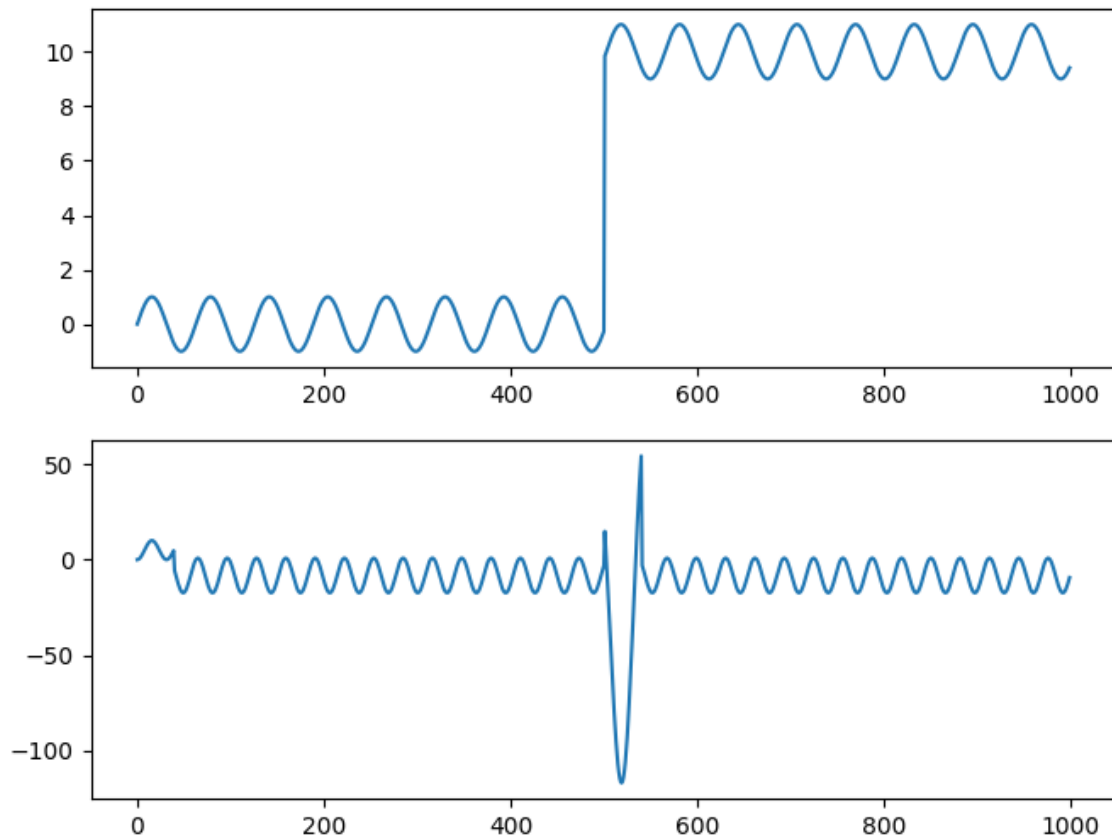
پایین خط نازجی: حاصل همبستگی از ابتدا تا انتها (عددی نزدیک به صفر)

پایین آبی: حاصل همبستگی در یک پنجره (مقدار آن بعضی جاها منفی و بعضی جاها مثبت است).

طبق منحنی بالا مشاهده می شود که در این روش برای فرکانس های نزدیک به فرکانس سیگنال ورودی، پاسخ همبستگی روی پنجره عددی مثبت و بزرگ خواهد شد. نکته دیگر در لحظه $t=t_0$ است که به دلیل اختلاف فاز ۱۸۰ درجه در دو تابع، حاصل همبستگی در نقطه اوج منفی خود قرار گرفته است.

برای بررسی عمل کرد این الگوریتم در تغییرات شدید سیگنال ورودی یک شبیه سازی کامپیوتری انجام شد. برای این کار ابتدا برنامه ای که امکان محاسبه همبستگی روی یک پنجره زمانی محدود را فراهم کند در محیط پایتون نوشته شد و ورودی مناسب به آن داده شد.

ورودی سیستم، سیگنالی به شکل مجموع یک سیگنال سینوسی با یک سیگنال پله است. سیگنال سینوسی نمایانگر حرکت نوسانی کوچک در سیگنال حوزه زمان ویدیو است و تابع پله نمایانگر تغییرات شدید در ویدیو است.



شکل 17: بررسی خروجی الگوریتم همبستگی دارای پنجره برای یک سیگنال با تغییرات شدید

بالا: سیگنال ورودی. این سیگنال به شکل مجموع یک سینوسی با یک پله است.

پایین: خروجی همبستگی

طبق این شکل مشاهده می‌شود که بعد از تغییرات شدید در سیگنال ورودی، سیستم تا مدت کمی خطا خواهد داشت و بعد از آن به حالت درست خود بازخواهدگشت.

۳.۳.۴.۴ محدودیت‌های روش محاسبه همبستگی با پنجره محدود زمانی

محدودیت اصلی این روش عدم نمایش صحیح حرکت‌ها در پنجره‌های کوچک است.

در این روش، تابع همبستگی فرصت کافی برای رسیدن به حالت پایدار خود نخواهد داشت چرا که اندازه پنجره کوچک است. در نتیجه یک حرکت نوسانی نیز خواهد داشت. حرکت‌ها همچنان قابل تشخیص هستند اما به خاطر کوچک بودن پنجره‌ها، دامنه DC تابع همبستگی برای حرکت‌ها کوچک است و در نتیجه بیشتر حرکت‌های تابع همبستگی دیده خواهد شد که فرکانسی برابر با حرکت اصلی ندارند بلکه فرکانس پایه آن‌ها دو برابر فرکانس حرکت اصلی است.

بنابراین در روش همبستگی که از پنجره زمانی محدود استفاده می‌کند، در صورت کوچک بودن اندازه پنجره، حرکت‌ها همچنان مشخص است اما شکل آن‌ها و جهت حرکت آن‌ها به شکل دقیق تشخیص داده نخواهد شد.

۳.۳.۴.۴.۱ شبیه‌سازی و بررسی محدودیت‌های روش همبستگی روی پنجره زمانی محدود

برای بررسی بیشتر محدودیت‌های این روش شبیه‌سازی کامپیوتری انجام شد. یک برنامه پایتون برای آن نوشته شد و سپس سیگنال ورودی به آن داده شد. سیستم بزرگنمایی با دو اندازه پنجره متفاوت طراحی شد که در یکی از آن‌ها اندازه پنجره ۱۰ برابر دیگری در نظر گرفته شد تا تفاوت این دو قابل مشاهده باشد. سیگنال ورودی جمع یک تابع سینوسی با یک تابع پله است که تابع سینوسی نمایانگر حرکت‌های کوچک در سیگنال حوزه زمان ویدئو و تابع پله نمایانگر تغییرات شدید در ویدئو است.

برای محاسبه همبستگی، ابتدا در سیگنال ورودی مقدار DC داخل یک پنجره حذف می‌شود و سپس همبستگی این سیگنال با یک سیگنال سینوسی هم‌فرکانس با خودش روی یک پنجره زمانی محدود محاسبه و نتیجه به شکل یک نمودار ترسیم می‌شود.

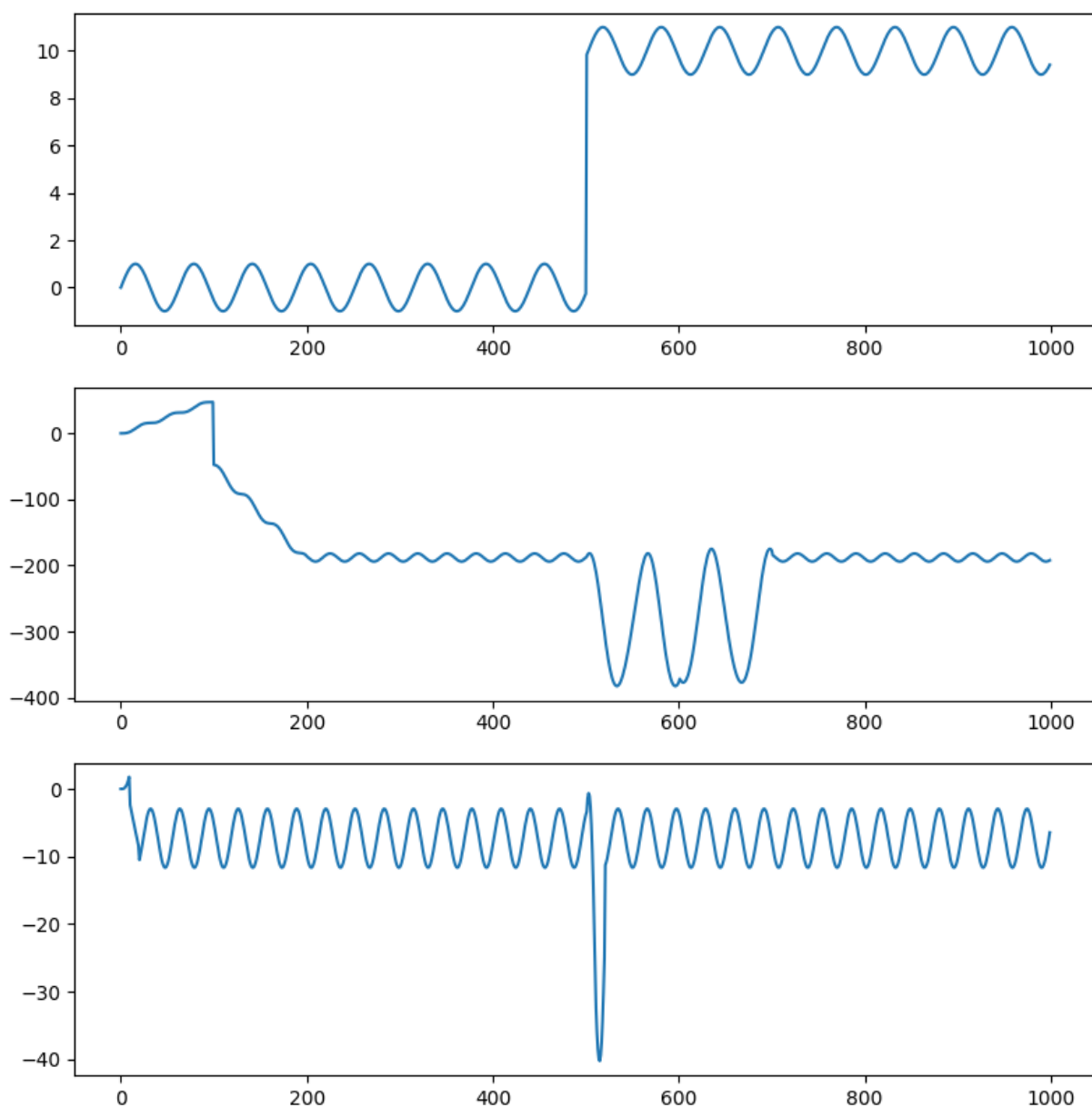
این بخش را خودم بررسی کنم.

در حالتی که اندازه پنجره کوچک‌تر است، بزرگنمایی سریع‌تر شروع می‌شود و در صورت وجود تغییرات شدید در ویدئو، اثر آن به سرعت از بین می‌رود و بزرگنمایی به کار خود ادامه می‌دهد. در این حالت تابع

بزرگ‌نمایی به حالت پایدار خود نزدیک نمی‌شود چرا که زمان کافی برای این کار را ندارد. در نتیجه حرکتی نوسانی با فرکانس دوبرابر سیگنال ورودی در آن مشاهده می‌شود و میزان دامنه DC خود سیگنال کم‌تر است.

در حالی که اندازه پنجره بزرگتر است، بزرگنمایی دیرتر آغاز می‌شود چرا که بافر هنوز پر نشده است و

در حالی که اندازه پنجره بزرگتر است، بزرگنمایی دیرتر آغاز می‌شود چرا که بافر هنوز پر نشده است و

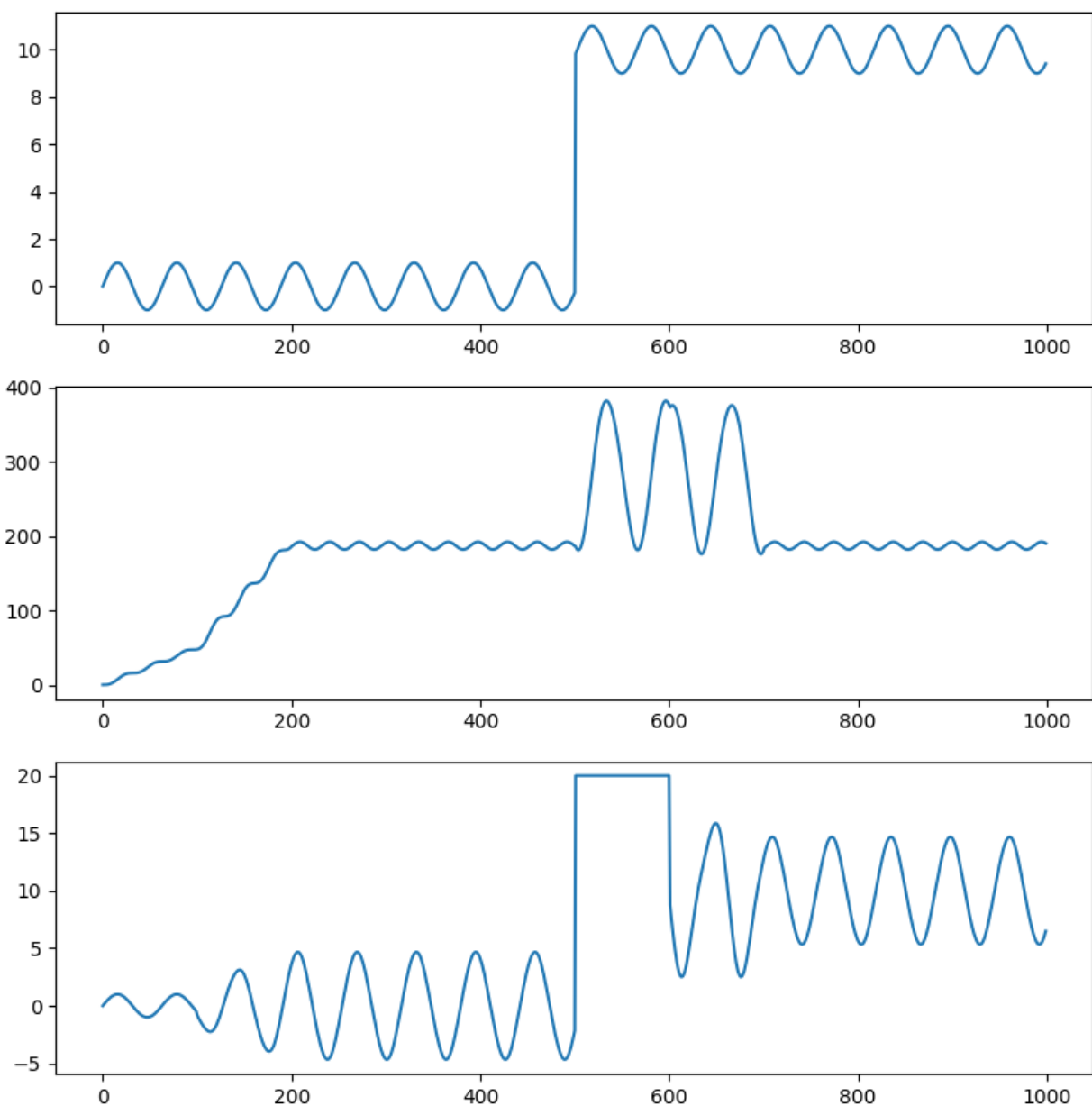


شکل 18: مقایسه اندازه پنجره در روش همبستگی

بالا: سیگنال ورودی

وسط و پایین: خروجی سیستم. در شکل وسط اندازه پنجره ۱۰ برابر شکل پایین انتخاب شده است. در شکل پایین دامنه سیگنال کمتر است و بیشتر حرکت‌های نوسانی دیده می‌شود. در شکل وسط دامنه سیگنال زیاد است و حرکت‌های نوسانی کمتر دیده می‌شوند.

در شکل وسط به خاطر بزرگ بودن اندازه پنجره، در آغاز ویدیو و در حین تغییرات شدید، مدت بیشتری خطا مشاهده می‌شود درحالی که در شکل پایین مدت زمان این خطا کمتر است.



شکل 19: سیگنال بزرگنمایی شده با روش همبستگی روی پنجره

بالا: سیگنال ورودی

وسط: سیگنال خروجی همبستگی با سینوسی نمونه. اندازه پنجره به حد کافی بزرگ است به طوری که مقدار DC offset بسیار بیشتر از مقدار تناوب است. در میانه این سیگنال به خاطر تغییرات شدید در سیگنال ورودی، خطا دیده می شود که زمان باقی ماندن این خطا روی سیگنال به اندازه طول پنجره است. پایین: سیگنال بزرگنمایی شده با ضرب سیگنال روی پنجره در تابع همبستگی و جمع سیگنال اصلی با آن.

در این شکل مشخص شده است که به خاطر بزرگ بودن اندازه پنجره، شروع بزرگنمایی تا زمان پر شدن بافر از فریم ها، بزرگنمایی دیده نخواهد شد و در زمان بروز تغییرات شدید، سیگنال خروجی اشباع خواهد شد.

بنابراین می‌توان در صورت کم‌تر بودن دامنه بخش متناوب سیگنال، از آن برای بزرگنمایی سیگنال اصلی استفاده کرد.

می‌توان با روش‌های ریاضی و پردازش سیگنال مختلف، مقدار نوسان را در این سیگنال کم‌تر هم کرد یا از بین برد. یک روش برای این کار محاسبه میانه این سیگنال روی یک پنجره زمانی است. یعنی استفاده از فیلتر میانه (Moving Median Filter)

یک برنامه پایتون برای شبیه‌سازی این نیز نوشته شد. این برنامه تابع همبستگی سیگنال با یک سیگنال سینوسی نمونه را روی یک پنجره با اندازه کوچک محاسبه کرده و فیلتر میانه را روی آن اعمال می‌کند. سپس با ضرب کردن آن در سیگنال اصلی، به سیگنالی که اطلاعات متناوب سیگنال اصلی را در خود دارد می‌رسد. با ضرب کردن آن در یک ضریب بزرگنمایی و جمع آن به سیگنال اصلی، به سیگنال بزرگنمایی می‌رسد.

۳.۳.۴.۵ پیاده‌سازی و تست الگوریتم همبستگی روی پنجره زمانی محدود

الگوریتم طراحی شده در بخش قبل که با استفاده از پنجره زمانی محدود تابع همبستگی را حساب کرده و ویدیو را بزرگنمایی می‌کند، در پایتون پیاده‌سازی شد و با ویدیوهای مختلف بررسی شد.

در ویدیوی نتیجه حرکت‌ها آشکار شده و دو محدودیت قبلی برطرف شده بودند:

۱. نیازی به تنظیم دقیق فرکانس سیگنال سینوسی نمونه برای آشکارسازی حرکت نبود.

۲. وقتی ویدیو در وسط تغییر شدیدی می‌کرد، تا زمان خارج شدن فریم‌های قدیمی از بافر، خطا مشاهده می‌شد اما بعد از این بزرگنمایی به حالت عادی خود بازمی‌گشت و به درستی انجام می‌شد.

ولی محدودیت جدید عدم نمایش دقیق فرکانس حرکت موجود در محتوای ویدیو قابل رویت بود.

۳.۳.۴.۶ محدودیت‌های روش بزرگنمایی ویدیو با استفاده از الگوریتم محاسبه

همبستگی روی پنجره زمانی محدود

وجود نویزهای تصویر باعث افت کیفیت در ویدیوی خروجی می‌شود. این نویزها که بخش‌های قبلی به آن‌ها اشاره شد، حرکت‌های مطلوب ما نیستند ولی اندازه آن‌ها به حدی بزرگ است که می‌تواند همراه با حرکت‌های اصلی ویدیو بزرگنمایی شده و در خروجی اثر نامطلوب بگذارد. البته برخلاف حرکت‌های جسم داخل محتوای ویدیو این نویزها پایدار نیستند و دایم تغییر می‌کنند به همین دلیل حرکت‌های ویدیو همچنان قابل تشخیص است اما با این حال در ویدیوی خروجی اثرات نامطلوبی حاصل می‌شود که ناخواسته‌اند.

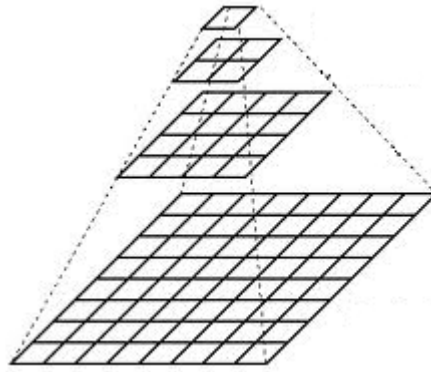
۳.۳.۵ استفاده از هرم در روش همبستگی بزرگنمایی ویدیو برای رفع محدودیت

نویز در بزرگنمایی

همانطور که گفته شد محدودیت این روش بزرگنمایی نویزهای مکانی فریم‌ها در ویدیو است. برای این کار مشابه روش دیفرانسیلی، می‌توان با اعمال یک فیلتر گوسی طرز کار سیستم را بهبود داد. اعمال یک فیلتر گوسی باعث از بین رفتن فرکانس‌های بالای تصویر می‌شود و مانند یک فیلتر پایین‌گذر عمل می‌کند. اما برای اینکه بتوان فرکانس‌های مکانی را به شکل کامل‌تری تقسیم کرد، می‌توان از هرم‌های گوسی و لاپلاسی استفاده کرد.

هرم گوسی یک فریم، مجموعه‌ای از فریم‌هایی است که فرکانس‌های مختلف یک فریم را در بر دارند. هر یک از طبقات این هرم مثل یک فیلتر پایین‌گذر روی فریم اصلی عمل می‌کند.

طبقه اول هرم گوسی با تصویر ورودی یکسان است. برای ساختن هر طبقه از هرم، طبقه قبلی فیلتر شده و سپس اندازه طول و عرض آن نصف می‌شود. یعنی ابتدا یک فیلتر گوسی با کرنل اندازه ۵ روی طبقه قبلی اعمال شده و سپس با نمونه‌کاهی (downsampling) تصویر فیلترشده به تصویری با طول و عرض نصف، مرتبه بعدی ایجاد می‌شود. با تکرار این کار می‌توان یک هرم گوسی مطابق شکل زیر را ایجاد کرد.^{۳۴}



شکل 21: هرم گوسی

لایه‌های هرم از پایین به بالا شماره‌گذاری می‌شوند.

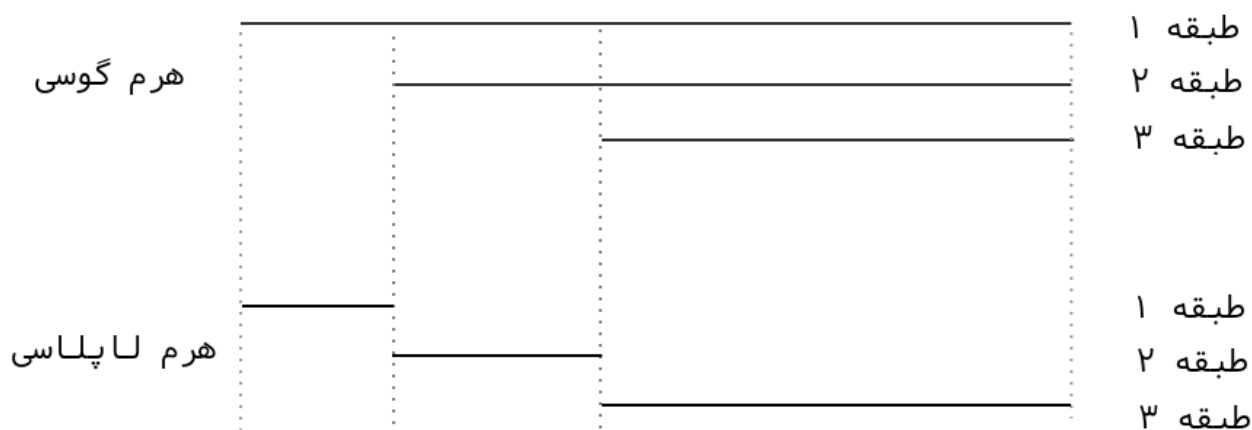
همچنین ماتریس کرنل (kernel) ۵ در ۵ فیلتر گوسی را می‌توان در عبارت زیر مشاهده کرد.

$$\frac{1}{16} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

شکل 22: یک نمونه کرنل فیلتر گوسی

هرم لاپلاسی نیز مشابه هرم گوسی است با این تفاوت که تصویر را به باندهای فرکانسی مختلفی افراز می‌کند که با یکدیگر هم‌پوشانی ندارند. درواقع به جای فیلتر پایین‌گذر، از فیلترهای میانگذر در آن استفاده شده‌است. برای ساخت هرم لاپلاسی لازم است اختلاف طبقات متوالی هرم گوسی از یکدیگر حساب شود

تا طبقات هرم لاپلاسی ایجاد شود. تقسیم‌بندی باندهای فرکانسی این دو نوع هرم را در تصویر زیر مشاهده می‌کنید.



شکل 23: مقایسه تقسیم فرکانسی در طبقه‌های مختلف هرم‌های گوسی و لاپلاسی

محور افقی: فرکانس مکانی است.

طبق شکل بالا مشاهده می‌شود که هرم گوسی مثل یک مجموعه فیلتر پایین‌گذر عمل می‌کند در حالی که هرم لاپلاسی مثل یک مجموعه فیلتر میان‌گذر است.

۳.۳.۶ الگوریتم نهایی روش همبستگی در بزرگنمایی ویدیو

بعد از بررسی‌ها و بهبودهای انجام شده، الگوریتم نهایی طراحی شد. این الگوریتم با معماری بلادرنج طراحی شد که جزئیات آن به شرح زیر است.

برای بزرگ‌نمایی ویدیو به روش اویلری همبستگی، ابتدا با استفاده از یک هرم لاپلاسی فریم‌های ویدیو به باندهای فرکانس مکانی مختلف تقسیم می‌شود. سپس هر یک از طبقات هرم در تابع سینوسی نمونه ضرب می‌شود و هرم حاصل در یک بافر ذخیره می‌شود. پس از آن طبقات متناظر هرم‌های داخل باف با هم جمع می‌شود تا هرم همبستگی حاصل شود. سپس هر طبقه این هرم در ضریب بزرگ‌نمایی مربوط به

خود ضرب شده و با طبقه متناظر هرم فریم خوانده شده جمع می‌شود تا هرم بزرگ‌نمایی شده حاصل شود. در نهایت با فرو ریختن هرم حاصل، فریم ویدیوی خروجی ساخته می‌شود. با تکرار این عمل برای هر فریم ویدیوی ورودی، ویدیوی بزرگ‌نمایی شده حاصل خواهد شد.

۳.۳.۶.۱ توصیف و شبه‌کد الگوریتم نهایی همبستگی

توصیف الگوریتم نهایی همبستگی در زیر آورده شده است که تکه‌هایی از شبه‌کد نیز در آن نوشته شده است.

1. یک ویدیو را فریم به فریم بخوان.

```
frame_now = read_new_frame()
```

2. هرم لاپلاسی فریم را حساب کن.

```
pyramid_now = Laplacian_pyramid(frame_now)
```

3. با توجه به نرخ فریم بر ثانیه و شماره فریم، زمان فریم را محاسبه کرده و از روی آن مقدار تابع سینوسی نمونه را در آن زمان حساب کن.

```
T = number_of_frame / frame_rate  
sin_value = sin( $\omega$  * T)
```

4. هر طبقه هرم را در مقدار محاسبه شده تابع سینوسی نمونه ضرب کن. نتیجه را که یک هرم است در یک بافر ذخیره کن. با ذخیره کردن این هرم در بافر، قدیمی‌ترین هرم موجود در بافر را از بافر حذف کن.

```
pyramid_correlation_now = pyramid_now * sin_value  
buffer.add(pyramid_correlation_now)  
buffer.remove(buffer.oldest)
```

5. میانگین مقادیر داخل بافر را حساب کن تا هرم همبستگی حاصل شود

```
correlation_pyramid = sum(buffer) / buffer_size
```

6. طبقات هرم همبستگی را با ضریب‌های مشخص بزرگ‌نمایی کن. یعنی آن‌ها را در ثابت‌های بزرگ‌نمایی باندهای فرکانس مکانی ضرب کن.

```
pyramid_correlation_amplified = correlation_pyramid *  
spatial_frequency_coefficients
```

7. هرم حاصل را با هرم فریم خوانده شده جمع کن. تا هرم بزرگ‌نمایی شده حاصل شود.

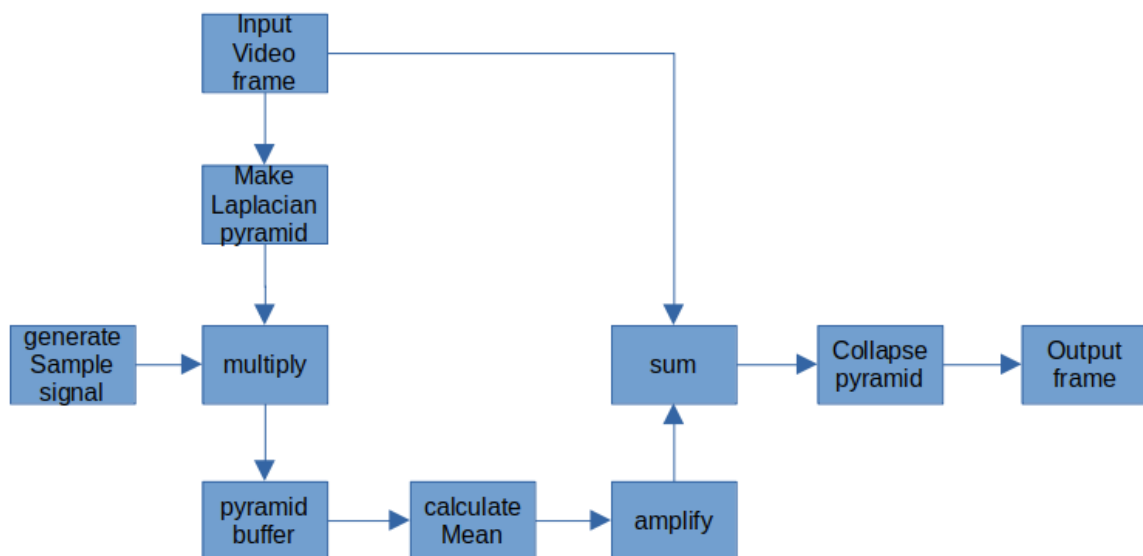
```
pyramid_magnified = pyramid_correlation_amplified + pyramid_now
```

8. هرم بزرگنمایی شده را فرو بریز تا یک فریم حاصل شود و آن را به عنوان خروجی نمایش بده.

```
result_frame = laplacian_pyramid_collapse(pyramid_magnified)
show(result_frame)
```

۳.۳.۶.۲ بلوک دیاگرام الگوریتم نهایی همبستگی

بلوک دیاگرام این الگوریتم به شکل زیر است.



شکل 24: بلوک دیاگرام الگوریتم نهایی همبستگی

توضیح خیلی خلاصه و جمع و جور اینجا آورده شود.

۳.۳.۶.۳ پیاده سازی الگوریتم نهایی همبستگی

این الگوریتم در پایتون پیاده سازی و آزمایش های مختلف روی ویدیوها (مشخصات و تعداد ویدیوها) شد.

پلتفرم کامل آورده شود.

یک سری نتایج اینجا آورده شود.

۳.۴ آزمایش‌ها

برای بررسی صحت عمل‌کرد الگوریتم نهایی همبستگی آزمایش‌های مختلفی طراحی و انجام شد.

تست‌های مختلفی روی این الگوریتم‌های طراحی شده و برنامه‌های نوشته‌شده انجام شد.

۳.۵ مقایسه الگوریتم‌ها

در مرحله پایانی طراحی و توسعه الگوریتم‌ها نیاز به آزمایش‌های نهایی و مقایسه است. الگوریتم‌ها در این

پایان‌نامه از سه دیدگاه باهم مقایسه خواهند شد:

۱. مقایسه سرعت و کارایی و میزان حافظه مورد نیاز.

۲. مقایسه نویز و اثرات نامطلوب با روش SSIM

۳. مقایسه تاخیر و مقاوم بودن در برابر تغییرات شدید.

برای موارد ۱ و ۲ نیاز به تست‌های فراوانی وجود دارد درحالی که مورد ۳ در طراحی الگوریتم، مورد ارزیابی

قرار گرفته‌است.

در ادامه هر یک از این موارد مقایسه‌ها شرح داده خواهد شد.

۳.۵.۱ مقایسه سرعت و حافظه مورد نیاز

معیار مقایسه سرعت اجرای الگوریتم‌ها زمان خواهد بود. برای مقایسه پیچیدگی حافظه، معیار حجم

حافظه مصرف‌شده توسط برنامه خواهد بود. در صورتی که معیار ما برای سنجش عمل‌کرد برنامه‌ها زمان

و حافظه باشد، آزمایش نرم‌افزارها باید در شرایط یکسان انجام شود به این معنی که از منابع سخت‌افزاری

یکسانی بهره ببرند، روی سیستم‌عامل یکسانی اجرا شوند و از نسخه یکسان نرم‌افزارها و کتابخانه‌های

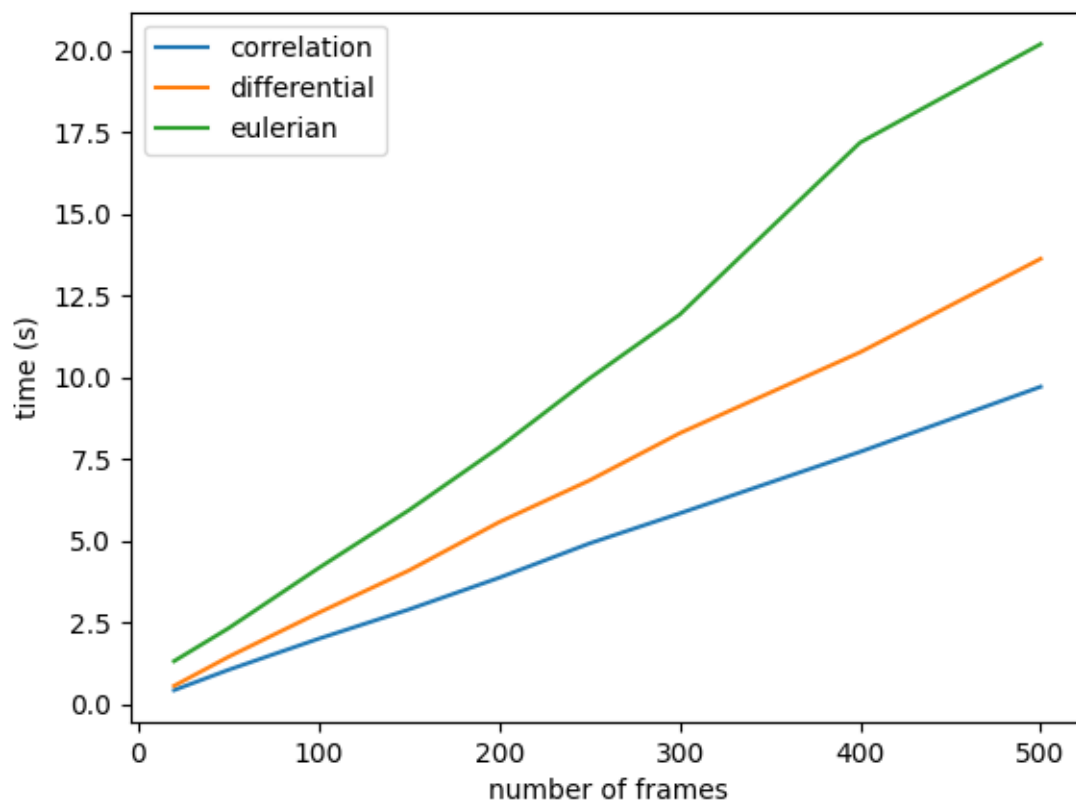
موجود استفاده کنند.

آزمایش‌ها برای سه الگوریتم دیفرانسیلی، اولیری خطی و همبستگی با ویدیوی ورودی یکسان، اندازه بافر

یکسان و عمق یکسان هرم لاپلاسی انجام شد. برای اندازه‌گیری زمان از ساعت سیستم موجود در

کتابخانه time در پایتون استفاده شد. متغیر آزمایش‌ها تعداد فریم‌های ویدیو بود. بعد از انجام

آزمایش‌ها به شکل خودکار، مدت زمان اجرای برنامه‌ها اندازه‌گیری شد و روی یک نمودار ترسیم شد که در شکل زیر قابل مشاهده است.



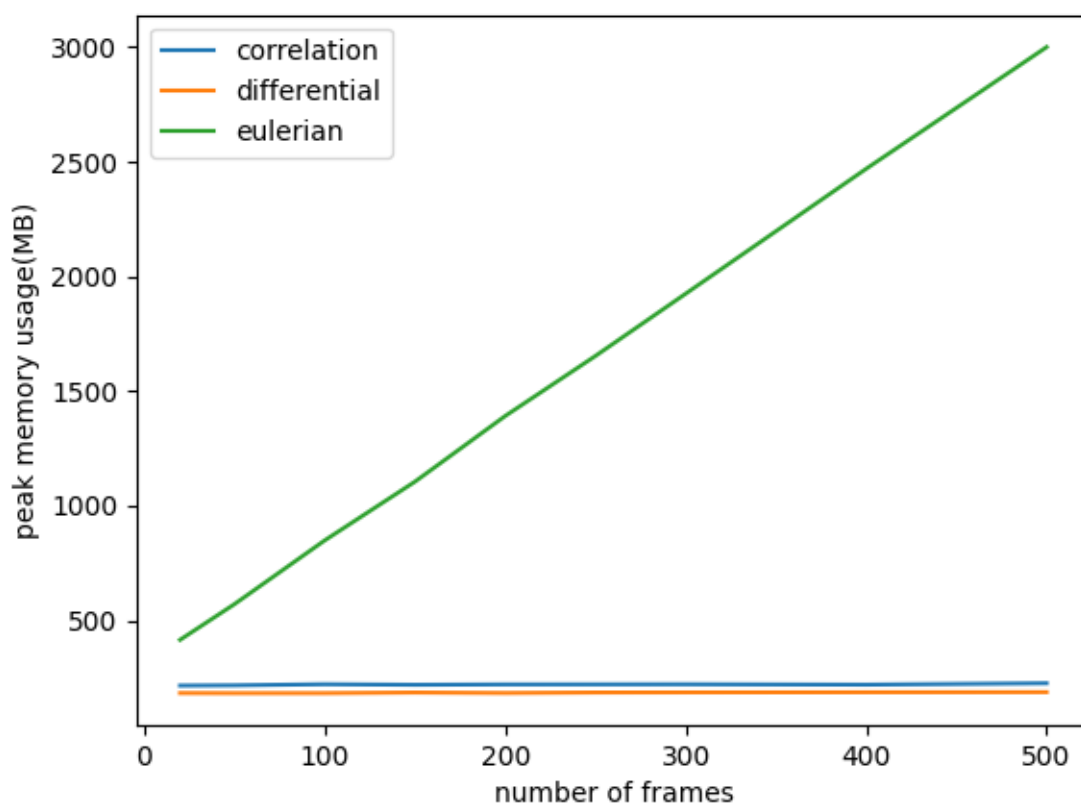
شکل 25: مقایسه زمان اجرای الگوریتم‌ها بر اساس تعداد فریم ویدیو

سپس آزمایش‌ها برای اندازه‌گیری میزان حافظه مصرفی انجام شد. مشابه آزمایش قبلی، از ویدیوی یکسان به عنوان ورودی استفاده شد و تعداد فریم‌های ورودی به عنوان پارامتر متغیر به برنامه‌ها داده شد. برخلاف اندازه‌گیری زمان که با گرفتن ساعت سیستم به راحتی قابل انجام است، برای سنجش حافظه مصرفی نیاز به برقراری ارتباط با سیستم‌عامل وجود دارد. بنابراین یک اسکریپت لینوکسی برای سنجش حافظه نوشته شد تا با برقراری ارتباط با سیستم‌عامل، میزان حافظه مصرفی پردازش مورد نظر را به دست آورد و سپس آن را در اختیار پایتون قرار دهد. نکته مهمی که در اندازه‌گیری حافظه وجود دارد این است که میزان حافظه مصرفی برنامه در طی زمان اجرا تغییر می‌کند و برنامه کامپیوتری نوشته شده باید بتواند

مقدار آن را در هر لحظه پایش کند تا بتواند تصویر درستی از این مقدار ارایه دهد. نکته دیگر اینکه بیشینه حافظه اشغال شده توسط برنامه برای ما اهمیت دارد چرا که بر خلاف زمان که می توان مقدار متوسط آن را در نظر گرفت، در مورد حافظه اگر برنامه بیشتر از حافظه موجود سیستم نیاز داشته باشد، امکان اجرا نخواهد داشت و از کار می افتد.

یک برنامه پایتون متناسب با نکات گفته شده ساخته شد تا آزمایش ها را اجرا کند و نتایج را ذخیره کند. سپس نتایج این آزمایش ها بر روی منحنی هایی ترسیم شد که در شکل زیر قابل مشاهده است.

برای مشخص شدن بهتر، یک نمودار تنها برای آبی و نارنجی هم بیاورم یا اینکه نمودار را لگاریتمی ترسیم کنم.



شکل 26: مقایسه بیشینه حافظه مصرفی برای تعداد فریم های ورودی در الگوریتم ها

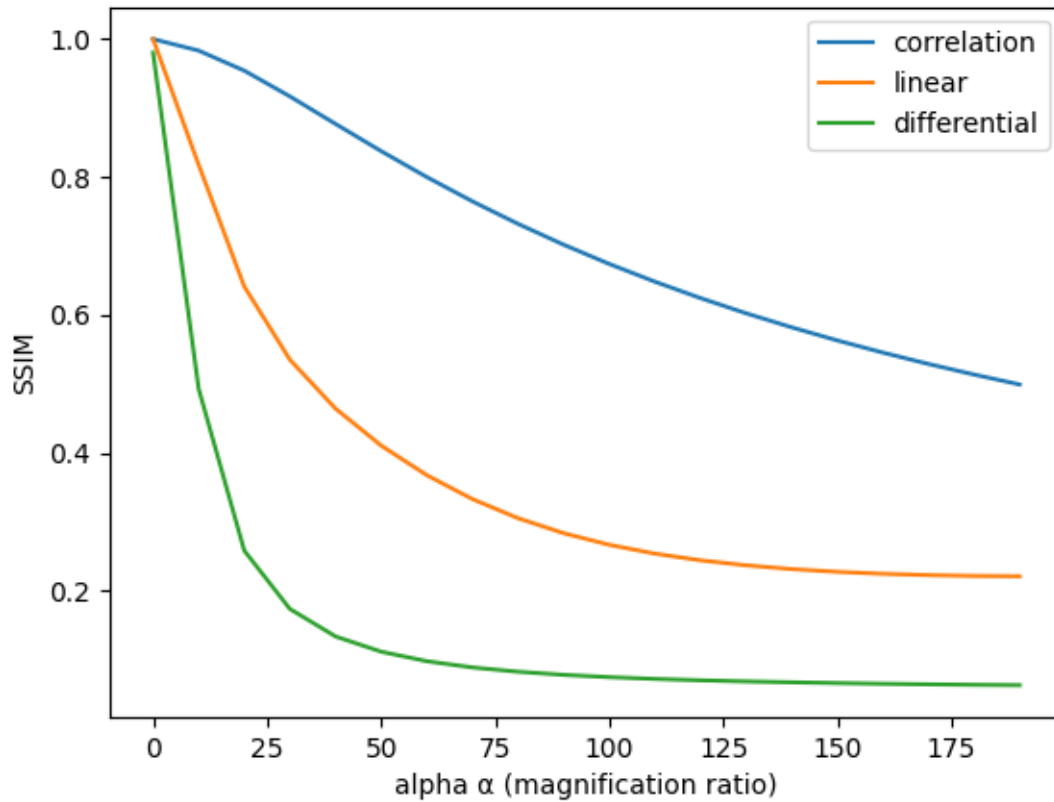
۳.۵.۲ مقایسه کیفیت ویدیوی بزرگنمایی شده با معیار SSIM

اولاً: کلی‌تر همیشه روش‌های subjective و objective و روش subjective به صورت real time نیست بنابراین به روش دیگر انجام می‌شود. و روش objective به سه روش FR, NR, reduced Reference (با مرجع محدود)

یکی از مهم‌ترین پارامترها برای مقایسه روش‌های بزرگنمایی ویدیو، کیفیت خروجی است. برای اندازه‌گیری کیفیت تصویر خروجی روش‌های مختلفی وجود دارند که به طور کلی به روش‌های متکی بر مرجع (full reference)، بدون مرجع (no reference) و با مرجع محدود (reduced reference) تقسیم می‌شوند. در روش‌های متکی بر مرجع با مقایسه یک تصویر مرجع با تصویر خروجی، کیفیت از روی میزان شباهت این‌ها به یک‌دیگر اندازه‌گیری می‌شود. در حالی که روش‌های بدون مرجع کیفیت یک تصویر را بدون مقایسه با تصویر دیگری می‌سنجند. در روش‌های با مرجع محدود، تنها بخشی از مرجع یا اطلاعات محدودی راجع به مرجع برای ارزیابی کیفیت تصویر استفاده می‌شود. هر یک از این دسته‌ها کاربردهای خود را دارد. دسته‌ای که مناسب اندازه‌گیری کیفیت ویدیوی خروجی در زمینه بزرگنمایی ویدیو است روش‌های متکی بر مرجع هستند چرا که ویدیوی اصلی موجود است و ویدیوی خروجی به دلیل بزرگنمایی نویز و ایجاد آثار نامطلوب ممکن است افت کیفیت داشته باشد.

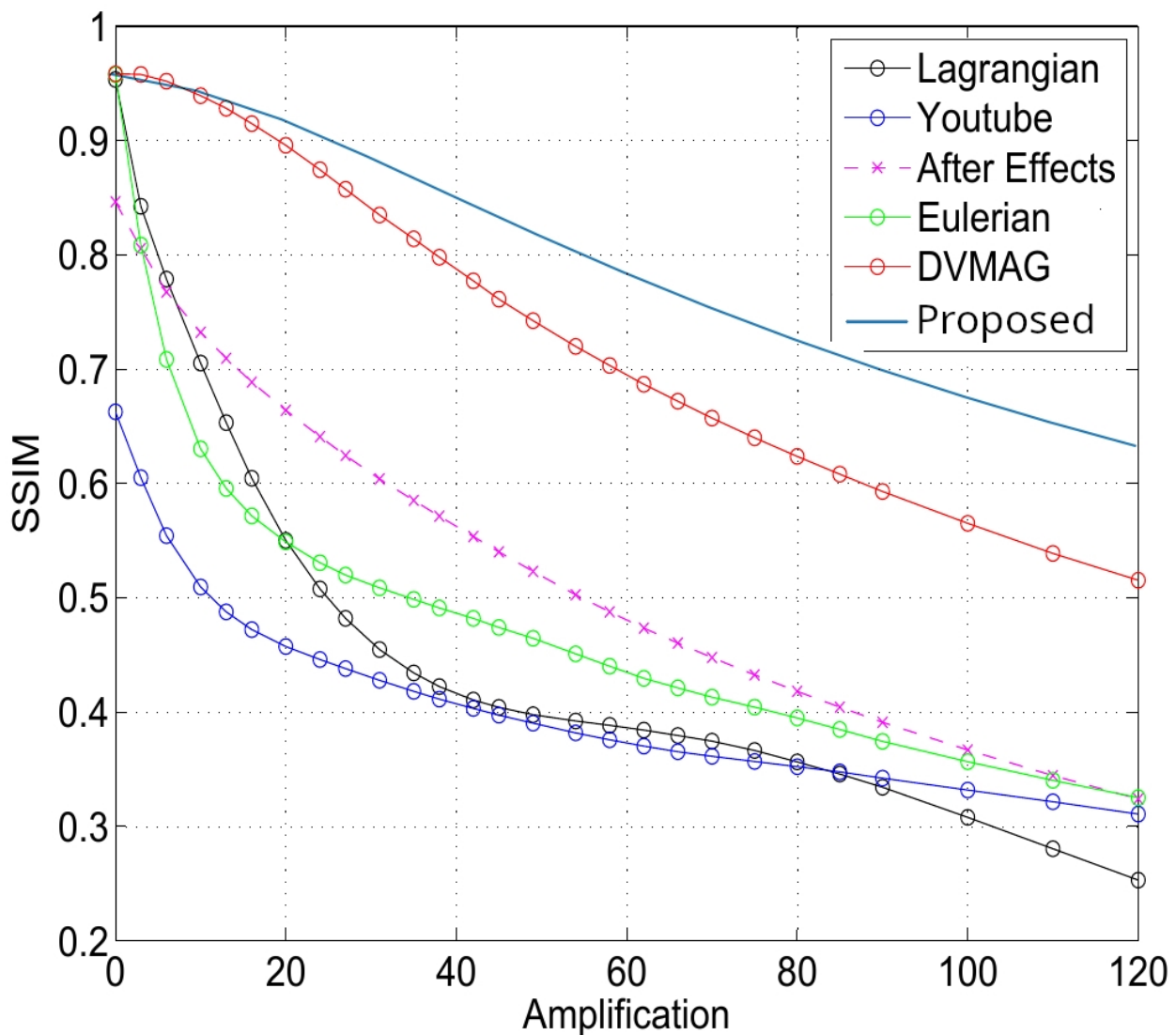
یکی از روش‌های سنجش متکی بر مرجع روش شباهت ساختاری (Structural Similarity) SSIM (Measurement Index) است^{۳۵} که در پژوهش‌های پیشین بزرگنمایی هم ویدیو استفاده شده است. در این پژوهش نیز از همین روش استفاده می‌شود و نتایج آن با پژوهش‌های پیشین مقایسه خواهد شد. برای ارزیابی کیفیت ویدیوی بزرگنمایی شده خروجی، یک برنامه پایتون نوشته شد تا آزمایش‌ها را به شکل خودکار انجام دهد و پس از اندازه‌گیری کیفیت به روش SSIM نتایج را ثبت کند. در این ارزیابی‌ها از ویدیوی baby استفاده شد. پارامتر متغیر در این ارزیابی‌ها، ثابت بزرگنمایی (α) است. برای بزرگنمایی برابر صفر، ویدیوی خروجی با ویدیوی ورودی از نظر ریاضی یکسان خواهد بود و تصویر بیشترین میزان کیفیت را خواهد داشت. اما با افزایش بزرگنمایی مقادیر SSIM ممکن است تغییر کنند. این برنامه برای هر سه الگوریتم اجرا شد و نتایج آزمایش‌ها در شکل زیر قابل مشاهده است.

در شکل اسم‌ها را جوری بگذارم که معلوم شود مال من است. مثلا: proposed differential و proposed correlation



شکل 27: مقایسه SSIM روش‌های پیاده‌شده بر حسب مقادیر مختلف α

همچنین در شکل زیر می‌توانید منحنی روش ارایه‌شده را در کنار روش‌های پژوهش‌های پیشین ببینید.



شکل 28: مقایسه SSIM برای الگوریتم همبستگی با الگوریتم‌های پیشین

۳.۵.۳ مقایسه تأخیر شروع و مقاوم بودن در برابر تغییرات شدید

همان‌طور که در این فصل اشاره شد، پارامتر تأخیر شروع رابطه مستقیم با اندازه بافر دارد. اندازه‌های بزرگ‌تر بافر تأخیر بیشتری خواهند داشت و در صورت وجود تغییرات شدید، زمان بیشتری در خروجی خطا خواهد بود. از طرفی بیشتر کردن اندازه بافر مزیت‌های افزایش کیفیت SSIM ویدیوی خروجی و افزایش دقت در آشکارسازی فرکانس‌ها را خواهد داشت. بنابراین اندازه بافر، بسته به کاربرد قابل انتخاب است.

۳.۶ بررسی نتایج آزمایش‌ها

در این پژوهش دو الگوریتم دیفرانسیلی و همبستگی با رویکرد اوپلری طراحی و پیاده‌سازی شدند و با همراه الگوریتم اوپلری خطی مقایسه شدند. برای این مقایسه‌ها معیارهای زمان اجرا، حافظه مصرفی، کیفیت ویدیوی خروجی و تأخیر مورد بررسی قرار گرفت.

الگوریتم همبستگی در مقایسه با الگوریتم اوپلری زمان اجرای پایین‌تری دارد. این زمان اجرا ممکن است تا ۲ برابر پایین‌تر باشد. در نتیجه برای پیاده‌سازی در سیستم‌های با پردازشگر ضعیف‌تر مانند سیستم‌های توکار یا سیستم‌های موبایل بهینه‌تر عمل می‌کند.

الگوریتم همبستگی در مقایسه با الگوریتم اوپلری خطی حافظه بسیار کم‌تری اشغال می‌کند. میزان حافظه اشغال‌شده در روش همبستگی تنها تابعی از اندازه بافر است و مستقل از تعداد فریم‌های ویدیو خواهد بود. در حالی که در روش اوپلری خطی مقدار حافظه مصرف‌شده به خاطر اعمال فیلتر رابطه خطی با تعداد فریم‌ها دارد و در نتیجه برای ویدیوهای بزرگ کارایی خود را از دست می‌دهد. برای غلبه بر این محدودیت روش اوپلری خطی استفاده از فیلترهای FIR با پیاده‌سازی‌های بهینه‌تر پیشنهاد می‌شود.

کیفیت الگوریتم همبستگی از روش اوپلری خطی بالاتر است. به طوری که برای مقادیر بزرگ‌نمایی با α های بالاتر از ۵۰ روش اوپلری خطی به خاطر اثرات مخرب زیاد در ویدیوی خروجی کارکرد خود را از دست می‌دهد درحالی که روش همبستگی می‌تواند برای بزرگ‌نمایی‌های تا ۲۰۰ برابر کارایی داشته‌باشد. همچنین این روش برای بزرگ‌نمایی‌های پایین‌تر از ۲۰ برابر، SSIM بالای ۹۰ درصد دارد که یعنی از کیفیت بیشتری نسبت به روش اوپلری برخوردار است.

~~روش اوپلری تک‌کاناله برای ۱۰ فریم از ویدیوی baby با اندازه ۵۴۴x۹۶۰ پیکسل و مرتبه ۷ احتیاج به ۱.۹۹ گیگابایت رم داشت.~~

~~تنها خواندن ویدیوی baby با همه فریم‌ها ۳.۸ گیگابایت در حافظه جا اشغال می‌کند. این درحالی است که حجم خود ویدیوی غیر فشرده از این عدد کمتر است. یعنی کم‌تر از یک گیگابایت.~~

۳.۷ اثرات نامطلوب ناشی از فشرده‌سازی در حوزه زمان و حوزه مکان

ویدیوهای ورودی برنامه اغلب با کدگذاری‌های با تلاف مثل h.264 ذخیره می‌شوند. این کدگذاری‌ها جزئیاتی که برای چشم انسان قابل تشخیص نباشند را از ویدیو حذف می‌کنند تا حجم آن کاهش پیدا کند. این جزئیات هم در حوزه مکان و هم در حوزه زمان وجود دارند.

برای مثال در کدگذاری h.264 برای فشرده‌سازی در حوزه مکان، هر فریم ابتدا به بلوک‌های کوچکی تقسیم شده و سپس داخل هر بلوک اطلاعات با فرکانس بالای مکانی حذف می‌شود که برای این کار از تبدیل DCT (تبدیل کسینوسی گسسته) استفاده می‌شود. اگر در یک بلوک توزیع توان در فرکانس‌های پایین بیشتر باشد، میزان بیشتری از فرکانس‌های بالا حذف خواهد شد چرا که این کار اطلاعات زیادی را از بین نمی‌برد. الگوریتم‌های بزرگ‌نمایی ویدیو برای تشخیص تغییرات کوچک نیاز به این اطلاعات جزئی دارند و از بین رفتن آن‌ها در فریم‌ها باعث کاهش کیفیت ویدیوی خروجی و ظاهر شدن اثری به اسم «آثار نامطلوب فشرده‌سازی» (compression artifact) در خروجی خواهد شد. آثار نامطلوب اثر مخربی روی تصویر است که در اثر پردازش‌های رایانه‌ای روی تصویر ایجاد می‌شود.

در کدگذاری h.264 برای فشرده‌سازی در حوزه زمان ابتدا فریم‌ها به دسته‌های چندتایی (مثلاً ۳۲ تایی) به اسم GoP (Group of Picture) تقسیم می‌شوند و سپس با زدن برچسب‌های i ، p و b به فریم‌ها، اختلاف آن‌ها با یک‌دیگر به جای خود فریم اصلی برای ذخیره استفاده می‌شود. هر فریم اختلاف داخل GoP در حوزه مکان فشرده‌سازی شده و سپس ذخیره می‌شود. فریم‌های داخل یک GoP به خاطر فشرده‌سازی با تلاف و حذف جزئیاتی که در تفاوت فریم‌ها وجود دارد، همبستگی زیادی با هم دارند. همچنین هر فریم از یک GoP با فریمی از یک GoP دیگر در ویدیو همبستگی کم‌تری دارد چرا که جزئیاتی که در اختلاف آن با فریم‌های گروه خودش دارد حذف شده‌است درحالی که نسبت به گروه‌های بعدی و قبلی کاملاً مستقل کد شده‌اند. این موضوع باعث از دست رفتن یک سری اطلاعات در حوزه زمان و همچنین به وجود آمدن یک فرکانس ناخواسته زمانی با دوره‌ای به اندازه طول GoP در ویدیو خواهد شد. در بزرگ‌نمایی ویدیو، تقویت این تفاوت شدید مصنوعی بین هر GoP نیز باعث ایجاد اثر نامطلوب در خروجی خواهد شد.

۳.۷.۱ بررسی در حوزه زمان

برای بررسی این پدیده تست‌هایی با ویدیوهای مختلف انجام شد. در بعضی از این ویدیوها نتایج بهتر از بقیه بودند. یکی از پدیده‌هایی که در زمان تست با ویدیوهای خروجی دوربین موبایل مشهود بود، تفاوت زیاد بین هر ۳۰ فریم بود. بررسی‌های بیشتر نشان داد که این تفاوت به خاطر فشرده‌سازی AVC با اندازه GOP برابر ۳۰ فریم است.



شکل 29: نمودار خط مکانی - زمان. محور عمودی زمان است. تفاوت‌های زیاد بین هر یک ثانیه از ویدیو دیده می‌شود.

۳.۷.۲ حوزه مکان

در بخش‌های مختلف تصویر از جمله بخش‌های با جزئیات‌تر، قسمت‌هایی به اندازه اغلب ۱۶ در ۱۶ پیکسل از تصویر خراب می‌شود.



شکل 30: یک فریم از خروجی. اطلاعات فرکانس بالا در مربع‌هایی اغلب به اندازه ۱۶ در ۱۶ از بین رفته است و بزرگ‌نمایی آن‌ها باعث برجسته‌تر شدن آثار نامطلوب فشرده‌سازی شده‌است.

۴ نتیجه‌گیری و پیشنهادهایی برای ادامه پژوهش

بزرگ‌نمایی ویدیو با روش‌های مختلفی قابل انجام است. هر یک از این روش‌ها مزایا و محدودیت‌های خود را دارند.
نتیجه‌گیری خودم.

با این سه روش انجام شد. روش اول اینطوری بهتر است، روش دوم اینطوری بدتر است و غیره.

۴.۱ نتیجه‌گیری

بزرگ‌نمایی ویدیو با روش‌های مختلفی قابل انجام است که در این پژوهش سه روش اولری یعنی دیفرانسیلی، خطی و همبستگی مورد بررسی قرار گرفت. روش بلادرنگ دیفرانسیلی به خاطر سادگی مفاهیم می‌تواند روش مناسبی برای درک اولیه از بزرگ‌نمایی ویدیو باشد. این روش سرعت اجرای بالا و مصرف حافظه پایینی دارد اما به خاطر پایین بودن کیفیت ویدیوی خروجی، کاربردهای کم‌تری خواهد داشت. روش اولری خطی از طرف دیگر پیچیدگی بیشتری دارد و درک و پیاده‌سازی آن دشوارتر است. این روش سرعت اجرای پایین‌تر و مصرف حافظه بیشتری دارد که برای سامانه‌های ضعیف‌تری مثل سامانه‌های توکار، پیاده‌سازی فعلی قابل استفاده نیست و باید روش‌های پیاده‌سازی جدیدتری برای آن طراحی شود. در این روش کیفیت ویدیوی خروجی قابل قبول است اما در بزرگ‌نمایی‌های زیاد کارایی خود را از دست می‌دهد. در نهایت روش همبستگی از مزایای هر دو روش برخوردار است. الگوریتم این روش طبق مفاهیم پایه مهندسی برق مخابرات طراحی شده و به این خاطر درک الگوریتم آن آسانتر است. همچنین به خاطر اینکه نیازی به طراحی فیلتر ندارد، پیاده‌سازی آن آسانتر است. این روش سرعت بالا و مصرف حافظه پایینی دارد بنابراین می‌تواند بر روی سیستم‌های با مصرف پایین‌تر عمل‌کرد مناسبی داشته باشد. محدودیت این روش دقت پایین در آشکارسازی فرکانس‌ها است که برای برطرف ساختن آن می‌توان از اندازه‌های بزرگ‌تر بافر استفاده کرد.

۴.۲ پیشنهادهایی برای ادامه کار و پژوهش

پیاده‌سازی بهینه الگوریتم‌ها با زبان‌های کمپایل‌شده و سریعی مثل C می‌تواند قدم بعدی باشد. استفاده بیشتر از منابع سیستم نیز برای چنین الگوریتم‌هایی بسیار مفید خواهد بود. این استفاده‌ها می‌تواند شامل استفاده از thread های مختلف پردازنده مرکزی یا استفاده از کارت گرافیک برای تسریع عملیات محاسباتی باشد.

پیاده‌سازی روش اوپلری به شکل بلادرنگ (real time) نیز برای کاربرد صنعتی از این سودمندتر است.

۴.۲.۱.۱ حذف فرکانس‌های بالای ویدیو

همانطور که گفته شد با استفاده از یک هرم لاپلاسی می‌توان فرکانس‌های مکانی را از هم جدا کرد و جداگانه هر یک را بزرگنمایی کرد. حال اگر فرکانس‌های بالا را از خود ویدیوی نیز حذف کنیم و تنها فرکانس‌های پایین را نگه داریم، بعد از بزرگنمایی به ویدیویی خواهیم رسید که جزئیات آن حذف شده است اما حرکت‌ها در آن باقی مانده است. در واقع نسبت بخش‌های متحرک ویدیو به جزئیات آن افزایش پیدا می‌کند. در نتیجه با کاهش جزئیات تصویر، حرکت‌های کوچک بیشتر دیده می‌شوند.

۵ پیوست: ویدیوهای آزمایش

در متن ارجاع داده شود.




در پژوهش پیش‌رو برای آزمودن الگوریتم‌ها و روش‌های پیاده‌شده از ویدیوهای مختلفی استفاده شد. این مجموعه شامل ویدیوهای گروه بزرگ‌نمایی ویدئو دانشگاه MIT است که در مقاله‌های مختلف بزرگ‌نمایی ویدئو برای ارزیابی و مقایسه روش‌ها از آن‌ها استفاده می‌شود.

این مجموعه شامل ۹ عدد ویدیوی استاندارد برای تست است که با کدگذاری H.264 و فرمت MPEG4 ذخیره شده‌اند و در در زمان نگارش این نوشته از آدرس وب زیر قابل دسترسی است:

<http://people.csail.mit.edu/mrub/vidmag/#videos>

نام و تصاویر بندانگشتی این ویدیوها در جدول زیر قابل مشاهده است.

Table 1: نام و تصاویر بندانگشتی ویدیوهای مجموعه آزمایش استفاده شده در این پژوهش

نام ویدیو	تصویر بندانگشتی ویدیو
baby	
face	
guitar	



camera



Shadow



Baby2



Face2



subway



wrist

۶ پیوست: کدها

کدهای نوشته شده در این پژوهش در این بخش هستند.

برای اجرای کدها نیاز به نصب مفسر cpython نسخه ۳.۶ یا بالاتر به همراه کتابخانه های numpy، scipy، openCV و matplotlib است. این کدها در چند توزیع مختلف لینوکس از جمله ، KDE neon MX linux و fedora و همچنین روی یکی از توزیع های ویندوز نسخه 10 تست شده اند. اما به غیر از این ها روی هر دستگاه و سیستم عاملی که از نرم افزارهای ذکر شده پشتیبانی کند قابل اجرا است.

در ادامه، این کدها به تفکیک روش آمده است.

۶.۱ روش دیفرانسیلی

```
# استفاده خواهد شد opencv از کتابخانه .
import cv2

# for working with matrices
import numpy as np

# for making delay while showing frames
from time import sleep

# for cli
import os

# for cli
import sys
import matplotlib.pyplot as plt

# for storing time signal
import pickle

from time import time

from pyramids import pyramid_make, pyramid_rendr

# do not let pixel values of image get out of their limit
def guard_image(image:np.ndarray):

# any pixel value bigger than 255 will be cut down to 255
image[image > 255] = 255

# any pixel value smaller than 0 will be 0
image[image < 0] = 0

# magnify video. differential method
def differential(input_file_name, alpha,
number_of_frames, spatial_frequency_coefficients, buffer_size,
number_of_jump_frames=0, time_signal_x=0, time_signal_y=0, show_video=True,
pyramid_type='gaussian'):
    """
# magnify video with differential algorithm

- `input_file_name`: name of the file
- `alpha`: magnification ratio
```

- `number_of_frames`: number of video frames to read and magnify.
- `spatial_frequency_coefficients`
- `spatial_frequency_coefficients`: coefficients of summision for pyramid.
- `buffer size`: size of buffer for calculating correlation.
- `number_of_jump_frames`: number of initial frames to skip. this is useful when your time of interest is not the start of video. also useful when you want to cut video into pieces and magnify each seperately.
- `time_signal_x` and `time_signal_y`: the pixel on image to extract time signal from. three time signals will be extracted. one from y'th row. one from x's column. other from pixel (x,y)
- `show_video` : if set to True, a video will be shown while rendering. results will always be stored in ./results folder.

```
"""
```

```
spatial_coeff = spatial_frequency_coefficients
number_of_pyramid_levels = len(spatial_coeff)
```

```
# print_mode = 'none'
# print_mode = 'time'
print_mode = 'number'
```

```
# if you provide "camera" as input filename, camera of system will be used indtead of file.
```

```
# if you want to use camera
if input_file_name == 'camera':
```

```
# open camera as a video file
videoFile = cv2.VideoCapture(0)
# if a plain filename is provided
else:
```

```
# open videofile.
videoFile = cv2.VideoCapture(input_file_name)
```

```
# تاخیر مناسب بین فریم‌ها تا ویدیو با سرعت عادی پخش شود
# به خاطر وجود تاخیر در حین پردازش، این تاخیر نصف شده‌است.
frame_sleep_duration = 1.0/60 / 2
```

```
# get frame rate of the video
# (frames per second. it is 30 for most standard videos)
video_frame_rate = videoFile.get(cv2.CAP_PROP_FPS)
```

```
# if we need to skip some frames, we can just read a bunch of frames and simply toss them away
for i in range(number_of_jump_frames):
```

```

# read a frame and toss it out!
# these frames are going to be skipped. so let it go!
_, frame_now = videoFile.read()

# خواندن فریم اول و تعریف متغیرهای اولیه
_, frame_0 = videoFile.read()

# frame is basically a matrix of pixels. each pixel has 3 variables for each corresponding
channel.
# the variables are 8-bit unsigned integer numbers between 0 and 255 (uint8)
# for further processings we need floating point numbers.
# so here we convert the variable types from what it is into float32
frame_0_float = np.float32(frame_0)

# get frame size.
frame_height, frame_width, number_of_channels = np.shape(frame_0_float)

just_magnify_image = np.zeros((frame_height, number_of_frames, 3), np.uint8)

# get input file name
just_file_name = os.path.basename(input_file_name)

time_signal = []

# derivative output will be stored in this file
derivative_outfile =
cv2.VideoWriter(f'results/out_differential_derivative_{just_file_name}.avi',
cv2.VideoWriter_fourcc(*'MJPG'), video_frame_rate, (frame_width, frame_height))

# just amplify output will be stored in this file
amplify_outfile = cv2.VideoWriter(f'results/out_differential_amplify_{just_file_name}.avi',
cv2.VideoWriter_fourcc(*'MJPG'), video_frame_rate, (frame_width, frame_height))
original_outfile = cv2.VideoWriter(f'results/orig_differential_{just_file_name}.avi',
cv2.VideoWriter_fourcc(*'MJPG'), video_frame_rate, (frame_width, frame_height))

# extracted temporal signals from video will be stored in these arrays.

# define a temporal signal of zeros
image_signal_out_x = np.zeros((frame_height, number_of_frames, 3), np.uint8)

# define a temporal signal of zeros
image_signal_out_y = np.zeros((number_of_frames, frame_width, 3), np.uint8)

# define a temporal signal of zeros
image_signal_original_x = np.zeros((frame_height, number_of_frames, 3), np.uint8)

```

```

# define a temporal signal of zeros
image_signal_original_y = np.zeros((number_of_frames, frame_width, 3), np.uint8)

# define a pyramid of black frames.
# this is for defining pyramid buffer.
# default value is zero.
pyramid_zero = np.array([

np.zeros(
(frame_height//2**(j), frame_width//2**(j), number_of_channels),
dtype=np.float32)

for j in range(number_of_pyramid_levels)
], dtype=np.ndarray)

# define a buffer of pyramids for storing correlation pyramids.
pyramid_buffer = np.array([
pyramid_zero.copy()
for i in range(buffer_size)
], dtype=object)

# this counter is for buffer to work
# it is index of buffer where the latest frame is stored
buffer_count = 0

t0 = time()

frame_previous = frame_0_float
print()

for i in range(number_of_frames):

# next frame on buffer. it is also oldest frame stored on buffer that is not popped out yet
buffer_count_next = (buffer_count + 1) % buffer_size

_, frame_now = videoFile.read()

if frame_now is None:
break
frame_now_float = np.float32(frame_now)
pyramid_now = pyramid_make(frame_now_float, number_of_pyramid_levels, pyramid_type)
diff_pyramid = pyramid_now - pyramid_buffer[buffer_count_next]

```

```

diff_frame = pyramid_rendr(diff_pyramid, spatial_coeff=spatial_coeff,
pyramid_type=pyramid_type)
# pyramid_differentially_amplified = pyramid_now + diff_pyramid * alpha
# amplified_rendered = pyramid_rendr(pyramid_differentially_amplified,
spatial_coeff=spatial_coeff, pyramid_type=pyramid_type)
amplified_rendered = frame_now + diff_frame * alpha
guard_image(amplified_rendered)

pyramid_buffer[buffer_count] = pyramid_now

# diff = frame_now_float - frame_previous
# frame_differentially_amplified = frame_now_float + diff * alpha
# guard_image(frame_differentially_amplified)

# pyramid_just_amplified = pyramid_now
# frame_just_amplified = frame_now_float * alpha - frame_0_float * (alpha - 1)
# guard_image(frame_just_amplified)

if show_video==True:

cv2.imshow('original', frame_now)
cv2.imshow('out', np.uint8(amplified_rendered))
# cv2.imshow('just magnify', np.uint8(frame_differentially_amplified))
# cv2.imshow('just amplify', np.uint8(frame_just_amplified))

frame_previous = frame_now_float

# just_magnify_line = np.uint8(frame_made*100)[: ,x]
# just_magnify_line = np.uint8(frame_just_amplified)[: ,time_signal_x]
# guard_image(just_magnify_line)
# just_magnify_image[:, i] = just_magnify_line

# time_signal.append(frame_just_amplified[time_signal_y, time_signal_x])

# # derivative_outfile.write(np.uint8(frame_differentially_amplified))
# derivative_outfile.write(np.uint8(amplified_rendered))
# amplify_outfile.write(np.uint8(frame_just_amplified))
# original_outfile.write(frame_now)

# output
# output_color_int = np.uint8(frame_differentially_amplified)
output_color_int = np.uint8(amplified_rendered)

# extract time signal informaion from rendered video

# get a column of pixels from the image.

```



```

line_signal_out_x = output_color_int[:, time_signal_x]

# get a raw of pixels from the image.
line_signal_out_y = output_color_int[time_signal_y, :]

# line_signal_correlation_all = np.uint8(p*100)[:x]
# image_signal_correlation_all[:, i] = line_signal_correlation_all

# store the column into array of time signals
image_signal_out_x[:, i] = line_signal_out_x

# store the raw into array of time signals
image_signal_out_y[i, :] = line_signal_out_y


# extract time signal informaion from original video

# get a column of pixels from the image.
line_signal_original_x = frame_now[:, time_signal_x]

# get a raw of pixels from the image.
line_signal_original_y = frame_now[time_signal_y, :]

# store the column into array of time signals
image_signal_original_x[:, i] = line_signal_original_x

# store the raw into array of time signals
image_signal_original_y[i, :] = line_signal_original_y


buffer_count = buffer_count_next

if print_mode == 'time':
    t1 = time()
    t = t1 - t0
    t0 = t1
    print(t)

elif print_mode == 'number':
    sys.stdout.write("\033[F") # back to previous line
    sys.stdout.write("\033[K") # clear line
    print('frame:', i)

pressed_key = cv2.waitKey(1)
if pressed_key == ord('q'):
    break
elif pressed_key == ord('p'):

```

```

cv2.waitKey(0)
# sleep(frame_sleep_duration)

videoFile.release()

with open('time_signal.pickle', 'wb') as tsfile:
    pickle.dump(time_signal, tsfile)

cv2.imwrite(f'results/out_just_magnify_{just_file_name}_x_{time_signal_x}.jpg',
            just_magnify_image[:, :i])

original_outfile.release()
derivative_outfile.release()
amplify_outfile.release()

# image signal might be bigger than number of rendered frames and some of it might be
# empty.
# here we crop that image!
# cut non-rendered parts of image signal out
image_signal_out_x = image_signal_out_x[:, :i]

# cut non-rendered parts of image signal out
image_signal_out_y = image_signal_out_y[:i, :]

# cut non-needed parts of image signal
image_signal_original_x = image_signal_original_x[:, :i]

# cut non-needed parts of image signal
image_signal_original_y = image_signal_original_y[:i, :]

# write column of time signals from rendered video as an image
cv2.imwrite(f'results/out_differential_derivative_{just_file_name}_x_{time_signal_x}.jpg',
            image_signal_out_x)

# write row of time signals from rendered video as an image
cv2.imwrite(f'results/out_differential_derivative_{just_file_name}_y_{time_signal_y}.jpg',
            image_signal_out_y)

# write column of time signals from original video as an image
cv2.imwrite(f'results/orig_differential_{just_file_name}_x_{time_signal_x}.jpg',
            image_signal_original_x)

# write row of time signals from original video as an image
cv2.imwrite(f'results/orig_differential_{just_file_name}_y_{time_signal_y}.jpg',
            image_signal_original_y)

```

```
cv2.destroyAllWindows()
```

```
return (  
f'results/orig_differential_{just_file_name}.avi',  
f'results/out_differential_amplify_{just_file_name}.avi'  
)
```

۶.۲ روش اویلری خطی

```
#!/usr/bin/python3

# python source code for eularian video motion magnification with gaussian pyramid as
# spatial filter.

# usage:
# $ python3 nameOfThisFile.py inputVideoFileName

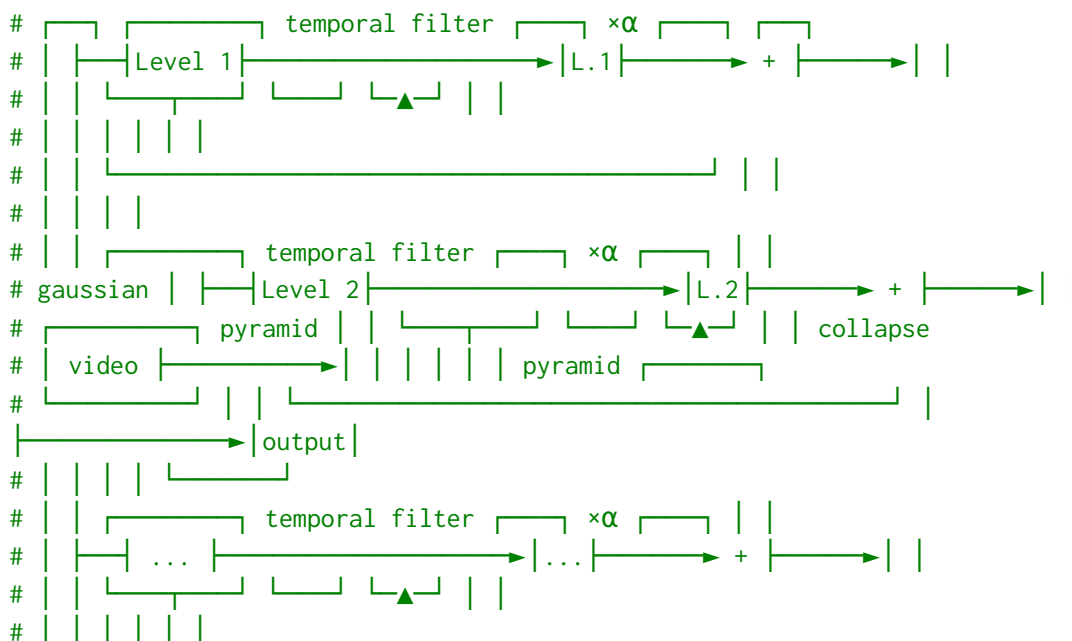
# written by Erfan Kheyrollahi in 2021

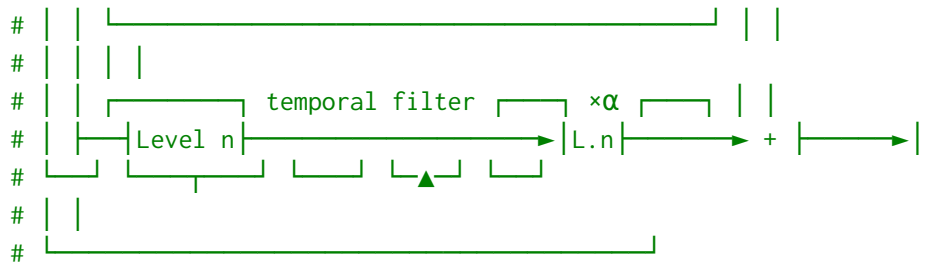
"""
"unlike people, comupters understand". Erfan Kheyrollahi - 2018
"""

# in this code we are going to implement the block diagram

# algorithm:
#
# step 1: take a video and convert each frame into a pyramid then store all of them in a
# list.
# step 2: take the pyramid and do a temporal filtering on each level of the pyramid.
# step 3: read original video frame by frame and convert them to pyramids.
# then add up corresponding filtered pyramid to it and finally collapse pyramid into a
# rendered frame.
# store the resulting frames in a video file.
```

Block Diagram





drawn using asciiflow.com

import libraries

for reading and writing videos and gaussian blur

import cv2

for working with matrices

import numpy as np

for cli

from os import path

for butterworth and other filters

from scipy import signal

for cli

import sys

from pyramids import gaussian_pyramid_make, gaussian_pyramid_rendr

from pyramids import laplacian_pyramid_make, laplacian_pyramid_rendr

from pyramids import pyramid_make, pyramid_rendr

do not let pixel values of image get out of their limit

def guard_image(image:np.ndarray):

any pixel value bigger than 255 will be cut down to 255

image[image > 255] = 255

any pixel value smaller than 0 will be 0

image[image < 0] = 0

magnify video. eulerian method

```
def eulerian(input_file_name:str, alpha:float, temporal_frequency_bands,
spatial_frequency_coefficients, number_of_frames:int, number_of_jump_frames:int=0,
filter_band_type:str='bandpass', time_signal_x:int=0, time_signal_y:int=0,
show_video:bool=True, pyramid_type='gaussian'
):
    """
```

magnify video with eulerian algorithm

- ``input_file_name``: name of the file
- ``alpha``: magnification ratio
- ``temporal_frequency_bands``: number or list or tuple of two numbers. high and low cutoff frequencies. for lowpass and highpass filters, only one number is enough.
- ``spatial_frequency_coefficients``: coefficients of summision for pyramid.
- ``number_of_frames``: number of video frames to read and magnify.
- ``number_of_jump_frames``: number of initial frames to skip. this is useful when your time of interest is not the start of video. also useful when you want to cut video into pieces and magnify each seperately.
- ``filter_band_type``: can be either ``lowpass``, ``highpass``, ``bandpass``, ``bandstop``. band type.
- ``time_signal_x`` and ``time_signal_y``: the pixel on image to extract time signal from. three time signals will be extracted. one from y'th row. one from x's column. other from pixel (x,y)
- ``show_video`` : if set to True, a video will be shown while rendering. results will always be stored in `./results` folder.

```
"""
```

```
# parameters
```

```
# parameters are these:
```

```
# 1. temporal frequency band
```

```
# 2. filter order
```

```
# 3. spatial bands coefficients
```

```
# 4. alpha (amplitution factor)
```

```
# amount of amplification. same as the  $\alpha$  factor in block diagram
```

```
# 5. frame range of the video to process
```

```
# lets start the job by opening the video file!
```

```
# open the video file to start the job
```

```
videoFile = cv2.VideoCapture(input_file_name)
```

```
# get frame rate of the video (frames per second. it is 30 for most standard videos)
```

```
video_frame_rate = videoFile.get(cv2.CAP_PROP_FPS)
```

```
# this program will read frames i to j of the video and magnify them
```

```
# the frame program should start reading from
```

```
# number of consecutive frames to read from video and then process
```

```
# these frames will be stored in a tank.
```

```
tank_size = number_of_frames
```

```
# this program uses pyramid for filtering into different spatial frequencies
```

```
# each level of pyramid is half of the previous one by both length and width
```

```
# this parameter sets depth of the pyramid. same as the n in number of pyramid levels in
block diagram
```

```
# spatial filters summition coefficients
```

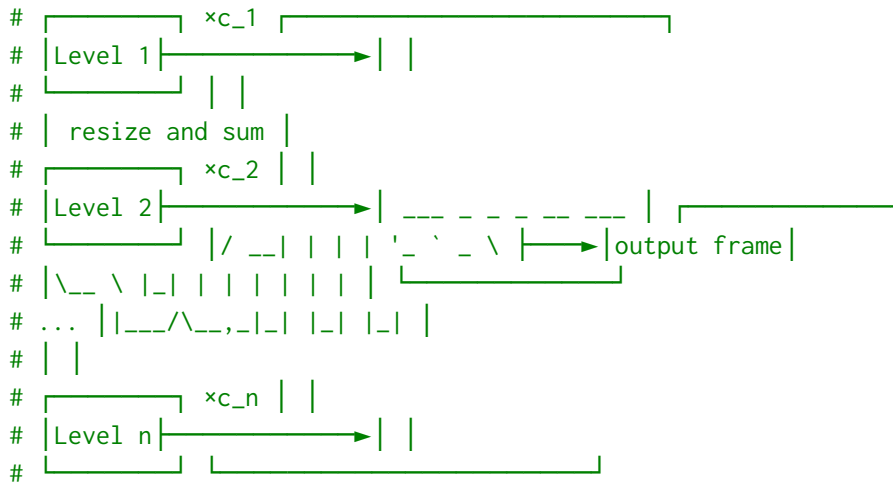
```
# these numbers will be multiplied into pyramid levels whn rendering before summition
```

```
# diagram below depicts how it's done
```

```
# pyramid coefficients resize levels and rendered
```

```
# levels add them together frame
```

```
#
```



```
# pyramid levels coefficients
```

```
# spatial frequency bands coefficients for rendering
```

```
# NOTE: if not all coefficients are set here,
```

```
# the default value will be used for all of them which is 1.00
```

```
spatial_coeff = spatial_frequency_coefficients
```

```
# number of levels in pyramid is equal to size of spatial coefficients provided.
```

```
# this is for making it more flexible.
```

```
number_of_pyramid_levels = len(spatial_coeff)
```

```
# sampling frequency of time-domain signal.
```

```
# this is the same as frame rate of the video.
```

```
# this parameter is necessary to set since filter cutoff frequencies are dependant on this.
```

```
fs = video_frame_rate
```

```
# filter design
```

```
# get cutoff frequencies from function input
```

```
# filter cutoff frequency (Hz)
```

```
# if a list of tuple is provided
```

```
if isinstance(temporal_frequency_bands, (list, tuple)):
```

```
# if there is two numbers in the list or tuple
```

```

if len(temporal_frequency_bands) > 1:

    # filter cutoff frequency (Hz)
    # lower cutoff frequency of bandpass or bandstop filter. also cutoff frequency of lowpass
    filter
    cutoff_low = temporal_frequency_bands[0]

    # higher cutoff frequency of bandpass or bandstop filter. also cutoff frequency of highpass
    filter
    cutoff_high = temporal_frequency_bands[1]

    # if there is only one number in list or tuple
    else:

        # filter cutoff frequency (Hz)
        # set low cutoff frequency to the provided number
        cutoff_low = temporal_frequency_bands[0]

        # set high cutoff frequency to the provided number
        cutoff_high = temporal_frequency_bands[0]

    # if a single number is provided
    else:

        # filter cutoff frequency (Hz)
        # set low cutoff frequency to the provided number
        cutoff_low = temporal_frequency_bands

        # set high cutoff frequency to the provided number
        cutoff_high = temporal_frequency_bands

    # filter type ('lowpass' or 'highpass' or 'bandpass' or 'bandstop')
    band_type = filter_band_type

    # nyquist frequency. for filter design
    nyq = 0.5 * fs

    # filter order. the order parameter for butterworth or other filters when designing them.
    order = 5
    # NOTE : you may need to change this.

    # normalized cutoff frequency

    # if filter should be bandpass or bandstop
    if band_type in ['bandpass', 'bandstop']:

        # get a list of normalized cutoff frequencies by deviding provided values by nyquist
        frequency
        normal_cutoff = [cutoff_low / nyq, cutoff_high / nyq]

```



```

# if filter should be highpass
elif band_type in ['high', 'highpass']:

# get normalized frequency by deviding provided values by nyquist frequency
normal_cutoff = cutoff_high / nyq

# if filter should be lowpass
elif band_type in ['low', 'lowpass']:

# get normalized frequency by deviding provided values by nyquist frequency
normal_cutoff = cutoff_low / nyq

# if filter band type is not specified (correctly) then print error message and exit
else:

# there is an error!
print("filter type is not correctly set", file=sys.stderr)

# exit the program to prevent it from bugs.
exit(2)

# design filter and get filter coefficients.
# this is a butterworth filter
filter_coef_b, filter_coef_a = signal.butter(order, normal_cutoff, btype=band_type,
analog=False)

# this is the iir filter of scipy.signal lib
# filter_coef_b, filter_coef_a = signal.iirfilter(order, normal_cutoff, btype=band_type)

# read first frame of the input video
_, frame_0 = videoFile.read()

# convert colorspace of first frame from BGR into gray
# we do this so that there is only one channel in the frame which is luminance
frame0_gray = cv2.cvtColor(frame_0, cv2.COLOR_BGR2GRAY)

# get frame height and width of the video. also number of color channels which is usually 3
frame_h, frame_w, number_of_channels = np.shape(frame_0)

# print needed detail about video
print(f'frame size is ({frame_w}, {frame_h}). frame rate is {np.round(video_frame_rate)}')

# define a list for storing pyramids

print('allocating memory')

# define pyramid for a video.
# dimentions of a video is number of frames × size of a frame

```

```

# a pyramid is like several videos. each with different frame sizes but same number of
frames.
# so it is defined like this:
#
# [ video of level 1 : number of frames × frame height × frame width ]
# [ video of level 2 : number of frames × frame height × frame width ]
# [ ... ]
# [ video of level n : number of frames × frame height × frame width ]
#
# note that frame size is different for each level

# generate video pyramid list
pyramid = [np.zeros((tank_size, frame_h//2**(i), frame_w//2**(i)), np.float32)
for i in range(number_of_pyramid_levels)]

# filtered video pyramid will be stored here. it should be the same size of the original
video pyramid
result = [np.zeros((tank_size, frame_h//2**(i), frame_w//2**(i)), np.float32)
for i in range(number_of_pyramid_levels)]

# start reading video frames and storing them into pyramid

print('storing video in pyramids')

# if we need to skip some frames, we can just read a bunch of frames and simply toss them
away
for i in range(number_of_jump_frames):

# read a frame and toss it out!
# these frames are going to be skipped. so let it go!
_, frame_now = videoFile.read()

# start reading frames one by one and storing them into pyramid.
for i in range(tank_size):

# read one frame from the video file
_, frame_now = videoFile.read()

# if there is no more frame in the video left
if frame_now is None:

# change tank size to number of frames read.
tank_size = i

# break the loop. stop magnifying frames.
break

# frame is basically a matrix of pixels. each pixel has 3 variables for each corresponding
channel.

```

```

# the variables are 8-bit unsigned integer numbers between 0 and 255 (uint8)
# for further processings we need floating point numbers.
# so here we convert the variable types from what it is into float32
frame_float = np.float32(frame_now)

# convert frame from BGR color space into single-channeled gray colorspace
frame_float_gray = cv2.cvtColor(frame_float, cv2.COLOR_BGR2GRAY)

# decrement result from the first frame. we do this to remove DC in fourier spectrum
# when using highpass filters, this is not really needed but in lowpass filtering it is
# necessary to remove DC. this method is not the best one since we may use FIR filters with
# windows smaller than tank_size. that way each window will have to be decremented from a
# different frame in order to remove DC
frame_float_gray_diff = frame_float_gray- np.float32(frame0_gray)

# generate a gaussian pyramid from the frame
pyramid_frame = pyramid_make(frame_float_gray_diff, number_of_pyramid_levels, pyramid_type)

# store resulting pyramid in video pyramid array
# each channel of the pyramid needs to be stored in its corresponding vector
# so that we can apply temporal filter to them seperately
# so here we do this
for level in range(number_of_pyramid_levels):

# store frame pyramid into video pyramid
pyramid[level][i] = pyramid_frame[level]

# we have read all the frames we wanted.
# release the video file
videoFile.release()

# prepare for temporal filtering

# definition: video: a bunch of frames
# definition: time signal matrix: a bunch of time signals
# a video has the same amount of time signal matrix.
# to get time signal matrix out of a video, it should be transposed

# for applying a filter all over a signal, we are going to use filtfilt function
# from scipy.signal . this function works as it should on a vector and filters it.
# but given a matrix with more than one dimention, it selects latest raws of the matrix
# which are vectors and filters them seperately. given a video matrix, it will first select
# one frame, then filter each raw of the frame seperately. this will do some kind of
spatial filtering
# instead of temporal filtering.
# so to achieve our goal, we first transpose our matrix so that dimentions
# change from (number of frames, height, width) into (width, height, number of frames)
# and latest dimention of this matrix is a time sequence for each pixel on image.
# and if we filter it, it will filter temporal signal of each pixel seperately and

```

```

# this is exactly what we want

print('transposing the tensor')

# transpose each level of video pyramid
for level in range(number_of_pyramid_levels):

    # note that transposed_video[x,y] is a time signal.
    # so we can get time signals this way.
    # and then process them by filtering or whatever else that we want
    # transpose video matrix
    pyramid[level] = np.transpose(pyramid[level])

# start filtering

print('time-domain filtering')
print()

# filter each level of the pyramid
for level in range(number_of_pyramid_levels):

    # write progress
    sys.stdout.write("\033[F") # back to previous line
    sys.stdout.write("\033[K") # clear line
    print(f'pyramid level {level}')

# filter each level of pyramid (which is a video matrix) temporally
result[level] = signal.filtfilt(filter_coef_b, filter_coef_a, pyramid[level])

print('transposing it again')

# remove original pyramid for memory optimization
del pyramid

# transpose the result once again to get the original video matrix
# here again, we need to transpose each video matrix in the video pyramid
for level in range(number_of_pyramid_levels):

    # transpose video matrix
    # this way we transpose time signals once again into a video.
    result[level] = np.transpose(result[level])

# if you want to keep the pyramid, you might also want to transpose it again.
# pyramid = np.transpose(pyramid)

# start rendering the video

print('showing the result: (press q to stop)\n')

```

```

# filename to write the result video in
just_file_name = path.basename(input_file_name)
# open an empty video file
outFile = cv2.VideoWriter(f'results/out_eulerian_{just_file_name}.avi',
cv2.VideoWriter_fourcc(*'MJPG'), video_frame_rate, (frame_w, frame_h))

outFile_orig = cv2.VideoWriter(f'results/orig_eulerian_{just_file_name}.avi',
cv2.VideoWriter_fourcc(*'MJPG'), video_frame_rate, (frame_w, frame_h))

# start reading the video file once again.
# because we need original frames to render the result and for memory optimization we have
deallocated them
videoFile = cv2.VideoCapture(input_file_name)

# let the first frame out. because when reading at first we did this
videoFile.read()

# extracted temporal signals from video will be stored in these arrays.

# define a temporal signal of zeros
image_signal_original_x = np.zeros((frame_h, number_of_frames, 3), np.uint8)

# define a temporal signal of zeros
image_signal_out_x = np.zeros((frame_h, number_of_frames, 3), np.uint8)

# define a temporal signal of zeros
image_signal_original_y = np.zeros((number_of_frames, frame_w, 3), np.uint8)

# define a temporal signal of zeros
image_signal_out_y = np.zeros((number_of_frames, frame_w, 3), np.uint8)

# start rendering output video
# first we need to read the frames one by one, add them up to its corresponding filtered
and
# amplified frame and show or store the result.
for i in range(len(result[0])):

# read a frame
_, frame_now = videoFile.read()

# if there is no frame to read
if frame_now is None:

# stop reading frames.
# and get out of the loop.
break

```

```

# convert color space from BGR into gray
frame_gray = cv2.cvtColor(frame_now, cv2.COLOR_BGR2GRAY)
# convert data type from uint8 into float32
frame_float_gray = np.float32(frame_gray)

# get corresponding part from the video pyramid
pyramid_frame_result = [result[level][i] for level in range(number_of_pyramid_levels)]

# collapse the pyramid and render it into one single frame
frame_result_rendered = pyramid_rendr(pyramid_frame_result, spatial_coeff=spatial_coeff,
pyramid_type=pyramid_type)

# amplify the filtered frame and sum it up with the original one
frame_mag = frame_float_gray * 1 + frame_result_rendered * alpha

# check for boundaries. since it is an 8bit video, all the pixels need to be between 0 and
255
# if there is overamplification and distortion because of clipping, we will see it as
artifacts.
guard_image(frame_mag)

# render luminance layer into BGR image.
# this is done by first splitting frame channels
# then adding luminance to each of them
# finally merging split channels again into a colorful frame.
output_color = cv2.merge(cv2.split(frame_now) +
frame_result_rendered * alpha)

# check for boundaries. since it is an 8bit video, all the pixels need to be between 0 and
255
# if there is overamplification and distortion because of clipping, we will see it as
artifacts.
guard_image(output_color)

# if we need to show the result while rendering
if show_video == True:

# show original frame
cv2.imshow('original', np.uint8(frame_now))

# show the magnified frame
cv2.imshow('result', np.uint8(frame_mag))
# show filtered frame
# cv2.imshow('filtered', np.uint8(frame_result_rendered))

# show one level of the pyramid
# cv2.imshow('pyr', np.uint8(result[1][i]))

```

```

# convert type of rendered image from float into 8 bit integer.
output_color_int = np.uint8(output_color)

# write original image into the output video file that is going to store original video.
outFile_orig.write(frame_now)

# extract time signal informaion from rendered video

# get a column of pixels from the image.
line_signal_out_x = output_color_int[:, time_signal_x]

# get a raw of pixels from the image.
line_signal_out_y = output_color_int[time_signal_y, :]

# store the column into array of time signals
image_signal_out_x[:, i] = line_signal_out_x

# store the raw into array of time signals
image_signal_out_y[i, :] = line_signal_out_y

# extract time signal informaion from original video

# get a column of pixels from the image.
line_signal_original_x = frame_now[:, time_signal_x]

# get a raw of pixels from the image.
line_signal_original_y = frame_now[time_signal_y, :]

# store the column into array of time signals
image_signal_original_x[:, i] = line_signal_original_x

# store the raw into array of time signals
image_signal_original_y[i, :] = line_signal_original_y

# write rendered frame into output file
outFile.write(np.uint8(output_color))
sys.stdout.write("\033[F") # back to previous line
sys.stdout.write("\033[K") # clear line
print('frame:', i)

# if there is any keypress, catch it.
pressed_key = cv2.waitKey(1)

# if pressed key was P
if pressed_key == ord('p'):

```

```
# pause the process and wait for next keypress
cv2.waitKey(0)
# if pressed key was Q
if pressed_key == ord('q'):

# stop rendering and get out of the rendering loop
break

# we are done reading all the frames we needed from original video file.
# so we can release it
videoFile.release()

# we are done writing original frames into output video file.
# so we can release it
outFile.release()

# we are done writing rendered frames into output video file.
# so we can release it
outFile_orig.release()

# write column of time signals from rendered video as an image
cv2.imwrite(f'results/out_eulerian_{just_file_name}_x_{time_signal_x}.jpg',
image_signal_out_x)

# write row of time signals from rendered video as an image
cv2.imwrite(f'results/out_eulerian_{just_file_name}_y_{time_signal_y}.jpg',
image_signal_out_y)

# write column of time signals from original video as an image
cv2.imwrite(f'results/orig_eulerian_{just_file_name}_x_{time_signal_x}.jpg',
image_signal_original_x)

# write row of time signals from original video as an image
cv2.imwrite(f'results/orig_eulerian_{just_file_name}_y_{time_signal_y}.jpg',
image_signal_original_y)

cv2.destroyAllWindows()

# that's it!

# return output file name
return (f'results/orig_eulerian_{just_file_name}.avi',
f'results/out_eulerian_{just_file_name}.avi')
```


۶.۳ روش همبستگی

"""

اینجا سعی می‌کنیم با کمک تابع همبستگی، حرکتهای نوسانی را آشکار کنیم

"""

to magnify a video, we have operation X which is this:

block diagram of this algorithm is like this:

#

#

#

this is a FIFO algorithm

both inputs and outputs sum up original frame

are read/written with corresponding

frame-by-frame correlation frame

#





Σ 



#

#

#

#

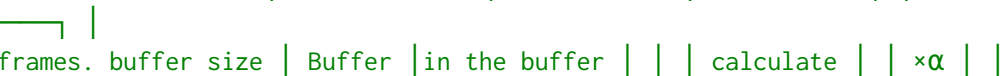
#

to avoid DC product of two functions |

from amplification can be a fair approach |

#

stores latest read 



affects correlation | 

function and the 

output | | |

| | | amplify/attenuate

#

first frame | |

in the buffer | |

#

#



generator |


```

import cv2

# for working with matrices
import numpy as np

# for cli
import os
# for cli
import sys

# for making delay while showing video
from time import sleep

# from pyramids import laplacian_pyramid_make, laplacian_pyramid_rendr
from pyramids import pyramid_make, pyramid_rendr


# do not let pixel values of image get out of their limit
def guard_image(image:np.ndarray):

# any pixel value bigger than 255 will be cut down to 255
image[image > 255] = 255

# any pixel value smaller than 0 will be 0
image[image < 0] = 0


def correlation(input_file_name, alpha, frequency, spatial_frequency_coefficients,
buffer_size, number_of_frames, number_of_jump_frames=0,
time_signal_x=0, time_signal_y=0, show_video=True, pyramid_type='gaussian'):

"""
# magnify video with correlation algorithm

- `input_file_name`: name of the file
- `alpha`: magnification ratio
- `frequency`: frequency of signal generator
- `spatial_frequency_coefficients`: coefficients of summision for pyramid.
- `buffer size`: size of buffer for calculating correlation.
- `number_of_frames`: number of video frames to read and magnify.
- `number_of_jump_frames`: number of initial frames to skip. this is useful when your time
of interest

```

is not the start of video. also useful when you want to cut video into pieces and magnify each seperately.

- `time_signal_x` and `time_signal_y`: the pixel on image to extract time signal from.

three time signals will be extracted. one from y'th row. one from x's column. other from pixel (x,y)

- `show_video` : if set to True, a video will be shown while rendering. results will always be

stored in ./results folder.

```
"""
```

```
# if you provide "camera" as input filename, camera of system will be used indtead of file.
```

```
# if you want to use camera
```

```
if input_file_name == 'camera':
```

```
# open camera as a video file
```

```
videoFile = cv2.VideoCapture(0)
```

```
# if a plain filename is provided
```

```
else:
```

```
# open videofile.
```

```
videoFile = cv2.VideoCapture(input_file_name)
```

```
# number of levels in pyramid is equal to size of spatial coefficients provided.
```

```
# this is for making it more flexible.
```

```
number_of_pyramid_levels = len(spatial_frequency_coefficients)
```

```
# pyramid levels coefficients
```

```
# spatial frequency bands coefficients for rendering
```

```
# NOTE: if not all coefficients are set here,
```

```
# the default value will be used for all of them which is 1.00
```

```
spatial_coeff = spatial_frequency_coefficients
```

```
spatial_coeff2 = [0, 0, 0, 1, 1]
```

```
spatial_coeff3 = [0, 0, 0, 1, 0]
```

```
spatial_coeff_correlation = spatial_frequency_coefficients
```

```
spatial_coeff_correlation_orig = [1] * len(spatial_frequency_coefficients)
```

```
# تاخیر مناسب بین فریمها تا ویدیو با سرعت عادی پخش شود
```

```
# به خاطر وجود تاخیر در حین پردازش، این تاخیر نصف شده است.
```

```
frame_sleep_duration = 1.0/60 / 2
```

```
# get frame rate of the video
```

```

# (frames per second. it is 30 for most standard videos)
video_frame_rate = videoFile.get(cv2.CAP_PROP_FPS)

# normalize alpha for size of the buffer
normalized_alpha = alpha / buffer_size

alpha2 = alpha * 0.01 * 0.5
normalized_alpha2 = alpha2 / buffer_size

# if we need to skip some frames, we can just read a bunch of frames and simply toss them
away
for i in range(number_of_jump_frames):

# read a frame and toss it out!
# these frames are going to be skipped. so let it go!
_, frame_now = videoFile.read()


# خواندن فریم اول و تعریف متغیرهای اولیه
_, frame_0 = videoFile.read()

# frame is basically a matrix of pixels. each pixel has 3 variables for each corresponding
channel.
# the variables are 8-bit unsigned integer numbers between 0 and 255 (uint8)
# for further processings we need floating point numbers.
# so here we convert the variable types from what it is into float32
frame_0_float = np.float32(frame_0)

# get frame size.
frame_height, frame_width, number_of_channels = np.shape(frame_0_float)

# define frame buffer. actually multiplication values will be stored here.
frame_buffer = np.zeros((buffer_size, frame_height, frame_width, number_of_channels),
dtype=np.float32)
# correlation_pyramid_buffer = [frame_buffer.copy() for _ in
range(number_of_pyramid_levels)]

# define a pyramid of black frames.
# this is for defining pyramid buffer.
# default value is zero.
pyramid_zero = np.array([

np.zeros(
(frame_height//2**(j), frame_width//2**(j), number_of_channels),
dtype=np.float32)

for j in range(number_of_pyramid_levels)
], dtype=np.ndarray)

```

```

# define a buffer of pyramids for storing correlation pyramids.
correlation_pyramid_buffer = np.array([
pyramid_zero.copy()
for i in range(buffer_size)
])

# generate pyramid of frame 0

pyramid_frame = pyramid_make(frame_0_float, number_of_pyramid_levels, pyramid_type)

# initial value of buffer is going to be frame 0

# for each frame in buffer
for i in range(buffer_size):

# store frame 0 in buffer
frame_buffer[i] = frame_0_float

#
correlation_pyramid = pyramid_zero.copy()

def freq_function(f:float, t:float) -> float:
return np.sin(t * f * 2 * np.pi / video_frame_rate)

frame_0_float_yuv = cv2.cvtColor(frame_0_float, cv2.COLOR_BGR2YUV)
frame_0_float_yuv_chans = cv2.split(frame_0_float_yuv)

print('buffer size -> ', correlation_pyramid_buffer.shape, f'× {frame_0.shape} <- frame
size ', 'fps=', np.round(video_frame_rate))

just_file_name = os.path.basename(input_file_name)

outFile = cv2.VideoWriter(f'results/out_correlation_{just_file_name}.avi',
cv2.VideoWriter_fourcc(*'MJPG'), video_frame_rate, (frame_width, frame_height))
OrigoutFile = cv2.VideoWriter(f'results/orig_correlation_{just_file_name}.avi',
cv2.VideoWriter_fourcc(*'MJPG'), video_frame_rate, (frame_width, frame_height))

tank_size = number_of_frames
image_signal_out = np.zeros((frame_height, tank_size, 3), np.uint8)
image_signal_original = np.zeros((frame_height, tank_size, 3), np.uint8)

correlation_all = np.zeros(frame_0_float.shape, np.float32)
image_signal_correlation_all = np.zeros((frame_height, tank_size, 3), np.uint8)

# extracted temporal signals from video will be stored in these arrays.

# define a temporal signal of zeros

```

```

image_signal_out_x = np.zeros((frame_height, number_of_frames, 3), np.uint8)

# define a temporal signal of zeros
image_signal_out_y = np.zeros((number_of_frames, frame_width, 3), np.uint8)

# define a temporal signal of zeros
image_signal_original_x = np.zeros((frame_height, number_of_frames, 3), np.uint8)

# define a temporal signal of zeros
image_signal_original_y = np.zeros((number_of_frames, frame_width, 3), np.uint8)


# this counter is for buffer to work
# it is index of buffer where the latest frame is stored
buffer_count = 0

# for better ui
print()


# start rendering output video
# frames will be read
# pyramids will be made from frame
# multiplication with signal will be calculated and stored in buffer
# correlation will be calculated
# magnification will be done
# pyramid will be rendered into frame
for i in range(number_of_frames):

# next frame on buffer. it is also oldest frame stored on buffer that is not popped out yet
buffer_count_next = (buffer_count + 1) % buffer_size

# read a frame
_, frame_now = videoFile.read()

# if there is no frame in video left
if frame_now is None:

# break the loop and stop rendering
break

# convert frame into float
frame_now_float = np.float32(frame_now)

# generate a pyramid of the read frame
pyramid_frame = np.array(pyramid_make(frame_now_float, number_of_pyramid_levels,
pyramid_type), dtype=np.ndarray)

# store the frame in the buffer
frame_buffer[buffer_count] = frame_now_float

```

```

pyramid_of_minus_frame = np.array(pyramid_make(frame_buffer[buffer_count_next],
number_of_pyramid_levels, pyramid_type), dtype=np.ndarray)
correlation_pyramid_now = (pyramid_frame - pyramid_of_minus_frame) *
freq_function(frequency, i)
correlation_pyramid_buffer[buffer_count] = correlation_pyramid_now
correlation_pyramid += correlation_pyramid_now
correlation_pyramid -= correlation_pyramid_buffer[buffer_count_next]

# correlation_all += (frame_now_float - frame_0_float) * freq_function(frequency, i)

# spatial_coeff_correlation_orig = [1] * 5
pyramid_magnified = [
correlation_pyramid[level] * spatial_coeff_correlation[level] *
freq_function(frequency*0.5, i) * normalized_alpha + pyramid_frame[level] *
spatial_coeff_correlation_orig[level]
for level in range(number_of_pyramid_levels)
]

# for level in pyramid_magnified:
# level[level < 0] = 0
# level[level > 255] = 255

# rendered_out = laplacian_pyramid_rendr(pyramid_magnified, spatial_coeff=[0,0,0,1,1])
# rendered_out = rendered_out * 10 + 100
rendered_out = pyramid_rendr(pyramid_magnified, pyramid_type=pyramid_type)

guard_image(rendered_out)
# pyramid_magnified = correlation_pyramid * normalized_alpha + pyramid_frame
# spatial_coeff2 = spatial_coeff
# rendered_out2 = pyramid_rendr(pyramid_magnified, spatial_coeff=spatial_coeff2,
pyramid_type=pyramid_type)

# guard_image(rendered_out2)

# magnified_plus_original = pyramid_rendr(pyramid_magnified, spatial_coeff=spatial_coeff[:-
1]+ [0], pyramid_type=pyramid_type) + frame_now_float
# magnified_plus_original = pyramid_rendr(pyramid_magnified, spatial_coeff=spatial_coeff,
pyramid_type=pyramid_type) + frame_now_float

# guard_image(magnified_plus_original)

# i1 = pyramid_rendr(correlation_pyramid, spatial_coeff=[0 , 0, 0, 1, 0],
pyramid_type=pyramid_type)
# guard_image(i1)
# i2 = pyramid_rendr(correlation_pyramid, spatial_coeff=[1 , 1, 1, 0, 0],
pyramid_type=pyramid_type)
# i3 = i1 * i2 + i1
# i4 = frame_now_float + i3 * 2

```



```

# guard_image(i4)

# p = correlation_all / (i + 1)
# q = frame_now_float * p * 1 + frame_now_float
# guard_image(p)
# guard_image(q)

if show_video == True:

# cv2.imshow('original', frame_now)
cv2.imshow('rendered pyramid', np.uint8(rendered_out))
# cv2.imshow('rendered pyramid+0', np.uint8(magnified_plus_original))
# cv2.imshow('rendered pyramid2', np.uint8(rendered_out2))
# cv2.imshow('i4', np.uint8(i4))
# cv2.imshow('correlation all', np.uint8(p*100))
# cv2.imshow('result', np.uint8(q))

output_color_int = np.uint8(rendered_out)
# output_color_int = np.uint8(rendered_out2)

# extract time signal informaion from rendered video

# get a column of pixels from the image.
line_signal_out_x = output_color_int[:, time_signal_x]

# get a raw of pixels from the image.
line_signal_out_y = output_color_int[time_signal_y, :]

# line_signal_correlation_all = np.uint8(p*100)[:x]
# image_signal_correlation_all[:, i] = line_signal_correlation_all

# store the column into array of time signals
image_signal_out_x[:, i] = line_signal_out_x

# store the raw into array of time signals
image_signal_out_y[i, :] = line_signal_out_y

# extract time signal informaion from original video

# get a column of pixels from the image.
line_signal_original_x = frame_now[:, time_signal_x]

```

```

# get a raw of pixels from the image.
line_signal_original_y = frame_now[time_signal_y, :]

# store the column into array of time signals
image_signal_original_x[:, i] = line_signal_original_x

# store the raw into array of time signals
image_signal_original_y[i, :] = line_signal_original_y


sys.stdout.write("\033[F") # back to previous line
sys.stdout.write("\033[K") # clear line
print('frame:', i)


pressed_key = cv2.waitKey(1)
if pressed_key == ord('q'):
    break
elif pressed_key == ord('p'):
    cv2.waitKey(0)
    # sleep(frame_sleep_duration)

buffer_count = buffer_count_next


outFile.write(np.uint8(rendered_out))
OrigoutFile.write(frame_now)


# output_color_int = np.uint8(rendered_out2)
# line_signal_out = output_color_int[:, x]
# image_signal_out[:, i] = line_signal_out

# line_signal_original = frame_now[:, x]
# image_signal_original[:, i] = line_signal_original

# line_signal_correlation_all = np.uint8(p*100)[: ,x]
# image_signal_correlation_all[:, i] = line_signal_correlation_all

correlation_all /= i


outFile.release()

```

```
OrigoutFile.release()
```

```
# image signal might be bigger than number of rendered frames and some of it might be empty.
```

```
# here we crop that image!
```

```
# cut non-rendered parts of image signal out
```

```
image_signal_out_x = image_signal_out_x[:, :i]
```

```
# cut non-rendered parts of image signal out
```

```
image_signal_out_y = image_signal_out_y[:i, :]
```

```
# cut non-needed parts of image signal
```

```
image_signal_original_x = image_signal_original_x[:, :i]
```

```
# cut non-needed parts of image signal
```

```
image_signal_original_y = image_signal_original_y[:i, :]
```

```
# write column of time signals from rendered video as an image
```

```
cv2.imwrite(f'results/out_correlation_{just_file_name}_x_{time_signal_x}.jpg',  
image_signal_out_x)
```

```
# write raw of time signals from rendered video as an image
```

```
cv2.imwrite(f'results/out_correlation_{just_file_name}_y_{time_signal_y}.jpg',  
image_signal_out_y)
```

```
# write column of time signals from original video as an image
```

```
cv2.imwrite(f'results/orig_correlation_{just_file_name}_x_{time_signal_x}.jpg',  
image_signal_original_x)
```

```
# write raw of time signals from original video as an image
```

```
cv2.imwrite(f'results/orig_correlation_{just_file_name}_y_{time_signal_y}.jpg',  
image_signal_original_y)
```

```
cv2.destroyAllWindows()
```

```
return (f'results/orig_correlation_{just_file_name}.avi',  
f'results/out_correlation_{just_file_name}.avi')
```

۶.۴ نرم افزار تست

```
# this code is for reproducing results

import videomag
from videomag import eulerian, differential, correlation
import os
import sys
import time

if len(sys.argv) > 1:
    filename = sys.argv[1]
else:
    filename = '/home/erfan/Downloads/baby.mp4'

if not os.path.exists('results'):
    os.system('mkdir results')

def get_filter_properties(cutoff_low, cutoff_high, order=5, bandtype='lowpass'):
    return (cutoff_low, cutoff_high, order, bandtype)

def reproduce_results(filename, alpha, number_of_frames, filter_properties,
frequency, spatial_coefficients, number_of_jump_frames=0,
time_signal_x=0, time_signal_y=0, show_video=True, pyramid_type='gaussian', buffer_size=5,
methods:list=[
    'eulerian',
    'correlation',
    'differential'
]):

    if 'eulerian' in methods:

        print('-----eulerian-----')

        time_0 = time.time()

        eulerian(filename, alpha, (filter_properties[0], filter_properties[1]),
        spatial_coefficients, number_of_frames,
        number_of_jump_frames=number_of_jump_frames,
        filter_band_type=filter_properties[3],
        time_signal_x=time_signal_x, time_signal_y=time_signal_y, show_video=show_video,
        pyramid_type=pyramid_type)

        time_1 = time.time()

        print('execution time:', time_1 - time_0)
```

```

if 'correlation' in methods:

    print('-----correlation-----')

    time_0 = time.time()

    correlation(filename, alpha, frequency, spatial_coefficients,
buffer_size=buffer_size, number_of_frames=number_of_frames,
number_of_jump_frames=number_of_jump_frames,
time_signal_x=time_signal_x, time_signal_y=time_signal_y, show_video=show_video,
pyramid_type=pyramid_type)

    time_1 = time.time()

    print('execution time:', time_1 - time_0)

if 'differential' in methods:

    print('-----differential-----')

    time_0 = time.time()

    differential(filename, alpha, number_of_frames,
spatial_frequency_coefficients=spatial_coefficients, buffer_size=buffer_size,
number_of_jump_frames=number_of_jump_frames,
time_signal_x=time_signal_x, time_signal_y=time_signal_y, show_video=show_video,
pyramid_type=pyramid_type)

    time_1 = time.time()

    print('execution time:', time_1 - time_0)

filter_properties = get_filter_properties(3, 3, order=5, bandtype='low')
alpha = 20
number_of_frames = 500
number_of_jump_frames = 0
frequency = 2
spatial_coefficients = [0, 0, 0, 1, 1]
time_signal_x = 1
time_signal_y = 1
buffer_size = 5
time_signal_x = 501
time_signal_y = 236
pyramid_type='laplacian'
show_video = True
show_video = False
methods = [
'eulerian',
'correlation',

```

```
'differential'  
]
```

```
reproduce_results(filename, alpha, number_of_frames, filter_properties, frequency,  
spatial_coefficients, number_of_jump_frames, time_signal_x, time_signal_y, show_video,  
pyramid_type, buffer_size, methods)
```

۶.۵ محاسبه هرم‌های گوسی و لاپلاسی

```
import numpy as np

import cv2

# make a laplacian pyramid out of a single frame
def laplacian_pyramid_make(image:np.ndarray, n_levels:int) -> list:
    """
    # laplacian pyramid make
    generates a laplacian pyramid from an image
    inputs:
    * image: image you want to make pyramid of
    * n_levels: pyramid depth
    output:
    * a list containing levels of pyramid
    """

    # pyramid levels will be stored in this list
    pyramids_list = []

    # get second level of gaussian pyramid
    one_level_down_image = image.copy()

    # current level of pyramid.
    current_image = image.copy()

    # get number of rows and columns of image
    rows, cols = map(int, image.shape[:2])

    # rows and columns of current level of pyramid
    current_rows, current_cols = rows, cols

    # start generating levels of the pyramid
    for i in range(n_levels - 1):

        # number of columns will be half
        cols //= 2

        # number of rows will be half
        rows //= 2

        # get next level of gaussian pyramid
        one_level_down_image = cv2.pyrDown(one_level_down_image, dstsize=(cols, rows))

        # remake one level up
        one_level_up_image = cv2.pyrUp(one_level_down_image, dstsize=(current_cols, current_rows))

        # subtract two images to get laplacian pyramid level
```

```

image_to_add = current_image - one_level_up_image

# add level into pyramid
pyramids_list.append(image_to_add)

# rows and columns of current level of pyramid
current_rows, current_cols = rows, cols

# current level of pyramid
current_image = one_level_down_image

# after adding all laplacian levels into pyramid, add latest level which is gaussian
pyramids_list.append(one_level_down_image)

# return the generated pyramid
return pyramids_list

# collapse a laplacian pyramid and render it into a frame
def laplacian_pyramid_rendr(pyramid:list, spatial_coeff:list=[]) -> np.ndarray:
    """
    # laplacian pyramid render
    collapses a laplacian pyramid into a single image
    inputs:
    * pyramid: the pyramid list you want to collapse
    * `spatial_coeff` (optional): coefficients to use while summing up levels

    NOTE: if not all spatial coefficients are provided, the default value will be used instead
    which is 1

    output:
    * a single image of rendered pyramid
    """

    # put frame size of pyramid levels in a list
    sizes = [tuple(map(int, pyramid[i].shape))[:2] for i in range(len(pyramid))]
    # get number of pyramid levels. it is just the length of the pyramid list
    n_levels = len(pyramid)

    # check if spatial coefficients are provided correctly in the input.
    # if not, we will use the default value which is 1 for all levels
    if len(spatial_coeff) != n_levels:
        spatial_coeff = [1] * n_levels

    # start collapsing pyramid
    # we will do it by upscaling levels one by one and adding them together.
    # it is done like this:
    # result = U(...(U(U(L_n) + L_{n-1}))...+L_1) where U is upscaling function

```



```

# so first we need to store the deepest level in a matrix
# store the last level of the pyramid in a frame and multiply it by its coefficient
rendered_image = pyramid[-1].copy() * spatial_coeff[-1]

# for each level in the pyramid from last to first, do:
for i in range(n_levels - 2, -1, -1):

# upscale the current image
rendered_image = cv2.pyrUp(rendered_image, dstsize=sizes[i][::-1])

# add one higher level of the pyramid to the current image
# we first multiply it by it's coefficient and then add it to the image
rendered_image += pyramid[i] * spatial_coeff[i]

# return the result
return rendered_image

```

```

# make a gaussian pyramid out of a single frame
def gaussian_pyramid_make(image: np.ndarray, n_levels: int) -> list:
    """
    # gaussian pyramid make
    generates a gaussian pyramid from an image
    inputs:
    * image: image you want to make pyramid of
    * n_levels: pyramid depth
    output:
    * a list containing levels of pyramid
    """

# pyramid levels will be stored in this list
# first element of list is first level of pyramid which is the input image itself
pyramids_list = [image]

# get number of rows and columns of image
rows, cols = map(int, image.shape[:2])

# generate other levels of the pyramid

# this is first level of the pyramid
# to generate next level of the pyramid previous level is needed
# and the first level is the image itself

# start generating other levels of the pyramid
for i in range(n_levels - 1):

```

```

# generate next level of the pyramid
# to do this, previous level of the pyramid is needed
one_level_down_image = cv2.pyrDown(
pyramids_list[i], dstsize=(cols // 2 ** (i + 1), rows // 2 ** (i + 1)))
# append generated level into pyramid
pyramids_list.append(one_level_down_image)

# return generated pyramid
return pyramids_list

# collapse a gaussian pyramid and render it into a frame
def gaussian_pyramid_rendr(pyramid: list, spatial_coeff: list = []) -> np.ndarray:
"""
# gaussian pyramid render
collapses a gaussian pyramid into a single image
inputs:
* pyramid: the pyramid list you want to collapse
* `spatial_coeff` (optional): coefficients to use while summing up levels

NOTE: if not all spatial coefficients are provided, the default value will be used instead
which is 1

output:
* a single image of rendered pyramid
"""

# put frame size of pyramid levels in a list
# sizes = [tuple(map(int, pyramid[i].shape)) for i in range(len(pyramid))]
sizes = [tuple(map(int, pyramid[i].shape))[:2] for i in range(len(pyramid))]

# get number of pyramid levels. it is just the length of the pyramid list
n_levels = len(pyramid)

# check if spatial coefficients are provided correctly in the input.
# if not, we will use the default value which is 1 for all levels
if len(spatial_coeff) != n_levels:
spatial_coeff = [1] * n_levels

# start collapsing pyramid
# we will do it by upscaling levels one by one and adding them together.
# it is done like this:
# result = U(...(U(U(L_n) + L_{n-1}))...+L_1) where U is upscaling function
# so first we need to store the deepest level in a matrix
# store the last level of the pyramid in a frame and multiply it by its coefficient
rendered_image = pyramid[-1].copy() * spatial_coeff[-1]

# for each level in the pyramid from last to first, do:
for i in range(n_levels - 2, -1, -1):

# upscale the current image

```

```

rendered_image = cv2.pyrUp(rendered_image, dstsize=sizes[i][::-1])

# add one higher level of the pyramid to the current image
# we first multiply it by it's coefficient and then add it to the image
rendered_image += pyramid[i] * spatial_coeff[i]

# devide the result by sum of all coefficients to normalize it
rendered_image /= sum(spatial_coeff)

# return the result
return rendered_image

# make a gaussian pyramid out of a single frame
def pyramid_make(image: np.ndarray, n_levels: int, pyramid_type:str='gaussian') -> list:
    """
    # gaussian or laplacian pyramid make
    generates a gaussian or laplacian pyramid from an image
    inputs:
    * image: image you want to make pyramid of
    * n_levels: pyramid depth
    * pyramid_type: can be either "gaussian" or "laplacian" . type of pyramid
    output:
    * a list containing levels of pyramid
    """
    if pyramid_type == 'gaussian':
        return gaussian_pyramid_make(image, n_levels)
    elif pyramid_type == 'laplacian':
        return laplacian_pyramid_make(image, n_levels)

# collapse a gaussian pyramid and render it into a frame
def pyramid_rendr(pyramid: list, spatial_coeff: list = [], pyramid_type:str='gaussian') ->
np.ndarray:
    """
    # gaussian or laplacian pyramid render
    collapses a gaussian or laplacian pyramid into a single image
    inputs:
    * pyramid: the pyramid list you want to collapse
    * `spatial_coeff` (optional): coefficients to use while summing up levels
    * pyramid_type: can be either "gaussian" or "laplacian" . type of pyramid

    NOTE: if not all spatial coefficients are provided, the default value will be used instead
    which is 1

    output:
    * a single image of rendered pyramid
    """

```

```
if pyramid_type == 'gaussian':  
    return gaussian_pyramid_rendr(pyramid, spatial_coeff)  
elif pyramid_type == 'laplacian':  
    return laplacian_pyramid_rendr(pyramid, spatial_coeff)
```

1. Wu, Hao-Yu, et al. "Eulerian video magnification for revealing subtle changes in the world." *ACM transactions on graphics (TOG)* 31.4 (2012): 1-8.
2. Balakrishnan, G., Durand, F., Guttag, J.: Detecting pulse from head motions in video. In: IEEE Conf. on Comput. Vis. and Pattern Recognit. pp. 3430–3437 (2013)
3. Cheng, Xiaogang, et al. "A pilot study of online non-invasive measuring technology based on video magnification to determine skin temperature." *Building and Environment* 121 (2017): 1-10.
4. Cha, Y.J., Chen, J., Büyüköztürk, O.: Output-only computer vision based damage detection using phase-based optical flow and unscented kalman filters. *Engineering Structures* 132, 300–313 (2017)
5. Gonzalez, R. C., & Woods, R. E. (2018). *Digital Image Processing*. Pearson.
6. Huang, Yong, and Lucas Hui. "An adaptive spatial filter for additive Gaussian and impulse noise reduction in video signals." *Fourth International Conference on Information, Communications and Signal Processing, 2003 and the Fourth Pacific Rim Conference on Multimedia. Proceedings of the 2003 Joint*. Vol. 1. IEEE, 2003.
7. Rao, D. H., and Patavardhan Prashant Panduranga. "A survey on image enhancement techniques: classical spatial filter, neural network, cellular neural network, and fuzzy filter." *2006 IEEE International Conference on Industrial Technology*. IEEE, 2006.
8. Gonzalez, R. C., & Woods, R. E. (2018). *Digital Image Processing*. Pearson.
9. Zhao, Ming, et al. "Super-resolution of cardiac magnetic resonance images using Laplacian Pyramid based on Generative Adversarial Networks." *Computerized Medical Imaging and Graphics* 80 (2020): 101698.
10. Ataky, Steve Tsham Mpinda, et al. "Data augmentation for histopathological images based on Gaussian-Laplacian pyramid blending." *2020 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2020.
11. Addesso, Paolo, Rocco Restaino, and Gemine Vivone. "An Improved Version of the Generalized Laplacian Pyramid Algorithm for Pansharpening." *Remote Sensing* 13.17 (2021): 3386.
12. Paris, Sylvain, Samuel W. Hasinoff, and Jan Kautz. "Local Laplacian filters: Edge-aware image processing with a Laplacian pyramid." *ACM Trans. Graph.* 30.4 (2011): 68.
13. Shao, Ling, et al. "Spatio-temporal Laplacian pyramid coding for action recognition." *IEEE Transactions on Cybernetics* 44.6 (2013): 817-827.
14. Yan, Li, and Qiao Yanfeng. "An adaptive temporal filter based on motion compensation for video noise reduction." *2006 International Conference on Communication Technology*. IEEE, 2006.
15. Karlsson, Linda S., Mårten Sjöström, and Roger Olsson. "Spatio-temporal filter for ROI video coding." *2006 14th European Signal Processing Conference*. IEEE, 2006.
16. Chen, Cheng, Jingning Han, and Yaowu Xu. "A Non-local Mean Temporal Filter for Video Compression." *2020 IEEE International Conference on Image Processing (ICIP)*. IEEE, 2020.
17. Palmieri, Pierpaolo, et al. "Human arm motion tracking by kinect sensor using kalman filter for collaborative robotics." *The International Conference of IFToMM ITALY*. Springer, Cham, 2020.
18. Wu, Hao-Yu, et al. "Eulerian video magnification for revealing subtle changes in the world." *ACM transactions on graphics (TOG)* 31.4 (2012): 1-8.
19. Brostow, Gabriel J., and Irfan Essa. "Image-based motion blur for stop motion animation." *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. 2001
20. J. R. Bergen, P. J. Burt, R. Hingorani, and S. Peleg, Computing Two Motions from Three Frames. In Proceedings of International Conference on Computer Vision 1990, pages 27-32, 1990.
21. Liu, Ce, et al. "Motion magnification." *ACM transactions on graphics (TOG)* 24.3 (2005): 519-526.
22. Wu, Hao-Yu, et al. "Eulerian video magnification for revealing subtle changes in the world." *ACM transactions on graphics (TOG)* 31.4 (2012): 1-8.

୧୩. Wadhwa, Neal, et al. "Phase-based video motion processing." *ACM Transactions on Graphics (TOG)* 32.4 (2013): 1-10.

୧୪. Justin G. Chen, Neal Wadhwa, Young-Jin Cha, Frédo Durand, William T. Freeman, Oral Buyukozturk

Structural Modal Identification through High Speed Camera Video: Motion Magnification

Proceedings of the 32nd International Modal Analysis Conference (2014)

୧୫. Oh, Tae-Hyun, et al. "Learning-based video motion magnification." *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018.

Oh, Tae-Hyun, et al. "Learning-based video motion magnification." *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018.

୧୬. Gonzalez, Rafael C., Richard E. Woods, and Barry R. Masters. "Digital image processing." (2009): 029901-029901.

୧୭. Wu, Hao-Yu, et al. "Eulerian video magnification for revealing subtle changes in the world." *ACM transactions on graphics (TOG)* 31.4 (2012): 1-8.

୧୮. ACM Computing Surveys. 21 (3): 359–411. doi:10.1145/72551.72554. S2CID 207637854.

୧୯. Chan, Ian H. "Swept sine chirps for measuring impulse response." *Power (dBVrms)* 50.40 (2010): 30.

୨୦. Verma, Rohit, and Jahid Ali. "A comparative study of various types of image noise and efficient noise removal techniques." *International Journal of advanced research in computer science and software engineering* 3.10 (2013).

୨୧. Wu, Hao-Yu, et al. "Eulerian video magnification for revealing subtle changes in the world." *ACM transactions on graphics (TOG)* 31.4 (2012): 1-8.

୨୨. Shannon, C. E. (1942). The theory and design of linear differential equation machines". Fire Control of the US National Defense Research Committee: Report 411, Section D-2 (N. J. A. Sloane & A. D. Wyner, Eds.; Reprint). Collected Papers. Wiley IEEE Press.

୨୩. Wu, Hao-Yu, et al. "Eulerian video magnification for revealing subtle changes in the world." *ACM transactions on graphics (TOG)* 31.4 (2012): 1-8.

୨୪. Bradski, Gary, and Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. " O'Reilly Media, Inc.", 2008.

୨୫. Hore, Alain, and Djemel Ziou. "Image quality metrics: PSNR vs. SSIM." *2010 20th international conference on pattern recognition*. IEEE, 2010.