

Progettazione e sviluppo di una web application per la stabilizzazione video

Scrofano Giuseppe
1000001136

18 aprile 2024

0.1 Introduzione

Il progetto tratta la realizzazione di una web application che permetta all'utente di stabilizzare il movimento di camera in un video, offrendogli inoltre la possibilità di scegliere tra tre algoritmi di stabilizzazione differenti.

Questa relazione è suddivisa in due parti:

- Quadro teorico: verranno analizzate tutti gli algoritmi utilizzati da un punto di vista torico;
- Quadro tecnico: verrà analizzata dettagliatamente la realizzazione tecnica dell'applicazione;

0.2 sistemi di stabilizzazione

Un sistema di stabilizzazione dell'immagine ha come scopo quello di attenuare i movimenti della camera da una sequenza di immagini. Distinguiamo due tipi di movimenti principali:

- Padding: movimenti intenzionali;
- Jitter: movimenti non intenzionali;

I sistemi di stabilizzazione si dividono in:

- Sistemi di stabilizzazione analogici
- Sistemi di stabilizzazione digitali (**DIS**)

0.2.1 Sistemi di stabilizzazione analogici

I **Sistemi di stabilizzazione analogici** assicurano movimenti fluidi durante le riprese e assorbono le vibrazioni. Se un sistema di stabilizzazione è regolato correttamente, la videocamera sarà sempre perfettamente in equilibrio, indipendentemente dai movimenti, aumentando la qualità delle registrazioni video.

Alcuni esempi di questi sistemi sono: accelerometri, giroscopi, sensori di velocità angolare e ammortizzatori meccanici.

0.2.2 Sistemi di stabilizzazione digitali

In alcune videocamere viene utilizzata la stabilizzazione digitale dell'immagine in tempo reale, chiamata anche stabilizzazione elettronica dell'immagine **DIS**.

Questa tecnica sposta l'area ritagliata letta dal sensore di immagine per contrastare il movimento in ciascun fotogramma.

Questo sistema riduce le vibrazioni dei video rendendo più fluida la transizione da un fotogramma all'altro. Ai fini della realizzazione del progetto siamo interessati proprio a quest'ultima tipologia di sistemi e ne studieremo il funzionamento in dettaglio.

Parte I

Quadro teorico

Capitolo 1

DIS

I Dis si distinguono in **DIS real time** e **DIS post processing** ed il loro funzionamento è suddiviso in tre fasi principali:

1. Stima del movimento
2. Filtraggio (correzione) del movimento
3. Deformazione (post-processing) dell'immagine

1.1 Stima del movimento

Gli spostamenti sono identificati dalla seguente funzione matematica

$$d(x) = x' - x \text{ dove } x \text{ è la posizione iniziale}$$

$d(x)$ prende il nome di **motion vector**(MV).

L'insieme dei $d(x)$ per ogni x nell'immagine è detto **Campo di movimento (motion field)** I motion field possono essere parametrizzati utilizzando una delle seguenti rappresentazioni:

- Rappresentazione globale
- Rappresentazione basata su pixel
- Rappresentazione basata su regioni
- Rappresentazione basata su blocchi

1.1.1 Criterio DFD

Il Displaced Frame Difference è un metodo che permette di stimare i parametri del modello di movimento. Si basa sulla differenza fra frame discostati. Siano:

- "Anchor frame I_1 a t_1 ": il frame iniziale
- "Target frame I_2 a t_2 ": il frame su cui stimare il relativo movimento

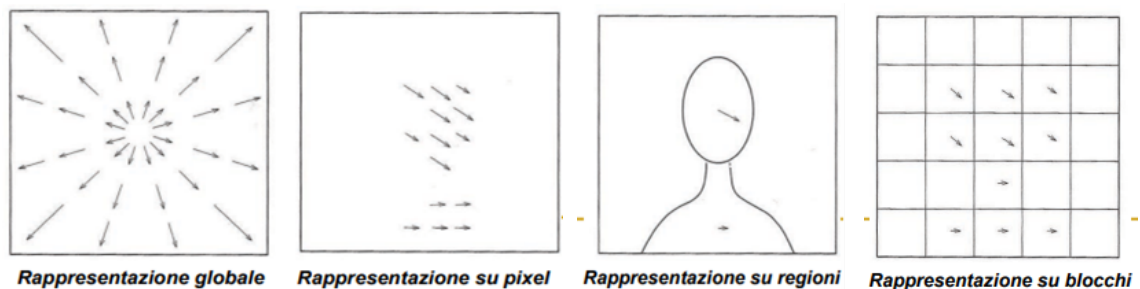


Figura 1.1: Rappresentazioni

Definiamo:

- $t_1 > t_2$: backward motion estimation
- $t_1 < t_2$: forward motion estimation

Siano inoltre:

$$w(x; a) = x + d(x; a) \text{ dove } a \text{ rappresenta i parametri di movimento da stimare}$$

Definiamo l'errore **DFD**:

$$E_{DFD}(a) = \sum_{x \in \lambda} |I_2(w(x; a)) - I_1(x)|^p$$

dove:

- λ : insieme dei pixel nell'anchor frame I_1
- p : intero positivo:
 - $p = 1 \rightarrow E_{DFD}$ è detto **mean absolute difference (MAD)**
 - $p = 2 \rightarrow E_{DFD}$ è detto **mmean squared error (MSE)**

Affinché la stima dei parametri di a sia la migliore, si dovrà minimizzare E_{DFD} ed essendo a un vettore, dovremo in generale imporre che il gradiente di E_{DFD} valga 0

1.1.2 Algoritmi di Block-Matching (BMA)

Consideriamo la rappresentazione del movimento basata su blocchi. Questi possono avere qualsiasi forma geometrica (solitamente quadrata), purché:

$$U_{m \in M} B_m = \lambda, \text{ e } B_m \cap B_n = \emptyset, m \neq n$$

Assumiamo che tutti i pixel di un blocco B_m si muovano nella stessa direzione un MV per blocco.

Dato l'anchor frame B_m vogliamo trovare il target frame B'_m che minimizza l'errore E_{DFD} , cioè voglio minimizzare

$$E_m(d_m) = \sum_{x \in B_m} |I_2(x + d_m) - I_1(x)|^p$$

Per trovare il migliore d_m confrontiamo (dopo averli calcolati) tutti i d_m fra l'anchor frame B_m e tutti i possibili target frame B'_m all'interno di una regione di ricerca

1.1.3 Algoritmi di Features Extraction

Algoritmo per estrarre punti chiave "**Features from Accelerated Segment Test (FAST)**".

Individua dei punti salienti (detti **corner**) nel frame da usare come features per "tracciare il movimento".

Si analizza ad ogni passo un insieme di 16 punti con configurazione a cerchio di raggio 3 per classificare se un punto p sia o meno di corner. In dettaglio, data l'intensità l_p del punto p , se per almeno 12 punti contigui con intensità l_x si ottiene che $|l_x - l_p| > t$, con t soglia, allora p sarà considerato corner.

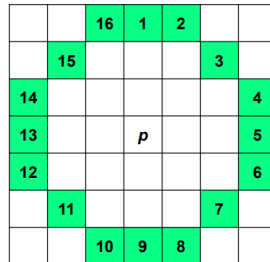


Figura 1.2: Cerchio di raggio 3

1.2 Filtraggio del movimento

L'obiettivo del filtraggio del movimento è distinguere i movimenti intenzionali (**padding**) da quelli casuali (**jitter**). L'algoritmo di **motion filtering** utilizzato in questo progetto è il **Filtro Kalman**.

1.2.1 Filtro Kalman

È uno strumento molto potente che prende in input una serie di misure osservate nel tempo e tiene conto di un eventuale rumore casuale. Il suo funzionamento è diviso in:

- Predizione: fissato un tempo t stima la misura z_{t+1}
- Correzione: osservata una misura può correggerla, eventualmente basandosi su una predizione

La versione discreta del filtro si basa su due equazioni differenziali stocastiche (lineari):

- Nel processo stocastico, lo stato x_k è dato da:

$$x_t = Ax_{t-1} + Bu_t + W_t$$

- La misura z_k di x_k è data da:

$$z_t = Hx_t + \nu_t$$

Dove t = istante temporale, A, B, H = *modelli transazionali*, w, ν = *rumore gaussiano* e u = *controllo dell'utente*

Affiancando il filtro di Kalman ad un tracciatore Bayesiano possiamo stimare le probabilità:

- Predizione: $P(x_t|z_{t-1})$
- Correzione(update): $P(x_t|z_t)$

1.3 Post-processing: Deformazione dell'immagine

Lo scopo principale è migliorare la qualità dei frame, per esempio, si può agire anche sui valori di intensità nei pixel, qualora l'esposizione del video non sia uniforme.



Figura 1.3: Esposizione

Capitolo 2

OpenCV

OpenCV (Open Source Computer Vision Library) è una libreria open source utilizzata per la Computer Vision in tempo reale che include diverse centinaia di algoritmi di visione artificiale.

OpenCV ha una struttura modulare, il che significa che il pacchetto include diverse librerie condivise o statiche. Sono disponibili i seguenti moduli:

- **Funzionalità core (core)**: un modulo compatto che definisce strutture dati di base, incluso il denso array multidimensionale Mat e le funzioni di base utilizzate da tutti gli altri moduli.
- **Elaborazione delle immagini (imgproc)**: un modulo di elaborazione delle immagini che include filtraggio delle immagini lineare e non lineare, trasformazioni geometriche delle immagini (ridimensionamento, deformazione affine e prospettiva, rimappatura generica basata su tabelle), conversione dello spazio colore, istogrammi e così via.
- **Analisi video (video)**: un modulo di analisi video che include algoritmi di **stima del movimento**, rimozione dello sfondo e tracciamento degli oggetti.
- **Calibrazione della fotocamera e ricostruzione 3D (calib3d)**: include calibrazione della fotocamera, stima della posa dell'oggetto, elementi di ricostruzione 3D.
- **Framework delle funzionalità 2D (features2d)**: include rilevatori di caratteristiche salienti, descrittori e abbinatori di descrittori.
- **Rilevamento oggetti (objdetect)**: si occupa del rilevamento di oggetti (ad esempio volti, occhi, tazze, persone, automobili e così via).
- **GUI di alto livello (highgui)**: un'interfaccia facile da usare.
- **Video I/O (videoio)**: un'interfaccia facile da usare per l'acquisizione video e i codec video.
- altri moduli di supporto, come FLANN.

2.1 Good Features to Track

Il rilevamento degli angoli (**Corner detection**) è un approccio utilizzato nei sistemi di visione artificiale per estrarre determinati tipi di caratteristiche e dedurre il contenuto di un'immagine.

Un angolo può essere interpretato come la giunzione di due bordi, dove un bordo è un improvviso cambiamento nella luminosità dell'immagine. Gli angoli sono le caratteristiche importanti dell'immagine e sono generalmente definiti come punti di interesse che sono invarianti alla traslazione, alla rotazione e all'illuminazione.

Per catturare gli angoli dall'immagine, i ricercatori hanno proposto molti rilevatori d'angolo diversi tra cui l'operatore Harris.

2.1.1 Algoritmo di Harris

Per quantificare l'angolarità di un punto viene definita una funzione di risposta

$$R = \lambda_1 \lambda_2 - K(\lambda_1 + \lambda_2)^2$$

L'algoritmo di Harris può essere riassunto come:

- conversione dell'immagine in scala di grigi;
- riduzione del rumore tramite un filtro Gaussiano;
- approssimazione del gradiente tramite l'operatore di Sobel;
- per ogni pixel dell'immagine, calcolo della funzione di risposta R in un intorno di dimensione 3×3
- determinazione dei punti di massimo locale di R , che rappresentano gli angoli rilevati nell'immagine.

OpenCV utilizza un'evoluzione dell'operatore di Harris sviluppato da Shi-Tomasi.

La funzione di risposta nello Shi-Tomasi Corner Detector è dato da:

$$R = \min(\lambda_1, \lambda_2)$$

Se è maggiore di una certa soglia, viene considerato un corner

2.1.2 Codice

OpenCV ha una funzione, **cv.goodFeaturesToTrack()** che trova gli N angoli(**corner**) in un'immagine con il metodo Shi-Tomasi (o Harris Corner Detection, se richiesto esplicitamente).

L'immagine deve essere in scala di grigi, successivamente è necessario specificare il numero di angoli che si vuole trovare e il livello di qualità, cioè un valore compreso tra 0 e 1 che denota la soglia minima di qualità di un corner al di sotto della quale qualunque altro angolo viene respinto.

L'ultimo parametro da specificare è la distanza euclidea minima tra i corner rilevati.

2.2 Flusso ottico

Il flusso ottico è lo schema del movimento apparente degli oggetti dell'immagine tra due fotogrammi consecutivi causato dal movimento dell'oggetto o della fotocamera.

È un campo vettoriale 2D in cui ciascun vettore è un vettore di spostamento che mostra il movimento dei punti da un fotogramma al successivo.

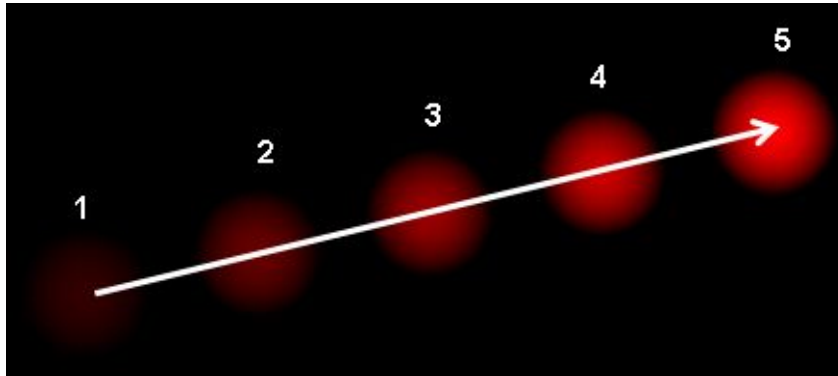


Figura 2.1: Palla che si muove in 5 fotogrammi consecutivi

Il flusso ottico funziona su diversi presupposti:

- Le intensità dei pixel di un oggetto non cambiano tra fotogrammi consecutivi.
- I pixel vicini hanno un movimento simile.

2.2.1 Metodo Lucas-Kanade

Consideriamo un pixel $I(x, y, t)$ nel primo fotogramma. Si muove per distanza (dx, dy) nel fotogramma successivo ripreso dopo dt tempo. Quindi, poiché quei pixel sono gli stessi e l'intensità non cambia, possiamo dire:

$$I(x, y, t) = I(x + dx, y + dy, t + dt)$$

Da qui si considera l'approssimazione in serie di Taylor del lato destro, si rimuovono i termini comuni e si dividono per dt per ottenere la seguente equazione:

$$f_z u + f_y v + f_t = 0$$

L'equazione sopra è chiamata equazione del flusso ottico. Per risolverla si utilizza il **metodo di Lucas-Kanade**. Lo scopo è trovare un modo per calcolare le variabili u e v dell'equazione sopra.

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \sum_i f_{x_i}^2 & \sum_i f_{x_i} f_{y_i} \\ \sum_i f_{x_i} f_{y_i} & \sum_i f_{y_i}^2 \end{bmatrix}^{-1} \begin{bmatrix} -\sum_i f_{x_i} f_{t_i} \\ -\sum_i f_{y_i} f_{t_i} \end{bmatrix}$$

Figura 2.2: Calcolo di u, v

2.2.2 Codice

In OpenCV si utilizza la funzione **cv.calcOpticalFlowPyrLK()**. Lista dei parametri:

- prevImg: prima immagine a 8 bit costruita da **buildOpticalFlowPyramid**;
- nextImg: seconda immagine della stessa dimensione e dello stesso tipo di prevImg;
- prevPts: vettore dei punti 2D per i quali occorre trovare il flusso;
- nextPts: vettore di output di punti 2D
- status: vettore di stato, ogni elemento del vettore è posto a 1 se è stato trovato il flusso per le caratteristiche corrispondenti, altrimenti è posto a 0.
- err: vettore degli errori in uscita;

Capitolo 3

Node.js

Node.js è un ambiente runtime JavaScript gratuito, open source e multi-piattaforma che consente agli sviluppatori di creare server, app Web, strumenti da riga di comando e script.

3.1 Express.js

Express è un framework per applicazioni web Node.js flessibile e leggero che fornisce una serie di funzioni avanzate per le applicazioni web e per dispositivi mobili.

3.1.1 Static

Per gestire i file statici, quali immagini, file CSS e file JavaScript, si utilizza la funzione middleware integrata `express.static`.

È necessario fornirgli come parametro il nome della directory che contiene gli asset statici per iniziare a gestire i file direttamente. Ad esempio:

```
app.use(express.static('public'))
```

3.1.2 Routing

Per Routing si intende determinare come un'applicazione risponde a una richiesta client a un endpoint particolare, il quale è un URI (o percorso) e un metodo di richiesta HTTP specifico (GET, POST e così via).

Ciascuna route può disporre di una o più funzioni dell'handler, le quali vengono eseguite quando si trova una corrispondenza per la route.

```
app.get('/', function(req, res){res.send('HelloWorld!')});
```

3.2 Socket.io

Socket.IO è una libreria che consente la comunicazione a bassa latenza , bidirezionale e basata su eventi tra un client e un server. La libreria è basata sui websocket.



Figura 3.1: Schema della comunicazione

3.2.1 Eventi

Socket.IO fornisce un modo conveniente per inviare un evento e ricevere una risposta:

```
socket.emit("hello", "world", (response) => console.log(response); // "gotit");
```

e dal mittente:

```
socket.on("hello", (arg, callback) => console.log(arg); // "world" callback("gotit"););
```

Parte II

Quadro Tecnico

Capitolo 4

Web Application

Tutti i file del progetto sono situati all'interno di un'unica directory "node script", con la seguente struttura:

- node_modules: i moduli di node installati tramite il gestore di pacchetti **npm**;
- public: la cartella contenente i file STATIC di Express.js introdotti nel capitolo precedente;
- video: cartella essenziale all'interno della quale vengono memorizzati e stabilizzati i video caricati dai client. La cartella contiene a sua volta
 - varie directory temporanee utilizzate per l'elaborazione dei video
 - lo script c++ "**Videostab.cpp**": contiene algoritmo di stabilizzazione originale implementato.
 - lo script c++ "**VideostabKalman.cpp**": modifica del primo algoritmo che utilizza il filtro Kalman
 - lo script python "**main.py**": implementazione python(tramite la libreria Vidstab) del primo algoritmo.
- : index.js;

4.1 Funzionamento dell'applicazione

Analizziamo i casi d'uso.

4.1.1 Client

Dal punto di vista dell'utente, la home permette di caricare un video e selezionare uno dei tre algoritmi implementati. Una volta finita l'elaborazione del video, verrà mostrato l'output(il video stabilizzato) direttamente sulla pagina stessa.

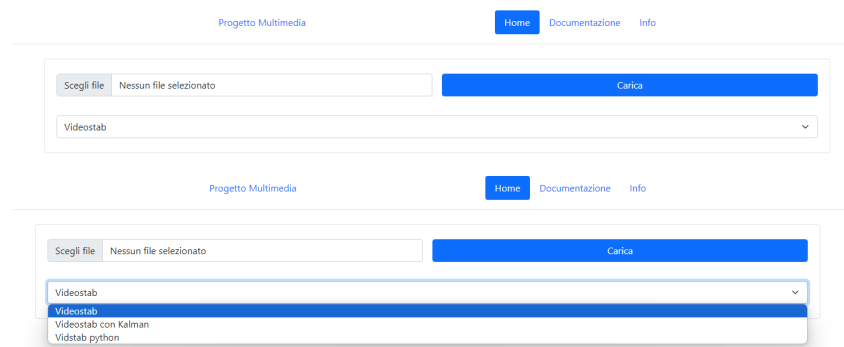


Figura 4.1: Home

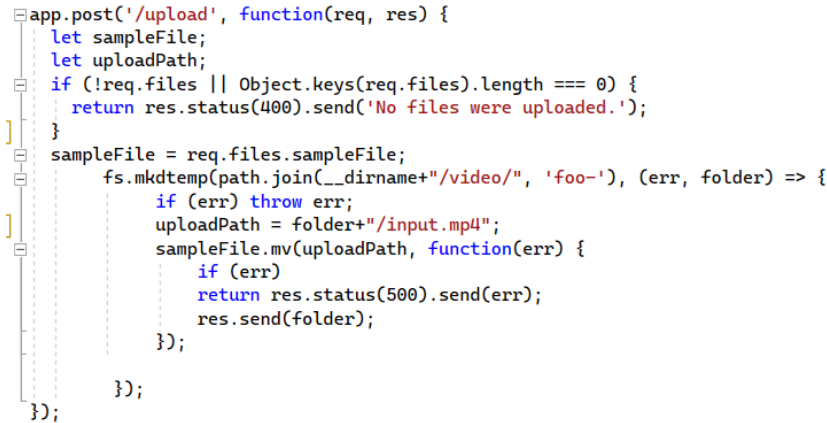
4.1.2 Server

Dal punto di vista del server, il funzionamento è molto più complesso.

Quando un utente si connette alla home viene stabilita una connessione tramite Socket.io.

Al click del pulsante "carica" il client manda una richiesta di tipo "POST" al server, che la gestisce tramite l'API di Express "app.post(...)". Il video viene memorizzato all'interno di una cartella temporanea creata per l'occasione dal metodo "mkdtemp(..)" del modulo **fs**. Ogni richiesta avrà una directory temporanea diversa all'interno della quale verranno memorizzati, oltre al video originale caricato dal client, anche l'output stabilizzato.

```
app.post('/upload', function(req, res) {
  let sampleFile;
  let uploadPath;
  if (!req.files || Object.keys(req.files).length === 0) {
    return res.status(400).send('No files were uploaded.');
```



```
  }
  sampleFile = req.files.sampleFile;
  fs.mkdtemp(path.join(__dirname+'video/', 'foo-'), (err, folder) => {
    if (err) throw err;
    uploadPath = folder+"/input.mp4";
    sampleFile.mv(uploadPath, function(err) {
      if (err)
        return res.status(500).send(err);
      res.send(folder);
    });
  });
});
```

Figura 4.2: Upload

Dopo aver completato l'upload del file, viene restituito all'utente il **percorso della cartella di lavoro** appena creata. A questo punto il client richiama l'evento(Socket.io) "elaboravideo" del server tramite il metodo "mandavideo()" a cui è stato passato come parametro proprio il percorso ricevuto nella risposta precedente.

Il Server, sfrutta i dati ricevuti dal client(che comprendono dunque il percorso della cartella di lavoro e il tipo di algoritmo da utilizzare) e avvia la funzione responsabile dell'elaborazione video.

```
socket.on('elaboravideo', (folder,algorithm) => {
  elabora(folder+"/input.mp4", folder, socket, algorithm);
});
```

Figura 4.3: Event "elaboravideo"

La funzione di elaborazione utilizza la libreria "**child_process**" per eseguire alcuni comandi in thread separati. In particolare, la funzione di elaborazione richiama due comandi shell, il primo esegue l'algoritmo scelto di stabilizzazione, il secondo utilizza il programma **FFMPEG** per aggiungere il corretto codec al video. Una volta terminata l'elaborazione, il client manderà una richiesta di tipo "GET" per richiedere il video elaborato.

```
var comando1= exec(a, (error, stdout, stderr) =>{
  if (error) {
    console.log('stderr: ' + stderr);
    return;
  }else{
    socket.emit("stabilizzato");
    a="ffmpeg -i '"+folder+"/input.mp4' -i '";
    a+=folder+"/outcpp.avi' -vsync 2 -filter_complex hstack '";
    a+=folder+"/output.mp4'";
    var comando2= exec(a, (error, stdout, stderr) =>{
      if (error) {
        console.log('stderr: ' + stderr);
      }else{
        socket.emit("ricevi", folder);
      }
      return;
    });
  }
});
```

```
app.get('/algo',function(req, res) {
  folder= req.query.folder;
  video=folder+"/output.mp4";
  res.sendFile(video);
});
```

Figura 4.4: Funzione di elaborazione e metodo GET

Capitolo 5

Analisi algoritmi

5.1 Videostab

Il progetto è basato sull'algoritmo sviluppato da "Nghia Ho", opportunamente modificato e adattato. Il codice sfrutta la libreria OpenCV e molte delle funzioni introdotte nei capitoli precedenti.

5.2 Funzionamento del Codice

Viene proposto un approccio per la stabilizzazione video (nello specifico per il filtraggio del movimento) chiamato "**Sliding Frame Window**", che consiste nello stimare una traiettoria calcolando il movimento tra fotogrammi continui utilizzando gli algoritmi Shi-Tomasi Corner Detection e Optical Flow per l'intera lunghezza del video.

La traiettoria viene quindi livellata (smussata) utilizzando una media mobile per fornire un risultato stabilizzato. Il funzionamento del codice è molto semplice:

1. Ottiene per ogni frame le trasformazioni(dx, dy, da) dal frame precedente a quello attuale
2. Accumula le trasformazioni per poter ricostruire la traiettoria dell'immagine
3. Smussa la traiettoria utilizzando la media mobile
4. Genera una nuova serie di trasformazioni dal frame precedente a quello attuale, in modo tale che la traiettoria finisca per essere la stessa di quella livellata(smussata)
5. Applica le nuove trasformazioni al video

5.3 Analisi del Codice

Di seguito analizzeremo le parti più interessanti del codice.

```
int main(int argc, char **argv)
{
    if(argc < 2) {
        cout << " ./VideoStab [video.avi]" << endl;
        return 0;
    }
    auto path = filesystem::current_path(); //getting path
    string s=path;
    filesystem::current_path(s+"/video/"); //setting path
    string stringa1=argv[1];
    string stringa2=argv[2];
    string directory=stringa1+" "+stringa2;

    filesystem::path dir1(directory);
    string dir = dir1.parent_path().string();
    string file = dir1.filename().string();
    filesystem::current_path(dir);

    // For further analysis
    ofstream out_transform("prev_to_cur_transformation.txt");
    ofstream out_trajectory("trajectory.txt");
    ofstream out_smoothed_trajectory("smoothed_trajectory.txt");
    ofstream out_new_transform("new_prev_to_cur_transformation.txt");

    VideoCapture cap(file);
    assert(cap.isOpened());
}
```

Figura 5.1: Apertura video

5.3.1 Apertura file in input

In figura 5.1 è presente la parte di codice responsabile dell'apertura del video in input. In questa parte è importante notare le modifiche effettuate tramite la funzione "**Autopath**", necessaria per poter impostare correttamente la cartella di lavoro dalla route di index.js. Intuitivamente il programma prende in input il percorso completo del video caricato per cui è semplice risalire alla cartella in cui è situato.

5.3.2 Ottenere le trasformazioni tra i frame

```
// Step 1 - Get previous to current frame transformation (dx, dy, da) for all frames
vector<TransformParam> prev_to_cur_transform; // previous to current
int k=1;
int max_frames = cap.get(cv::CAP_PROP_FRAME_COUNT);
Mat last_T;
while(true) {
    cap >> cur;
    if(cur.data == NULL) {
        break;
    }
    cvtColor(cur, cur_grey, COLOR_BGR2GRAY);
    vector<Point2f> prev_corner, cur_corner;
    vector<Point2f> prev_corner2, cur_corner2;
    vector<uchar> status;
    vector<float> err;
    goodFeaturesToTrack(prev_grey, prev_corner, 200, 0.01, 30);
    calcOpticalFlowPyrLK(prev_grey, cur_grey, prev_corner, cur_corner, status, err);
    // weed out bad matches
    for(size_t i=0; i < status.size(); i++) {
        if(status[i]) {
            prev_corner2.push_back(prev_corner[i]);
            cur_corner2.push_back(cur_corner[i]);
        }
    }
    // translation + rotation only
    Mat T = estimateRigidTransform(prev_corner2, cur_corner2, false); // false = rigid transform, no scaling/shearing
    // in rare cases no transform is found. We'll just use the last known good transform.
    if(T.data == NULL) {
        last_T.copyTo(T);
    }
    T.copyTo(last_T);
    // decompose T
    double dx = T.at<double>(0,2);
    double dy = T.at<double>(1,2);
    double da = atan2(T.at<double>(1,0), T.at<double>(0,0));
    prev_to_cur_transform.push_back(TransformParam(dx, dy, da));
    out_transform << k << " " << dx << " " << dy << " " << da << endl;
    cur.copyTo(prev);
    cur_grey.copyTo(prev_grey);
    k++;
}
```

Figura 5.2: Traiettorie video

Il codice nella figura 5.2 utilizza i metodi della libreria OpenCV già descritti nei capitoli precedenti.

Per prima cosa è necessario analizzare ciascun frame del video, per questo motivo si utilizza un ciclo while all'interno del quale ogni frame subisce le trasformazioni elencate nel capitolo 2.1.

Nel dettaglio "**goodFeaturesToTrack()**" prende in input un'immagine in scala di grigi ottenuta grazie al metodo "**cvtColor()**".

Una volta stabilito il flusso ottico tramite la funzione "**calcOpticalFlowPyrLK()**" è sufficiente salvare le informazioni ottenute in appositi vettori ("**prev_to_cur_transform**")

```
// Step 3 - Smooth out the trajectory using an averaging window
vector<Trajectory> smoothed_trajectory; // trajectory at all frames
for(size_t i=0; i < trajectory.size(); i++) {
    double sum_x = 0;
    double sum_y = 0;
    double sum_a = 0;
    int count = 0;
    for(int j=-SMOOTHING_RADIUS; j <= SMOOTHING_RADIUS; j++) {
        if(i+j >= 0 && i+j < trajectory.size()) {
            sum_x += trajectory[i+j].x;
            sum_y += trajectory[i+j].y;
            sum_a += trajectory[i+j].a;
        }
        count++;
    }
    double avg_a = sum_a / count;
    double avg_x = sum_x / count;
    double avg_y = sum_y / count;
    smoothed_trajectory.push_back(Trajectory(avg_x, avg_y, avg_a));
    out_smoothed_trajectory << (i+1) << " " << avg_x << " " << avg_y << " " << avg_a << endl;
}
```

Figura 5.3: Traiettorie video

5.3.3 Livellare la traiettoria

Il codice per livellare la traiettoria nella figura 5.3 è semplice e intuitivo, per ogni componente (x,y,a) si calcola la media. La nuova traiettoria è salvata in "smoothed_trajectory".

5.3.4 Generazione di una nuova serie di trasformazioni

Da un punto di vista tecnico il codice in figura 5.4 è facilmente comprensibile. Per prima cosa si calcola la differenza tra le x , y e a della traiettoria originale e di quella livellata. Successivamente si somma questa differenza alla prima per ottenere, come introdotto precedentemente, che la traiettoria originale finisca per essere la stessa di quella smussata.

```
for(size_t i=0; i < prev_to_cur_transform.size(); i++) {
    x += prev_to_cur_transform[i].dx;
    y += prev_to_cur_transform[i].dy;
    a += prev_to_cur_transform[i].da;
    // target - current
    double diff_x = smoothed_trajectory[i].x - x;
    double diff_y = smoothed_trajectory[i].y - y;
    double diff_a = smoothed_trajectory[i].a - a;
    double dx = prev_to_cur_transform[i].dx + diff_x;
    double dy = prev_to_cur_transform[i].dy + diff_y;
    double da = prev_to_cur_transform[i].da + diff_a;
    new_prev_to_cur_transform.push_back(TransformParam(dx, dy, da));
    out_new_transform << (i+1) << " " << dx << " " << dy << " " << da << endl;
}
```

Figura 5.4: Traiettoria video

5.4 VideostabKalmn

Questo algoritmo è del tutto identico a quello precedente con la differenza che utilizza un filtro Kalman al posto della "Sliding Frame Window" dell'algoritmo originale. Di seguito la parte principale del codice che implementa il filtro:

```
//
if(k==1){
    // initial guesses
    X = Trajectory(0,0,0); //Initial estimate, set 0
    P = Trajectory(1,1,1); //set error variance, set 1
}
else
{
    //time update"prediction"
    X_ = X; //X_(k) = X(k-1);
    P_ = P+Q; //P_(k) = P(k-1)+Q;
    // measurement update"correction"
    K = P_/(P_+R); //gain;K(k) = P_(k)/(P_(k)+R);
    X = X_+K*(z-X_); //z-X_ is residual,X(k) = X_(k)+K(k)*(z(k)-X_(k));
    P = (Trajectory(1,1,1)-K)*P_; //P(k) = (1-K(k))*P_(k);
}
```

Figura 5.5: Filtro Kalman

5.5 Videostab python

L'ultimo algoritmo implementato sfrutta la libreria "**Vidstab**", una libreria python creata a partire dall'algoritmo originale. Il funzionamento è del tutto identico al primo algoritmo ma implementa una migliore gestione della memoria e un sistema di plottaggio che permette di analizzare in dettaglio come agisce il filtraggio del movimento.

Capitolo 6

Conclusioni

6.1 Confronto degli algoritmi

In questo capitolo verranno comparati i tre algoritmi descritti precedentemente. I criteri presi in considerazione sono i seguenti:

1. Percezione visiva dell'utente finale
2. Filtraggio del movimento
3. Tempo di elaborazione
4. Trasformazioni tra i frame
5. Gestione della memoria

6.1.1 Percezione visiva dell'utente finale

Tutti gli algoritmi forniscono un output perfettamente stabilizzato. La differenza tra gli output non è percepibile da parte dell'utente finale, tuttavia la libreria OpenCV permette di generare un output all'interno del quale è possibile vedere il movimento del video stesso (il video è circondato da **bordi neri** che permettono di vedere le trasformazioni che subisce). Focalizzandoci su quest'ultimo è possibile notare come **VideostabKalman** abbia un movimento visibilmente più fluido.

6.1.2 Filtraggio del movimento

Come precedentemente introdotto, Videostab e Videostab Python utilizzano la tecnica "**Sliding Frame Window**" per il filtraggio del movimento che differisce dal filtro di Kalman utilizzato dal terzo algoritmo.

Ogni algoritmo è inoltre dotato di una funzione di plotting che permette di visualizzare come varia la traiettoria del video prima e dopo l'applicazione della funzione di smussatura. Di seguito viene riportata la differenza nella traiettoria ottenuta dallo stesso video input denominato "inputA"

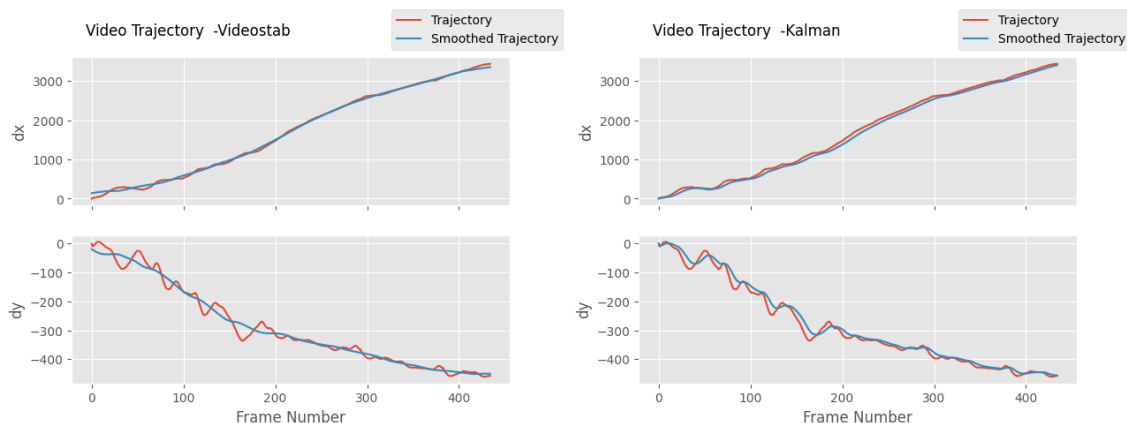


Figura 6.1: Confronto tra Videostab e VideostabKalman

In figura 6.1 è possibile vedere facilmente come la traiettoria viene smussata nei due algoritmi. Nonostante il risultato sia molto simile, è facilmente intuibile come il filtro di media agisce in Videostab (basta guardare la linea blu di dy). Il filtro di Kalman d'altro canto, introduce una smussatura più "netta" nella componente dx.

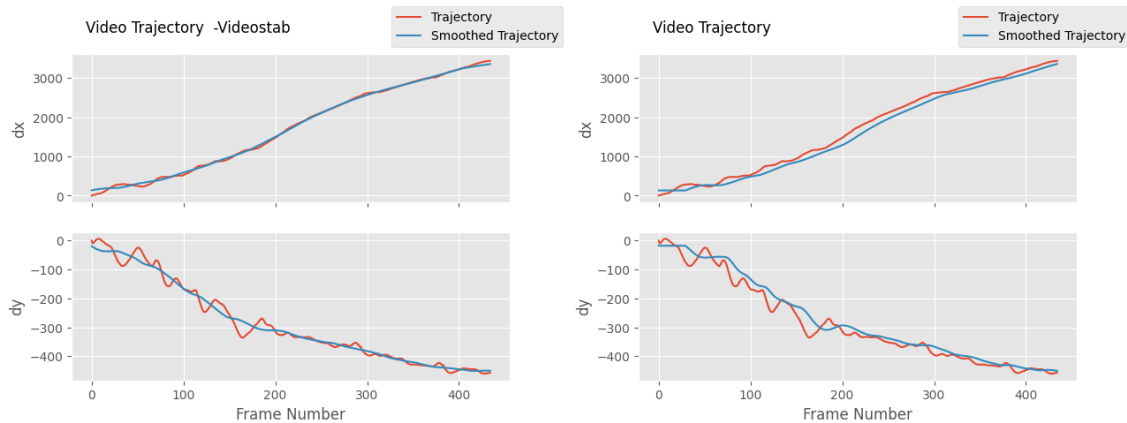


Figura 6.2: Confronto tra Videostab e Videostab Python

È interessante notare come nonostante Videostab e Videostab Python utilizzino la stessa tecnica di smussatura abbiano risultati leggermente diversi, probabilmente dovuti alla diversa implementazione.

6.1.3 Tempo di Elaborazione

Su uno stesso video in input della durata di 15 secondi sono state osservate le seguenti tempistiche:

- Videostab: 50 secondi circa
- VideostabKalman: 30 secondi circa
- Videostab Python: 40 secondi circa

E' opportuno ricordare che molti fattori possono influire sulle tempistiche ottenute quali ad esempio la potenza della cpu. Queste misure sono state osservate su una macchina virtuale Ubuntu dotata di processore Intel i78565U e 8 giga di ram.

6.1.4 Trasformazioni tra i frame

Di seguito viene riportata la differenza in termini di trasformazioni tra i frame ottenuti dallo stesso video input denominato "inputB".

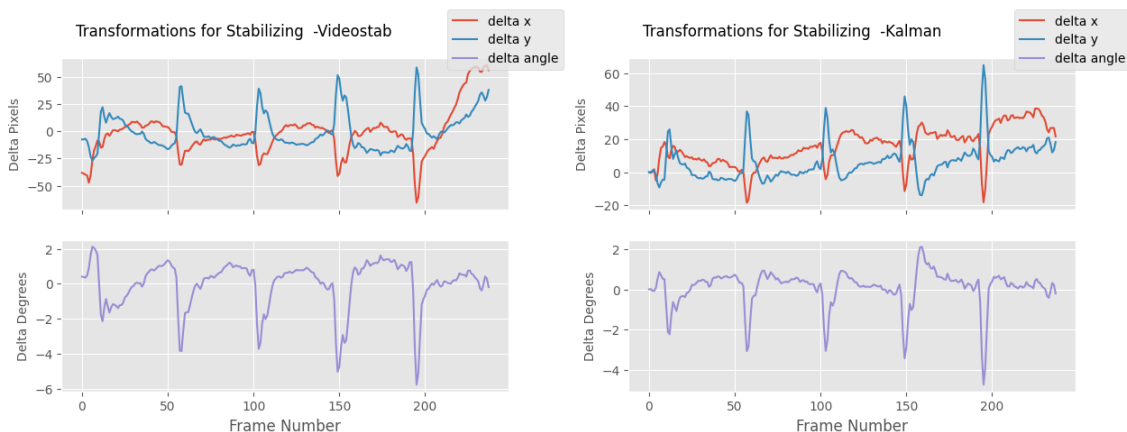


Figura 6.3: Confronto tra Videostab e Videostab Python

6.1.5 Gestione della memoria

Un dettaglio fondamentale da tenere in mente utilizzando la web application è la gestione della memoria. Nello specifico gli algoritmi scritti in C (**Videostab** e **VideostabKalman**) hanno un problema che impedisce loro di elaborare video di grandi dimensioni ($> 30Mb$).

Questo perché per effettuare le varie trasformazioni necessitano di caricare tutto il video in memoria. Videostab Python invece, ha una gestione della memoria nettamente migliore dovuto alla diversa implementazione, che permette di stabilizzare video di qualsiasi dimensione e lunghezza.

6.2 Conclusione

Escludendo Videostab Python, che è da considerarsi come un'implementazione ottimale di Videostab, possiamo notare come ambedue gli algoritmi (Videostab e VideostabKalman) siano ottimi per la stabilizzazione video ed entrambi gli approcci forniscono all'utente finale un output perfettamente stabilizzato.

Da un punto di vista tecnico è necessario ribadire come **il filtro di Kalman** costituisca una funzione di smussatura che conduce ad una traiettoria leggermente più "fluida" (cap. 6.1.1).