UNIVERSITÀ DEGLI STUDI DI TRENTO

Dipartimento di Ingegneria e Scienza dell'Informazione

Corso di Laurea Triennale in

Ingegneria dell'Informazione e Comunicazioni

Progetto sul corso di Elaborazione e Trasmissione d'Immagini – Computer Vision

Stima della densità del traffico navale a Venezia

Supervisore:

Prof. Nicola Conci

Studenti:

Federico Favia, Martin De Pellegrini

179135, 180785

Anno Accademico

2017 - 2018

Indice

Introduzione

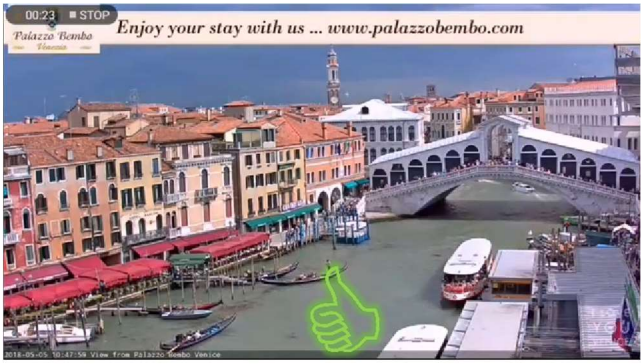
Abbiamo scelto di svolgere la modalità a progetto dell'esame di Elaborazione e Trasmissione d'Immagine con la finalità di vedere un'applicazione pratica delle teorie studiate a lezione e metterci così alla prova. Ci siamo subito indirizzati verso la macro-area della Computer Vision, supervisionata dal Prof. Nicola Conci, e tra i temi proposti abbiamo optato per l'analisi del traffico navale dei canali di Venezia tramite librerie di Computer Vision e linguaggio di programmazione C++. Dopo una riflessione col Prof., essendoci altri gruppi al lavoro su questa tematica, ci siamo focalizzati, piuttosto che sull'analisi di traiettorie delle barche o di una classificazione delle stesse, su una stima della densità del traffico.

Obiettivo del progetto

L'obiettivo del progetto è stato dunque analizzare i video dei canali della città lagunare ed elaborare un algoritmo che potesse estrarre da essi una semantica sulla stima del traffico delle barche transitanti, una questione problematica molto attuale e che vorrebbe arrivare a una regolarizzazione.

Scelta dei video da analizzare

Innanzitutto abbiamo scelto i video selezionandoli tra i link di YouTube delle live webcam dei canali di Venezia: ci siamo focalizzati sulla diretta video del Canal Grande[1] e di quella del Ponte di Rialto da Palazzo Bembo[2] (https://www.youtube.com/watch?v=vPbQcM4k1Ys). Dopo lunghe prove abbiamo condiviso col Prof. la decisione di usare i video provenienti dalla seconda telecamera perché migliori dal punto di vista dell'analisi che ci serviva (posizione della telecamera e area di interesse del video); in ogni caso entrambe le riprese sono turistiche e quindi non sono le migliori per fare analisi tecniche di video processing. Tramite il *sw* MediaInfo abbiamo verificato che i video sono



circa a 30 FPS. Per registrare video.mp4 di durata di 5 minuti abbiamo usato il *tool* 'registrazione schermo' dei nostri smartphone piuttosto che dei plug-in per browser, con lo scopo di ridurre i tempi.

Strumenti di programmazione e altri software utilizzati

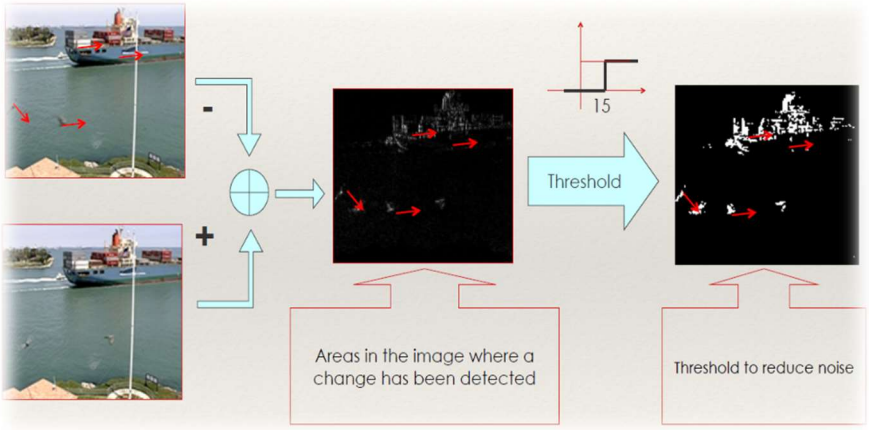
Per quanto riguarda l'ambiente di programmazione, abbiamo lavorato tramite macchina virtuale sul sistema operativo Linux Ubuntu con l'editor C++ Eclipse, con preinstallate le librerie di OpenCV 2.4. Dopo aver seguito alcuni tutorial per familiarizzare con questo nuovo ambiente di sviluppo e con le funzioni di questa libreria open source, una delle più utilizzate per la Computer Vision, da come compilare correttamente un progetto alle principali metodi di *image processing*, il Prof. ci ha fornito del materiale del Dottorando Niccolò Bisagno sui principali algoritmi di *Motion Detection*, ovvero il riconoscimento del movimento degli oggetti in una scena di un video. Per l'analisi degli output del nostro algoritmo C++ ci siamo invece affidati al software di calcolo matriciale MATLAB, il cui script che abbiamo realizzato produce a sua volta dei grafici di interesse sul nostro problema indagato.

Algoritmi di Motion Detection

Siccome il nostro obiettivo è stimare il traffico nautico di Venezia a partire da video di telecamere statiche, ci siamo dovuti rapportare ai principali algoritmi di *Motion Detection*:

- Frame differencing
- Background subtraction
- **Adaptive background subtraction (scelto)**
- Mixture of Gaussians

Abbiamo escluso l'approccio *MoG,* che si basa sul descrivere la storia del modello dello sfondo come somma di distribuzioni gaussiane ciascuna con un certo peso, perché non produceva risultati con i nostri video di input. Ci siamo poi concentrati sul *frame differencing* poiché oltre ad essere una tecnica semplice e intuitiva, funzionava bene sul nostro primo video di input (che però risultava essere quello della videocamera meno significativa da analizzare[1]). Il 1° passo è sottrarre dall'intensità dei pixel del video al frame (convertito in GrayScale) al tempo T quella dei pixel del frame al tempo T+N, rivelando così le aree della scena in cui è stato individuato un cambiamento (quindi un movimento di un oggetto nel nostro caso).



$$D(N) = \| I(t) - I(t+N) \| \quad \rightarrow \quad \texttt{absdiff(I(t),I(t+N),result)}$$

Poi si applica una soglia

```
threshold(InputArray src, OutputArray dst, double thresh, double maxval, int type)
```

per distinguere movimenti reali dal rumore: i frame potrebbero infatti differire di tanto, sia in scene indoor che outdoor (il nostro caso), specialmente per via di cambiamenti della luce o di altre condizioni. Il risultato finale è un'immagine binaria dove i pixel bianchi corrispondono ad oggetti in movimento. La principale difficoltà nella progettazione di un robusto algoritmo di *background subtraction* è proprio la selezione della soglia. Il *frame differencing* si comporta male se lo sfondo non è realmente statico (ad es. foglie svolazzanti, onde d'acqua – proprio il nostro caso – …), e quindi, storicamente, sono poi stati sviluppati metodi basati su modelli di background probabilistici. Quando infatti abbiamo sperimentato i video della seconda ripresa[2], quella migliore, il *frame differencing* ci dava qualche problema e, d'accordo col Prof., ci siamo concentrati su un metodo simile, detto *adaptive background subtraction*, che in effetti funziona meglio con entrambe le visuali, grazie ad un parametro di adattamento α.

Adaptive background subtraction

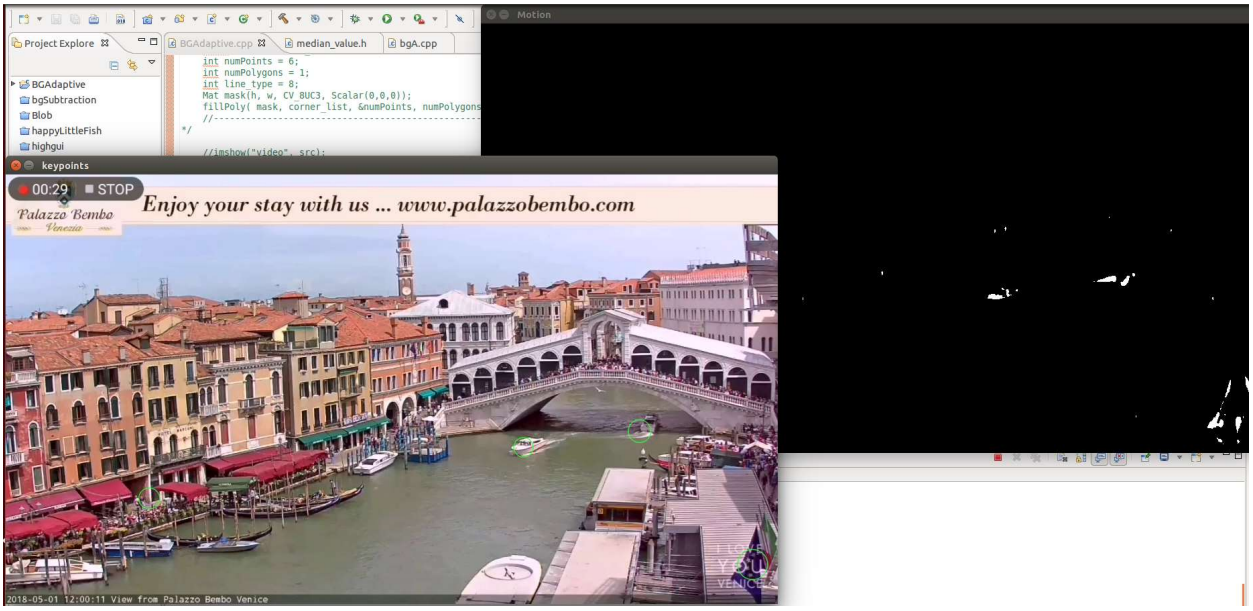
La novità di questo algoritmo consiste nell'aggiunta di una fase di allenamento e di una di aggiornamento del modello di background (B_t) tramite un parametro di aggiornamento α, detto *learning rate*, che determina i contributi sommati dell'immagine al frame corrente (αI_t) e del background al frame precedente ($(1-\alpha)B_{t-1}$).

$$B_t = \alpha I_t + (1-\alpha) B_{t-1} \begin{cases} \alpha = 0 \rightarrow \textit{bg sub}, \text{ no update} \\ \alpha = 1 \rightarrow \textit{frame differencing} \end{cases}$$

Se α=0, non c'è nessun aggiornamento e il background rimane costante allo scorrere dei frame. Se 0<α<1 (il caso del nostro codice) il background viene aggiornato gradualmente in base ad α, cioè l'immagine al frame corrente viene considerata parte del background a seconda del suo peso. All'aumentare di α aumenta la scorrelazione tra l'immagine corrente e il background al frame precedente; in particolare se α=1 il background viene sostituito dall'immagine al frame corrente e il principio è quello del *frame differencing*, strada da noi esclusa.

Nel nostro codice mostrato qui sotto, vediamo come per 20 frame (`training_nr`) viene allenato un modello di sfondo iniziale, mentre per i restanti viene applicata la sottrazione tra lo sfondo (`bg`) e l'immagine corrente (`tmp`), la sogliatura e l'aggiornamento dello sfondo. Nel nostro caso, tramite un processo *trial-and-error*, abbiamo trovato un *learning rate* α=0.02 che soddisfa i nostri requisiti (poi alla fine del report ci saranno altre considerazioni).

```
int tc=0;
const int training_nr = 20;
if(tc<training_nr){
        bg_train(tmp, &bg); // bg training
        tc++;
}else{
        absdiff(bg,tmp,motion); //bg subtraction
        threshold(motion, motion_mask, 50,255, THRESH_BINARY); //mask thresholding
        bg_update(tmp, &bg, motion_mask, 0.02); //incremental bg update
        …… }
```



Maschera ROI

Dai risultati dell'*adaptive background subtraction* ci siamo accorti come alcune zone della scena fossero molto rumorose per via del movimento di pedoni e/o volatili, oltre a essere inutili al fine del nostro obiettivo, ovvero la stima del traffico nautico. Abbiamo risolto il problema definendo una ROI (Region Of Interest) tramite un poligono di 8 lati, i quali vertici sono stati individuati in maniera sperimentale. Questa maschera azzera i pixel esterni

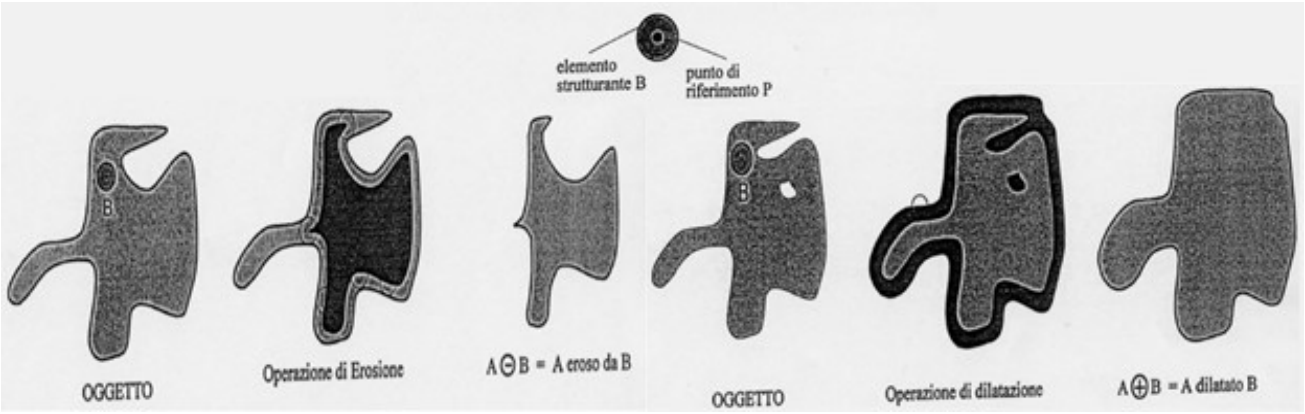
al poligono, lasciando visibile solo il canale in cui si vedono passare le imbarcazioni: dunque l'algoritmo di *motion detection* opera solo su questa area.

```
Point corners[1][6];
corners[0][0] = Point(0,640);
corners[0][1] = Point(950,640);
corners[0][2] = Point(1062,340);
corners[0][3] = Point(950,316);
corners[0][4] = Point(800,316);
corners[0][5] = Point(719,340);
const Point* corner_list[1] = {corners[0] };
int numPoints = 6;
int numPolygons = 1;
int line_type = 8;
Mat mask(h, w, CV_8UC3, Scalar(0,0,0));

fillPoly( mask, corner_list, &numPoints, numPolygons, Scalar(255, 255, 255), line_type);

for (int i = 0; i < nFrames; i++) {
       currentFrame = i+1;

       cap >> myPicture;
       bitwise_and(myPicture, mask, tmp);
       cvtColor(tmp,tmp,CV_RGB2GRAY);
       GaussianBlur(tmp, tmp, Size(3, 9), 20, 20);

       [Algoritmo di background subtraction]

       ……
}
```

Inoltre abbiamo utilizzato un filtro Gaussiano per rendere *smooth* la scena, pulendo gli effetti molto rumorosi causati dal movimento delle onde.

Filtri morfologici

In uscita dall'algoritmo di *background subtraction* abbiamo notato che non tutto il rumore filtrato è stato eliminato, in particolare quello causato dall'oscillazione di barche ormeggiate. A questo scopo sono stati utilizzati dei filtri morfologici elementari di **erosione** e in seguito di **dilatazione**: essi sono un particolare tipo di filtri non-lineari che operano sulla base della forma della maschera, che nel nostro caso vengono modellati come oggetti 2D in quanto sono stati applicati ad un'immagine binaria. Il principio di funzionamento di questi due filtri è il seguente: viene definito un elemento strutturale B con un determinato punto di riferimento P per entrambi gli operatori.



$$A \ominus B = \{x \mid B_x \subset A\}$$

$$A \oplus B = \{x \mid B_x \cap A \neq 0\} = \{x \mid B_x \cap A \subset A\}$$

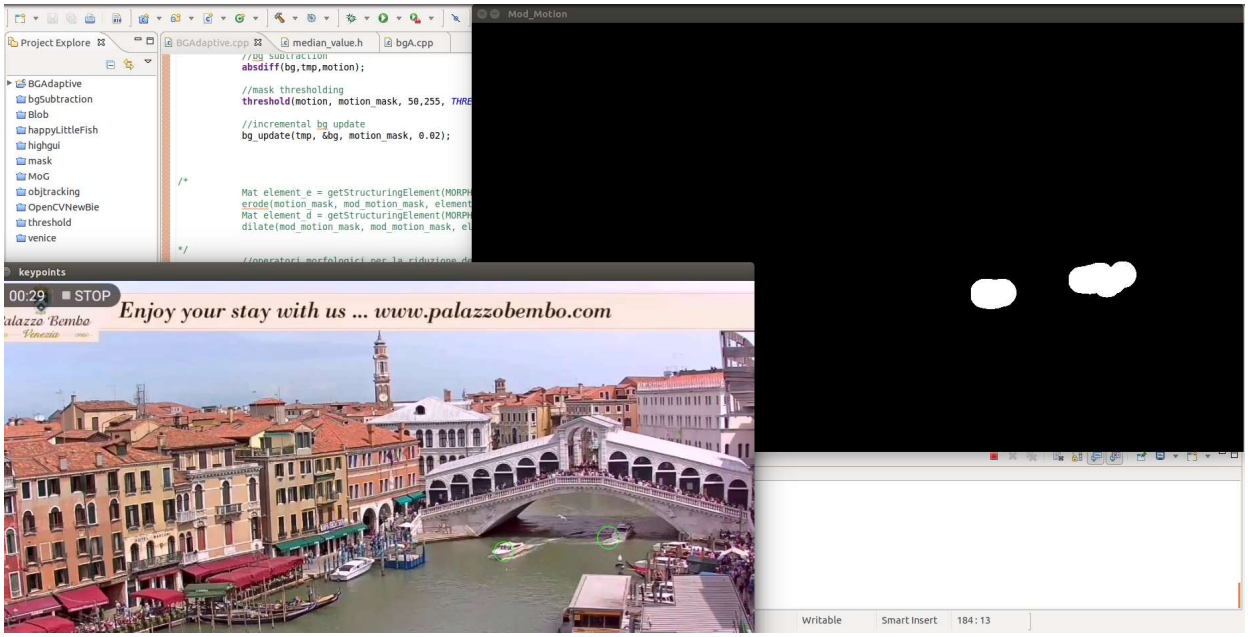
Quando si applica l'**erosione**, che ha l'effetto di erodere i contorni delle regioni, l'elemento strutturale B posiziona il suo punto di riferimento all'interno dell'oggetto di interesse in modo tale da scorrere tangenzialmente al contorno della figura: quando l'elemento strutturante viene traslato vicino ai bordi esso può non essere completamente contenuto, e vengono dunque eliminati i pixel dell'oggetto che non riescono a contenerlo. Nel nostro codice erodiamo con un ellisse per eliminare piccoli pixel bianchi di rumore dovuti a piccoli spostamenti.

```
Mat element_e = getStructuringElement(MORPH_ELLIPSE,Size(5,3),Point(2,2));
erode(motion_mask, mod_motion_mask, element_e);
```

Nel caso della **dilatazione**, il cui effetto è di espandere gli oggetti in un'immagine, riempire le lacune di piccola dimensione e connettere oggetti separati da una distanza minore della dimensione della matrice di B, l'elemento strutturale viene fatto scorrere in modo che il punto di riferimento sia sul contorno dell'immagine. Nel nostro codice dilatiamo per unire gruppi separati di pixel bianchi dovuti al movimento di una singola barca; infatti per il passaggio successivo di *blob detection* è necessario rilevare gruppi di pixel bianchi connessi con una certa area.

```
Mat element_d = getStructuringElement(MORPH_ELLIPSE,Size(45,35),Point(2,2));
dilate(mod_motion_mask, mod_motion_mask, element_d);
```

Normalmente eseguire in successione erosione e dilatazione definisce un'operazione di **apertura**, ma nel nostro caso non è propriamente così perché abbiamo definito due elementi strutturali di dimensioni diverse; ovviamente sono state fatte parecchie prove prima di trovare la combinazione ottimale, che però dipende dal tipo di video in input (angolazione e velocità di riproduzione).



Identificazione e conteggio degli oggetti in movimento

Su questa immagine in bianco e nero viene eseguito il riconoscimento e il conteggio degli oggetti: si possono usare varie strade. Una è quella di usare una linea di attraversamento, dopo aver usato una tecnica di *labeling* grazie alla libreria *cvblob*. Infatti, una volta eseguito il riconoscimento dell'oggetto è necessario seguirlo, ovvero verificare che l'oggetto sia lo stesso da un fotogramma all'altro. Per determinare ciò, bisogna calcolare il rapporto tra il centro dell'oggetto rispetto alla posizione e la variazione di dimensione dell'oggetto dal frame seguente. Infine si può determinare una linea di acquisizione e ogni volta che un oggetto la attraversa in qualsiasi direzione, un contatore può essere

aumentato; considerando la direzione dell'oggetto rispetto alla linea, verrà aumentato un contatore diverso. Essendo questa strada troppo complicata per il nostro tipo di problema, ci siamo concentrati sull'identificare i *blob* (gruppi di pixel bianchi degli oggetti in movimento) con il metodo *SimpleBlobDetector* (già incluso nella libreria OpenCv).

Blob detection

Come accennato prima, un *blob* è un gruppo di pixel connessi in un'immagine che condividono alcune proprietà comuni (e.g. valore GrayScale). Nell'immagine a fianco, le aree scure sono *blob* e l'obiettivo della *blob detection* è quindi identificare e contrassegnare queste regioni. Nel nostro codice, i *blob* sono invece i gruppi di pixel bianchi ottenuti erodendo e dilatando i frame risultanti dall'*adaptive background subtraction*. Fortunatamente OpenCv fornisce un modo comodo per rilevare i *blob* e filtrarli in base a caratteristiche diverse (una delle principali è l'area): il *SimpleBlobDetector*. Quest'ultimo, come suggerisce il nome, si basa su un algoritmo piuttosto semplice descritto di seguito. L'algoritmo è controllato da dei parametri (in grassetto) e ha i seguenti passaggi:

- **Thresholding:** converte le immagini sorgente in diverse immagini binarie sogliandole con valori a partire da *minThreshold*. Queste soglie vengono incrementate tramite *thresholdStep* fino ad arrivare a *maxThreshold*. Quindi la prima soglia è *minThreshold*, la seconda è *minThreshold* + *thresholdStep*, la terza è *minThreshold* + *2* · *thresholdStep* e così via.
- **Raggruppamento:** in ogni immagine binaria, i pixel bianchi connessi sono raggruppati: chiamiamo questi blob binari.
- **Unione:** vengono calcolati i centri dei blob binari nelle immagini binarie e vengono uniti i blob situati più vicino di *minDistBetweenBlobs*.
- **Calcolo del centro e del raggio:** i centri e i raggi dei nuovi blob uniti vengono calcolati e restituiti.

I parametri per il *SimpleBlobDetector* possono essere settati per filtrare il tipo di *blob* che desideriamo:

- Per **colore**: blobColor = **0** per selezionare blob **scuri**, blobColor = **255** per blob **chiari**, il nostro caso.
- Per **dimensioni**: valori appropriati per **minArea** e **maxArea**; ad esempio nel nostro caso settando minArea = 1000 vengono filtrati via tutti i blob che hanno meno di 1000 pixel, una dimensione appropriata per un'imbarcazione nella scena inquadrata.
- Per **forma** (1-circolarità; 2-convessità; 3-rapporto d'inerzia): questi parametri non sono di nostro interesse.

```
FILE "blob.h"

double to_blob(Mat& img_blob, Mat& original){
	//1^ PARTE SUI BLOB: setup parameters
	SimpleBlobDetector::Params params;

	//change color
	params.filterByColor = 1;
	params.blobColor = 255;
```

```
        //Change thresholds
        //params.minThreshold = 10;
        //params.maxThreshold = 200;

        //Filter by Area.
        params.filterByArea = true;
        params.minArea = 1000;
        params.maxArea = 10000;

        //Filter by Circularity
        params.filterByCircularity = false;
        //params.minCircularity = 0.001;

        //Filter by Convexity
        params.filterByConvexity = false;
        //params.minConvexity = 0.001;

        //Filter by Inertia
        params.filterByInertia = false;
        //params.minInertiaRatio = 0.3;

        //2^ PARTE SUI BLOB -> Set up detector with params
        SimpleBlobDetector detector(params);

         //Detect blobs
        std::vector<KeyPoint> keypoints;
        detector.detect(img_blob, keypoints);

        //Draw detected blobs as coloured circles
        Mat im_with_keypoints;
        drawKeypoints(original, keypoints, im_with_keypoints,Scalar(255,0,0),
DrawMatchesFlags::DRAW_RICH_KEYPOINTS);

         //Show blobs
        imshow("keypoints", im_with_keypoints );

        //#blob presenti in quel frame
        return keypoints.size();
}
```

Una volta settato i parametri del nostro *SimpleBlobDetector* abbiamo notato che il codice faceva molta fatica a identificare i blob corrispondenti a barche molto grosse, come se ci fosse un limite superiore non specificato; una volta settato maxArea = 10000 abbiamo risolto questa problematica.

Conteggio dei blob

Per contare il numero dei blob abbiamo sfruttato il fatto che *keypoints* sia il vettore dei blob identificati in ogni frame e dunque con la funzione `return keypoints.size();` otteniamo la dimensione di tale vettore, cioè il numero dei blob in ogni frame, il nostro obiettivo.

Stima del traffico

Per raccogliere dati e interpretarli per dare un'idea della stima del traffico abbiamo ragionato principalmente su 3 approcci, tutti implementati in Matlab (i primi 2 anche in C++ per darne una prima vista in output):

- Media integrale
- Mediana
- Flusso (limite del rapporto incrementale)

Media integrale

$$\frac{1}{T}\int_0^T f(t)dt$$

Naturalmente, essendoci un certo numero di barche che navigano nel canale durante il video e avendo a disposizione il numero di barche (approssimato perché è comunque una scena rumorosa) contate in ogni frame, abbiamo subito pensato di ricavarci la **media integrale** del numero di barche contate fino ad un certo frame. Per fare questo abbiamo dunque sommato il numero di blob man mano che passavano i frame, per poi dividere il risultato per il numero del frame corrente. Per semplificare abbiamo svolto questa operazione ogni 30 FPS (frame rate dei video in input), ottenendo dunque l'andamento della media integrale delle barche contate al passare dei secondi. Si noti però che questo dato non equivale al numero di barche passate nella scena ma solo di quelle contate mediamente: ci dà quindi un'idea generale sulla quantità del traffico.

```
[operazioni di erosione e dilatazione]

//Blob tracking and counting algorithm
nBlob = to_blob(modif_motion_mask,tmp); //1st param the pic where detect blobs, 2nd on
which pic show blobs
//cout << "nBlob=" << nBlob << endl;

//Mobile Average algorithm
totBlob = totBlob + nBlob;

[…]

if(x == fps){ //output media e mediana ogni fps frame
       media = (totBlob / currentFrame); //media integrale aggiornata al currentFrame
       cout<<"Contate finora una MEDIA di "<< media << " barche dopo " <<
currentFrame/fps <<" sec"  << endl;
       […]
       x=0; //riaggiorno
else{
       x++;
}
```

Mediana

In statistica, in particolare quella descrittiva, data una distribuzione di un carattere quantitativo ordinabile si definisce **valore mediano** il valore assunto dalle unità statistiche che si trovano nel mezzo della distribuzione. Nel nostro codice l'abbiamo calcolata con una funzione che, una volta inserito in un vettore il numero dei blob al passare dei frame, lo ordina e trova il valore centrale se la dimensione è dispari, o fa la media dei due valori centrali se sono dispari. Spesso veniva un risultato più alto di quello della media dopo lo stesso numero di secondi, un risultato comunque più attinente alla realtà, perché la mediana, sebbene generalmente meno informativa della media, è meno sensibile alla presenza di *outliers* rumorosi (valori estremi o anomali).

```
FILE "median_value.h"

double median_value(vector<double> _v){
       double median;
       sort(_v.begin(),_v.end());
       int mid=(_v.size())/2;
       if ((_v.size()) % 2 != 0){
              median= _v[mid];
       }else{
              median= (_v[mid-1]+_v[mid])/2;
       }
       return median;
}
```

```
FILE main.cpp

if(x == fps){ //output media e mediana ogni fps frame
	[…]
	v.push_back(nBlob); //inserimento nel vettore per mediana
	cout<<"Contate finora una MEDIANA di "<< median_value(v) << " barche dopo " <<
currentFrame/fps <<" sec"  << endl; //mediana
	[…]

	cout <<"median = "<< median_value(v) << endl;
	x = 0;
}
else{
	x++;
}
```

Flusso, flusso medio e andamento del flusso

Quando ci siamo interrogati a fondo sul significato di stima del traffico navale, ci siamo accorti di come media e mediana non fossero dei descrittori matematici/statistici sufficienti a dare un'idea complessiva della situazione. Volevamo infatti dare delle informazioni sul **flusso**, ovvero sul numero di "barche/s". Immaginiamo infatti di avere due video ad orari diversi, uno con una media che si attesta su 3 barche ed un altro che si attesta su 6 barche; normalmente ci verrebbe da dire che il secondo sia più trafficato, ma bisognerebbe verificare anche l'andamento con cui il canale (nella scena inquadrata) viene liberato dalle barche. Se il flusso aumenta significa che ci sono più barche che entrano rispetto a quelle che escono, mentre se diminuisce il contrario. Chiaramente questo è un dato che non ha senso osservare da solo, ma solo in associazione con la media integrale e/o mediana.

```
%% FLUSSO O BARCHE/S
for i = 1:length(media_barche)-1
    flusso(i,:) = (media_barche(i+1)-media_barche(i))/(1/fps); % diviso
l'intervallo di tempo che è (1/fps) in secondi
end
*flusso_medio = mean(flusso);
```

Nel nostro script Matlab calcoliamo il flusso di barche come rapporto incrementale della media integrale delle barche (in questo caso quella calcolata a passi di 1 frame): cioè sottraiamo al numero dei blob contati mediamente fino ad un certo frame quello del frame precedente, dividendo per il Δt, in questo caso uguale a 1/FPS. Il *plotting* di questo grafico mostra dunque la "derivata" della media integrale mostrandone il valore ogni 1/FPS secondi. Abbiamo scelto di fare il flusso della media e non del numero dei blob perché è un dato più *smooth* senza troppi outliers. Su molti video analizzati il flusso si stabilizza attorno allo zero (lo mostriamo calcolando anche il **flusso medio***) non mostrando significative differenze tra fasce orarie o giornate diverse, in quanto servirebbero grandi quantità di dati per ricavarne delle statistiche. Un flusso che si attesta attorno allo zero ha il significato che escono dal canale circa lo stesso numero di barche che entrano. Abbiamo dunque migliorato questo dato plottando un **andamento del flusso**, tramite un'interpolazione polinomiale di 3° grado, di cui non abbiamo però registrato la bontà.

$$f(t) = \frac{f(t + \Delta t) - f(t)}{\Delta t}$$

```
p=polyfit(time(2:end),flusso,2) %polinomio che fitta di ordine 3
y1=polyval(p,time(2:end));
subplot(1,3,3), hold on, grid on
plot(time(2:end),flusso,'o')
plot(time(2:end),y1,'LineWidth',2)
ylabel('flusso');
xlabel('t[s]')
title('Andamento del flusso')
```

Grafici di Matlab

Inizialmente avevamo pensato ad implementare i grafici grazie ad alcune funzioni grafiche di OpenCv direttamente dal codice sorgente, in modo che essi venissero generati "real-time" appena terminata l'analisi del video, ma dopo essere andati incontro a varie difficoltà abbiamo preferito la strada del post-processing dei dati grazie al software Matlab. In questo caso la controindicazione è stata quella del passare da ambiente Linux a Windows, poichè usiamo Matlab su quest'ultimo sistema operativo.

```
vector<double> vv;
ofstream myfile;
myfile.open("nBlob.txt"); //creazione file txt
[…]
vector<double>::iterator vi; //export nBlob vector passing frames

for(vi=vv.begin(); vi!=vv.end(); vi++){
      myfile << *vi << "\n";
}
myfile.close();
return 0;
}
```

In sintesi esportiamo dal codice C++ un file "nBlob.txt" su cui viene scritto il vettore contenente il numero dei blob contati al passare di tutti i frame, file che, se a sua volta viene processato da Matlab, produce in output 2 *immagini*.png, per un totale di 5 grafici:

- *frame_plots.*: *a*) numero dei blob al passare di ogni frame.

  b) media integrale e mediana del numero dei blob ogni FPS frame (cioè ogni secondo)

- *time_plots*: *a*) media integrale e mediana del numero dei blob al passare dei secondi (grafico quindi uguale a quello di *frame_plots-b*).

  b) flusso al passare di 1/FPS secondi.

  c) andamento del flusso al passare di 1/FPS secondi.

Orari e giorni dei video utilizzati

Dopo aver elaborato questi strumenti per stimare il traffico navale abbiamo iniziato a processare i video scaricati. Con l'obiettivo di dare una statistica temporale, molto vaga, sull'andamento del traffico e per valutare la bontà dei nostri metodi, abbiamo scelto video registrati live in 2 fasce orarie (h 12:00 e h 16:30) di alcuni giorni della settimana; in tutto abbiamo scaricato 7 video da 5 minuti l'uno:

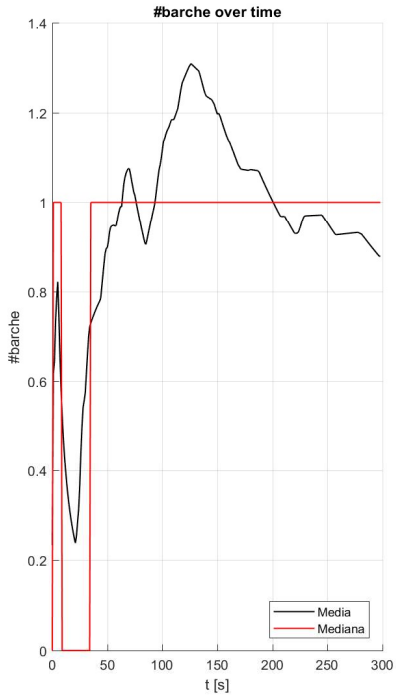
- DOMENICA 22/04/18 (h 12:00);
- MARTEDI' 01/05/18 (h 12:00 e h 16:30);
- GIOVEDI' 03/05/18 (h 12:00 e h 16:30);
- SABATO 05/05/18 (h 12:00 e h 16:30); *[Se si va a vedere il video registrato alle h. 12:00, dalla webcam si nota l'orario 10:40, evidentemente dovuto a un ritardo di trasmissione notevole di quel giorno]*.

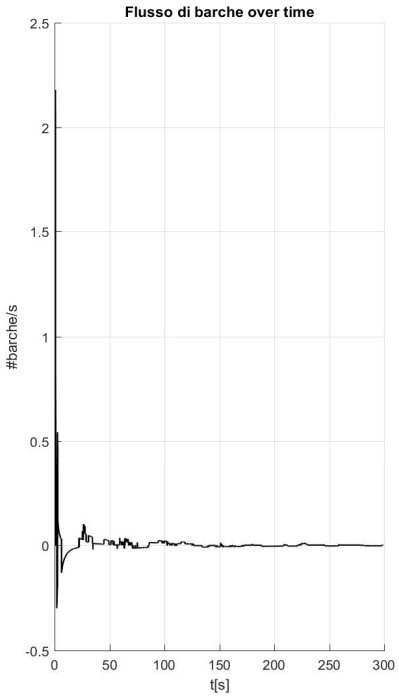
Si noti che il processing in C++ di ogni video di durata 5 minuti necessita di almeno 10 minuti, mentre poi la creazione dei grafici in Matlab avviene in pochi secondi; per questo abbiamo usato "solo" 8 video, pur consci che per ottenere una statistica ottimale servirebbero molti più video.

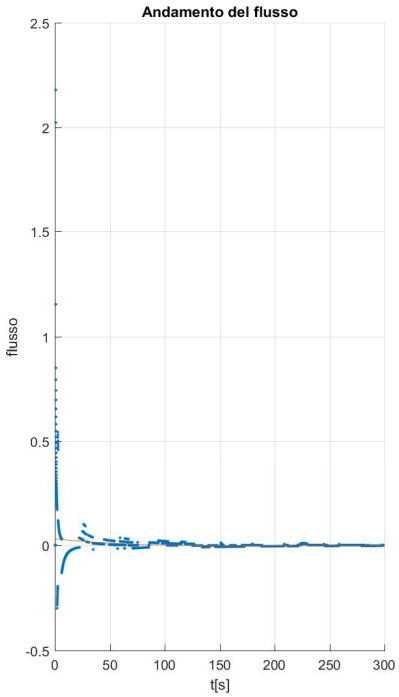
Commento sui grafici ottenuti

Per commentare i grafici che sono risultati dal nostro codice prendiamo in esempio il dato di **MARTEDI' 01/05/18** (omettiamo il grafico *frame_plots* perché meno interessante e perché il grafico *b* è uguale a quello *time_plots-a*, come già spiegato prima).

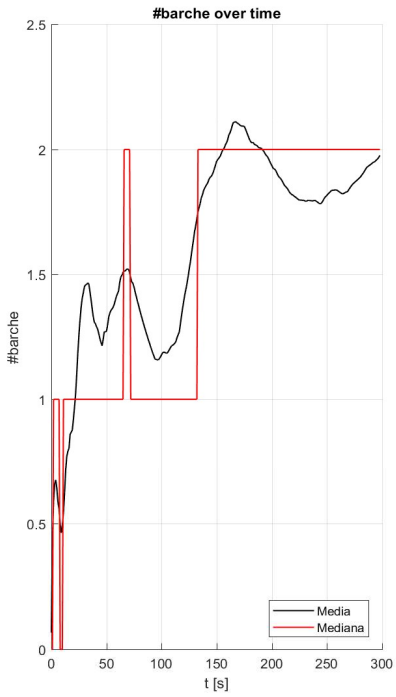
h 12:00

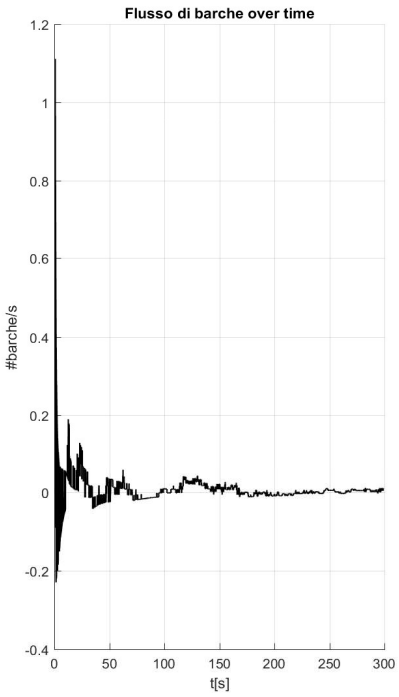


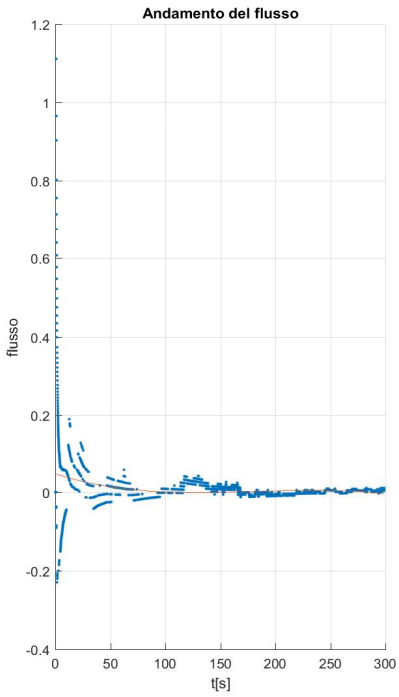




h 16:30





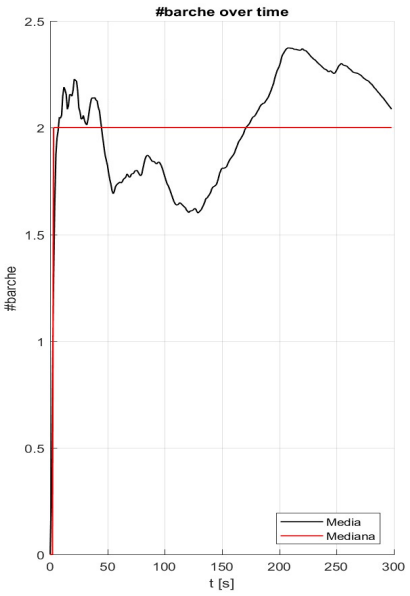


Di primo acchito sembra di poco più affollato l'orario delle 16:30: passati 5 minuti, la media (in **nero**) di barche contate risulta essere circa 1,98 e la mediana (in **rosso**) 2, a fronte dell'orario delle 12:00 in cui la media è di circa 0.88 barche e la mediana è 1. Ciò è verosimilmente in linea con la verità a terra se si guardano i video corrispondenti, seppur con i dovuti errori. Per quanto riguarda il flusso delle barche l'orario di 16:30 è molto più variabile e fluttuante rispetto a quello delle 12:00, come se ci siano dei momenti in cui le barche aumentano sensibilmente nella scena inquadrata e momenti successivi in cui le barche escono molto velocemente dalla scena. Per entrambi gli orari però, e anche per tutti gli altri video, il flusso si stabilizza sempre attorno allo zero non distanziandosi di tanto e dunque per discutere meglio di questo descrittore, ovvero la derivata della media del numero dei blob contati, servirebbero molti più dati e d'ora in poi considereremo significativi solo i dati su media integrale e mediana.

N. B.: A causa di un transitorio iniziale dell'algoritmo *adaptive background subtraction* in C++, nel quale viene allenato lo sfondo per 20 frame (`const int training_nr = 20;`), quando esportiamo il vettore del n° dei blob nel file.txt, esso risulta essere più corto di 20 elementi, portando poi a dei errori di compatibilità dei vettori nel *plotting*. Per risolvere ciò abbiamo aggiunto manualmente 20 elementi = 0 iniziali (cioè in cui il n° di barche contate è zero), che incidono relativamente poco nel calcolare media, mediana e flusso.

```
%% lettura file txt col numero di blob contati in ogni frame
filename = '05_01_12.00_MAR.txt';
delimiterIn = ' ';
blob_1 = importdata(filename,delimiterIn); %perde un tot di frame iniziali
dovuti all'adaptive bg sub

%% inizializzazione variabili
fps = 30; % frame per secondo video
T = 299; %[s] tempo totale, per video da 5 min
%T = 148; %[s] tempo totale, per video da 2.5 min
%T = 100; %[s] tempo totale, per video da 1.7 min
%T = 75; %[s] tempo totale, per video da 1.25 min

frame = (1:fps*T)'; %apice=trasposto --> vettore riga trasf a colonna

L = length(frame)-length(blob_1);
blob = zeros(fps*T,1);
for i = 1:length(blob_1);
    blob(L+i,:)=blob_1(i,:);
end
[…]
```

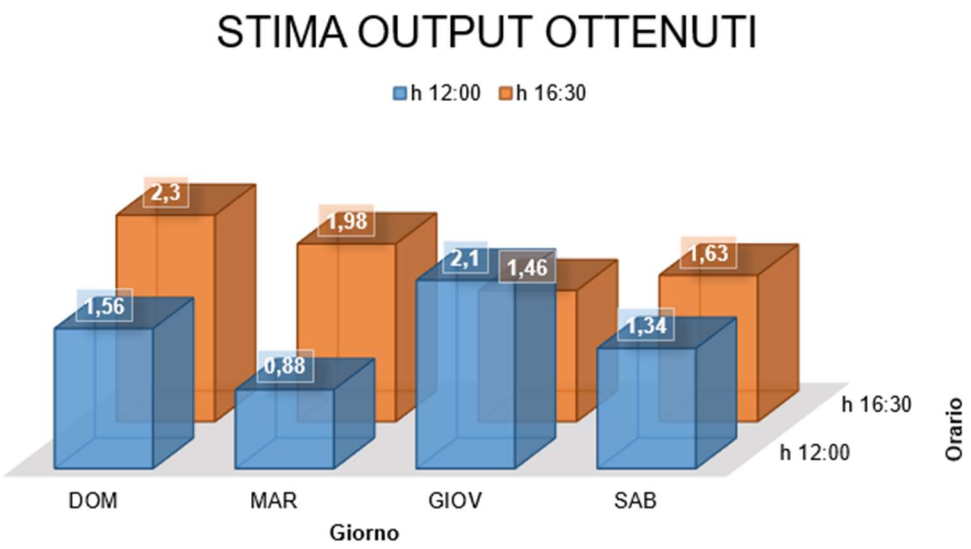
A lato portiamo in esempio il dato con la media integrale più alta, poco sopra il 2: GIOVEDI' 03/05/18, h 12:00. E' un dato di poco superiore a quello di MARTEDI' h 16:30 prima analizzato, e la mediana al termine dei 5 minuti è pari a 2 in entrambi. Dopo aver processato tutti i restanti video, di cui non parleremo nel dettaglio (tutti i grafici sono allegati), ci siamo accorti di come l'andamento di media e mediana sia abbastanza omogeno tra le varie giornate: il massimo raggiunto dalla mediana è quasi sempre 2, ad eccezione di DOMENICA 22/04/18, h 12:00, in cui si verifica per circa un minuto un picco a 3 nella parte iniziale del video ritornando poi a 2. La media invece varia in genere tra 1 e 2. Se ci limitassimo a queste considerazioni potrebbe sembrare che tra i vari giorni



analizzati il traffico sia pressappoco lo stesso e non particolarmente intenso. Il nostro algoritmo seppur robusto, è infatti sensibile alla visuale della camera, e presenta difficoltà a causa del movimento molto lento delle imbarcazioni, producendo quindi un output che può sembrare sottostimato rispetto alla *ground truth*. Discuteremo meglio di questi problemi in seguito.

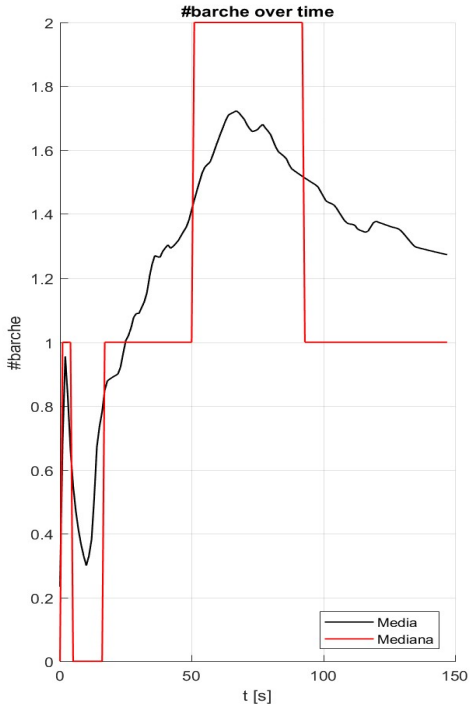
Idea sull'andamento settimanale

Di seguito, tenendo conto solo della media integrale al termine dei 5 minuti, mostriamo un andamento generale dei giorni analizzati, per avere un'idea sugli orari/giorni con più traffico navale. DOMENICA 22/04/18 alle h 16:30 non ci è stato possibile registrare un video, quindi abbiamo aggiunto noi un dato casuale (2.3), di poco più alto del *range* prima considerato, ma comunque pertinente in quanto abbiamo supposto che la domenica pomeriggio ci possa essere una circolazione di barche più elevata dovuta al turismo.



Velocità dei video

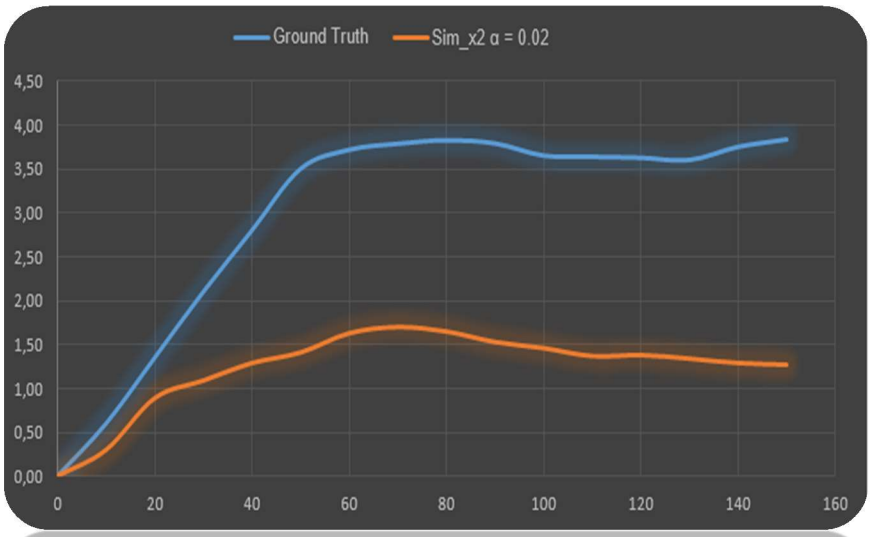
Facendo alcune considerazioni rispetto al processing dei video abbiamo notato che alcune barche, pur muovendosi, non venivano identificate come *blob* a causa dell'estrema lentezza del loro movimento. Per aggirare questo problema abbiamo optato per la velocizzazione 2x del video registrato, ottenendo un nuovo video di durata dimezzata (2½ minuti) con risultati decisamente migliori. A lato mostriamo come esempio il *time_plot* di MARTEDI' 01/05/18, h 12:00, dopo la velocizzazione. Se confrontiamo il risultato di questo giorno con quello mostrato prima del video a velocità normale, notiamo un incremento l'incremento della media da 0.88 a 1.27. Essendo risultati più realistici, consideriamo i video velocizzati come materiale più attendibile per il processing, anche perché il tempo di calcolo diminuisce significativamente. Da qui in poi i dati proposti saranno relativi a video velocizzati.



Ground Truth

Conseguentemente alle considerazioni precedenti e su suggerimento del Prof., abbiamo svolto un confronto tra i dati ottenuti e una ***ground truth* ogni 10 s**. La *ground truth* (o verità a terra) è un termine usato in vari campi scientifici per riferirsi alle informazioni fornite dall'osservazione diretta (cioè l'evidenza empirica) in contrapposizione alle informazioni fornite dall'inferenza, per determinare l'accuratezza di queste ultime. Per misurare la *ground truth* abbiamo contato manualmente il numero di barche in movimento presenti nella scena per una finestra temporale di 10 s. Abbiamo questa operazione in modo iterativo annotando il totale delle barche al passare dei 10 s e successivamente per ogni finestra temporale abbiamo eseguito la media. Di seguito mostriamo tabella e grafico del confronto tra la *ground truth* (in **blu**) e la simulazione della media del video di MARTEDI' 01/05/18, h 12:00, velocizzato 2x (in **arancione**).

Tempo	#Barche	Ground Truth	Sim2x α=0.02
0	0	0	0
10	6	0.60	0.30
20	27	1.35	0.89
30	63	2.10	1.09
40	112	2.80	1.29
50	175	3.50	1.41
60	223	3.72	1.63
70	265	3.79	1.70
80	306	3.83	1.65
90	341	3.76	1.53
100	365	3.65	1.46
110	400	3.64	1.37
120	435	3.63	1.38
130	468	3.60	1.34
140	525	3.75	1.29
150	575	3.83	1.27



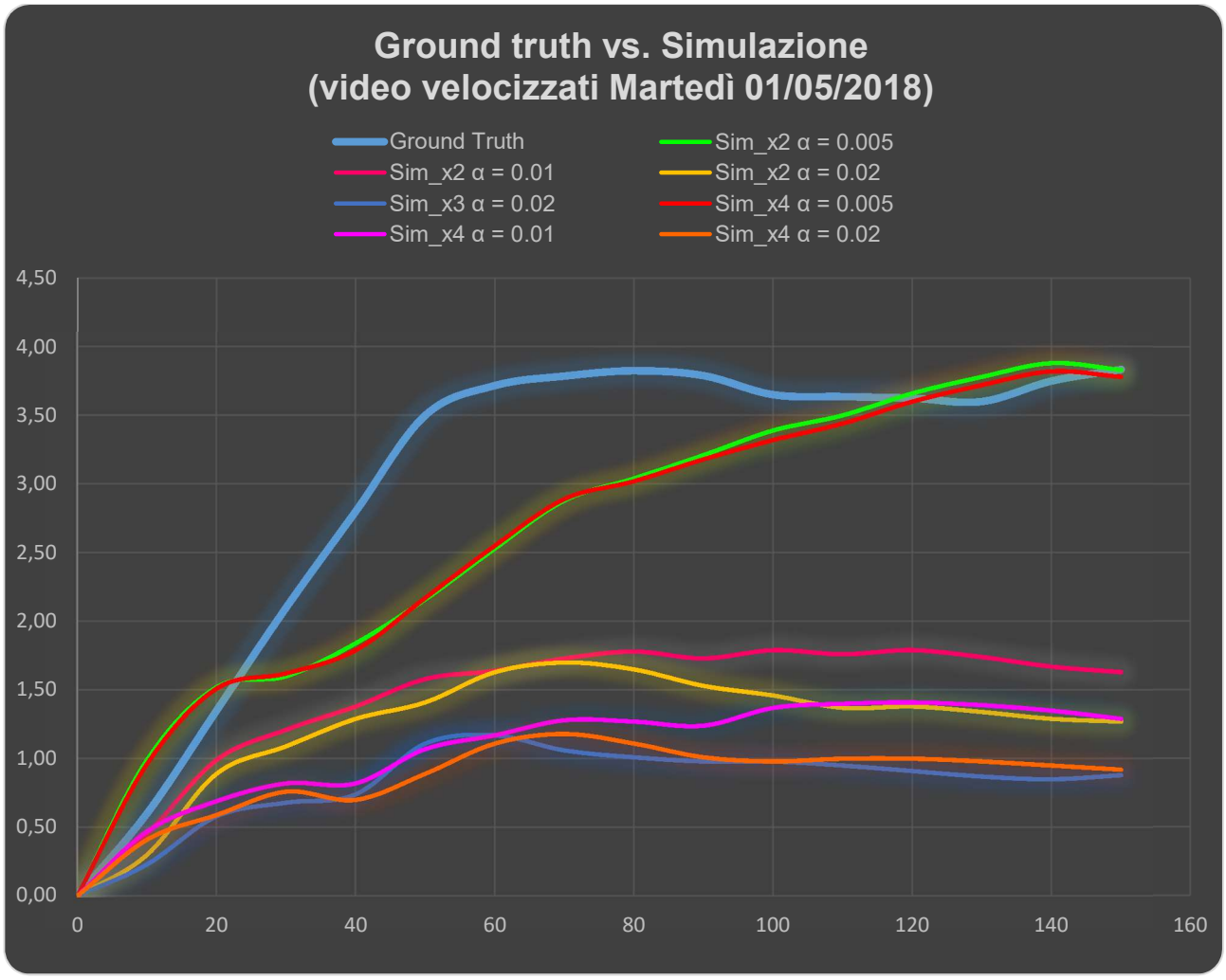
Analizzando il grafico vediamo che la media relativa alla simulazione 2x, sebbene sia un dato migliore del video non velocizzato, differisce dalla *ground truth* di circa "2½ barca"; per il contesto applicativo il risultato no è abbastanza buono. Infatti, analizzando da un punto di vista critico, questo si traduce in un errore percentuale di circa il 67% ($\frac{ground\ truth - simulazione}{ground\ truth} \bullet 100$), e abbiamo quindi ragionato su altri metodi da testare.

Velocità del video e *learning rate* α

In seguito a un incontro con il Prof., in cui gli abbiamo mostrato i nostri ragionamenti riguardo alla velocità del video, ci è stato consigliato di provare a interagire con il parametro *learning rate* α dell'algoritmo di *adaptive background subtraction* e/o con la velocità del video. Tutti i grafici prodotti finora avevano un α = 0.02 (come già detto nel paragrafo sull'algoritmo di *subtraction*) perché ci sembrava adatto alle nostre esigenze. Questo *learning rate* è però particolarmente adatto per il *motion detection* di pedoni, che si muovono circa a 5 km/h. Partendo dal video velocizzato 2x abbiamo dimezzato α portandolo a 0.01 in modo che lo sfondo si aggiornasse più lentamente, proprio per adattarsi alla lentezza delle imbarcazioni; successivamente abbiamo abbassato ulteriormente α a 0.005. Infine abbiamo fatto delle prove con video a velocità 3x (α=0.02) e a velocità 4x (α=0.02, 0.01,

0.005). Mostriamo quindi di seguito la tabella e il grafico di confronto tra la *ground truth* e le simulazioni con varie velocità e vari α (vedere la legenda).

Tempo	#Barche	Ground Truth	Sim2x α=0.005	Sim2x α=0.01	Sim2x α=0.02	Sim3x α=0.02	Sim4x α=0.005	Sim4x α=0.01	Sim4x α=0.02
0	0	0	0	0	0	0	0	0	0
10	6	0.60	0.99	0.45	0.30	0.23	0.96	0.47	0.41
20	27	1.35	1.52	0.99	0.89	0.58	1.51	0.69	0.59
30	63	2.10	1.60	1.21	1.09	0.68	1.62	0.82	0.76
40	112	2.80	1.84	1.38	1.29	0.74	1.79	0.82	0.70
50	175	3.50	2.16	1.58	1.41	1.11	2.17	1.07	0.89
60	223	3.72	2.53	1.64	1.63	1.17	2.55	1.17	1.11
70	265	3.79	2.88	1.73	1.70	1.06	2.89	1.28	1.18
80	306	3.83	3.04	1.78	1.65	1.01	3.02	1.27	1.11
90	341	3.76	3.21	1.73	1.53	0.98	3.18	1.24	1.01
100	365	3.65	3.39	1.79	1.46	0.98	3.32	1.37	0.98
110	400	3.64	3.50	1.76	1.37	0.95	3.44	1.40	1.00
120	435	3.63	3.66	1.79	1.38	0.91	3.60	1.41	1.00
130	468	3.60	3.78	1.74	1.34	0.87	3.72	1.39	0.98
140	525	3.75	3.88	1.67	1.29	0.85	3.82	1.35	0.95
150	575	3.83	3.83	1.63	1.27	0.88	3.78	1.29	0.92



Osservando il grafico notiamo che il parametro che influenza maggiormente la somiglianza tra la media integrale e la *ground truth* è il *learning rate* α, piuttosto che la velocità del video. Ad esempio i video velocizzati 3x e 4x con $\alpha = 0.02$ sono quelli che si comportano peggio, arrivando ad una media finale tra $0.88 \div 0.92$, molto distante dalla realtà a terra 3.83. Se consideriamo invece il video velocizzato 4x diminuendo α a 0.01 otteniamo un leggero miglioramento, che si appaia al risultato del video 2x, $\alpha = 0.02$. Se invece torniamo ad usare un video velocizzato 2x dimezzando α a 0.01 otteniamo un ulteriore progresso, portando la media a 1.63, che è però molto lontana dalla *ground truth* sia per andamento che per valore finale.

Un'ulteriore prova è stata fatta portando α a 0.005 e i risultati emersi sono stati decisamente migliori dal punto di vista del valore finale. Infatti sia il video velocizzato 2x che 4x con $\alpha = 0.005$ arrivano alla fine del video con un valore finale della media circa uguale a quello della *ground truth,* 3.83 (quello velocizzato 2x la raggiunge perfettamente). L'andamento delle ultime 2 simulazioni è circa uguale fra loro, ma non combacia con quello della *ground truth* presentando una velocità minore; questo potrebbe essere dovuto all'onere computazionale e alla tipica fase iniziale di aggiornamento.

In sintesi a parità di velocità del video, la bontà del risultato è inversamente proporzionali al valore di α. L'unico problema riscontrato, un po' tedioso, è la necessità di modificare il codice, diminuendo le dimensioni dell'elemento strutturale dei filtri morfologici all'aumentare della velocità del video. Cioè avviene in quanto lo spostamento è più significativo e l'effetto dei filtri morfologici non modificati risulterebbe più pesante. E' interessante considerare che l'utilità dell'aumentare la velocità dei video corrisponde ad una possibile simulazione dell'acquisizione in un contesto reale, ovvero campionare ad una frequenza minore per risparmiare risorse.

Conclusione e miglioramenti futuri

In conclusione possiamo affermare che, per le condizioni di lavoro e per il tipo di materiale a disposizione, il codice risulta abbastanza efficace per risolvere il quesito posto all'inizio: la stima del traffico navale a Venezia. Lavorando su alcuni parametri come velocità del video e *learning rate* α dell'algoritmo *abgs* abbiamo infatti ottenuto dei risultati soddisfacenti e non troppo distanti dalla realtà, inoltre lavorare con video accelerati diminuisce i tempi di calcolo: sta all'utente trovare il giusto *trade-off*. Per quanto riguarda i limiti, oltre al rumore dovuto alla posizione non ottimale della telecamera, al leggero movimento delle onde e alla lentezza delle barche, si può notare che velocizzando troppo il video senza diminuire il *learning rate* la media risulta troppo sottostimata rispetto alla *ground truth*. Oltretutto il codice è stato pensato per adattarsi al meglio al tipo di video analizzato e quindi può presentare dei problemi di *over-fitting*. La sfida futura potrebbe essere quella di migliorare questo lavoro con vari accorgimenti, alcuni dei quali per ora ipotizzati da noi solo a livello teorico (senza verificarne la fattibilità):

1) Trovare una posizione ottimale per le telecamere che riprendono il canale;
2) Rendere il codice più versatile e flessibile per varie situazioni;
3) Corredarlo di un classificatore che identifichi il tipo di imbarcazioni (e quindi la dimensione) al fine di fornire un parametro di regolarizzazione del traffico (ad esempio se c'è un maggior numero di barche grandi questo parametro aumenta);
4) Implementare un funzionamento *real-time* del software (o comunque di poco in ritardo) in corrispondenza di telecamere in punti nevralgici dei canali per comunicare i dati elaborati alle imbarcazioni che stanno arrivando in quel determinato punto: questa sarebbe sicuramente una delle operazioni più difficili, sia in termini di complessità di calcolo (servirebbero macchine potenti dotate di GPU) che di risorse di rete.

Bibliografia:

- Slides Teoria e Laboratorio “Elaborazione e Trasmissione di Immagini”, Prof. Francesco De Natale e Dott. Andrea Rosani
- https://docs.opencv.org/2.4/
- Slides “Lab 2: Motion Detection”, Niccolò Bisagno, Computer Vision & Multimedia Analysis Course - A.A. 2017/2018
- https://stayrelevant.globant.com/en/peopletracker-people-and-object-tracking/
- https://www.learnopencv.com/blob-detection-using-opencv-python-c/
- https://github.com/andrewssobral/simple_vehicle_counting
- https://www.youtube.com/watch?v=z1Cvn3_4yGo
- MATLAB help