Relazione progetto “Exposure Fusion per Effetti HDR”

Patrizio Renelli

Il progetto “Exposure Fusion per Effetti HDR” chiedeva di implementare una versione dell’algoritmo presentato nell’articolo “Exposure Fusion di Tom Mertens Jan Kautz Frank Van Reeth” che descrive una delle possibili implementazioni dell’HDR basandosi su mappe di peso generate da varie immagini di input, usate per pesarne i pixel, in base al contrasto, la saturazione e l’esposizione, infine sovrapporrà e fonderà le immagini fornite in un’unica immagini finale.

Implementazione

Per la realizzazione del progetto si è deciso di realizzare in python la classe “HDRElaborator” che implementa tutte le funzioni necessarie per l’elaborazione, sfruttando le librerie cv2 ed numpy.
Le funzioni implementate sono:

- loadImage: utilizzata per effettuare il caricamento e il resize delle immagini prese in input,
- align: utilizzata per allenare le immagini caricate,
- lowPassFilter: utilizzata per effettuare il resize ed applicare un filtro gaussiano alle immagini,
- gaussPyramid: utilizzata per calcolare le piramidi gaussiane,
- laplPyramid: utilizzata per calcolare le piramidi laplaciane,
- findImportance: utilizzata per calcolare i pesi delle immagini basandosi sul contrasto, saturazione ed esposizione,
- overlap: utilizzata per calcolare la fusione delle immagini basandosi sulle mappe pesate,
- getHDRImage: utilizzata per richiamare le funzioni sopra citate ed ottenere l’immagine elaborata

Inizialmente richiamando il costruttore della classe HDRElaborator, a cui passiamo un array contenente i nomi delle immagini di input e la profondità delle piramidi che andremo successivamente a creare, dopo aver creato un oggetto della classe appena citata, andremo a richiamare il suo metodo getHDRImage che richiamerà i vari metodi necessari per la creazione dell’immagine HDR.
Si richiamerà la funzione loadImage, che dopo aver caricato i vari file delle immagini prese in input controllerà che almeno 2 file siano state caricate, con un’immagine sola l’esecuzione dell’algoritmo sarà inutile, in caso contrario la funzione ritornerà, successivamente si calcola la dimensione media delle foto, si verifica se questa dimensione calcolata sia compatibile con le future resize che sarà necessario compiere nel momento che si calcolano le varie piramidi gaussiane e laplaciane e la fusione finale per ottenere l’immagine finale. Per compatibilità si intende che le immagini dovranno avere dimensioni il cui risultato del modulo per 2^profondità imposta all’algoritmo sia pari a 0, quindi andremo a trovare questa nuova

dimensione compatibile con le nostre esigenze ed effettuiamo un resize di tutte le immagini passate, andando ad usare usare un interpolazione differente nel caso si stia andando a ridurre o aumentare la dimensione, nel primo caso useremo un interpolazione, sempre fornita da cv2, chiamata INTER_CUBIC, mentre nel secondo caso useremo la INTER_AREA. Se quindi abbiamo caricato almeno 2 file, procediamo ad allineare le immagini, sfruttando la funzione process della classe AlignMTB che effettua un "allineamento soft" riuscendo ad annullare gli sfasamenti più piccoli. Una volta allineate le foto, richiamando la funzione overlap andremo ad eseguire l'algoritmo vero e proprio. Iniziamo invocando la funzione findImportance che dopo aver costruito una matrice "singleImportance" della dimensione delle foto totalmente inizializzata a 0, per ogni immagine andremo a portare i valori di tutti i pixel in un range [0,1] e costruiremo una matrice "weight" uguale alla matrice definita poco fa.

Per calcolare come richiesto nel paper i valori di contrasto, saturazione ed esposizione procediamo nel seguente modo.

Per il **contrasto** convertiamo la foto analizzata in scala di grigi attraverso la funzione cv2.cvtColor, passandogli come conversione richiesta "COLOR_BGR2GRAY", prendiamo il valore assoluto dell'immagine a cui applichiamo un filtro laplaciano, con una profondità finale impostata a "CV_32F". Il filtro laplaciano è un filtro che calcola le derivate seconde di un immagine, andando così a misurare con quale velocità cambiano le intensità, solitamente viene utilizzata per individuare i bordi e le texture presenti nell'immagine, aggiungiamo infine alla matrice "weight" la matrice relativa ai valori del contrasto.

Vogliamo andare a selezionare solo i colori più saturi, poiché un'eccessiva esposizione porta ad avere dei colori desaturati, che non vogliamo includere nella nostra immagine finale. Per calcolare il peso relativo alla **saturazione** andremo a calcolare la derivata dei vari pixel e aggiungiamo il valore della matrice pesata della saturazione alla matrice "weight". Nel calcolare un'esposizione ottimale vogliamo andare a mantenere tutti quei valori che in un intervallo tra 0 (che rappresenta un'immagine sottoesposta) e 1 (che rappresenta un'immagine sovraesposta), si aggirano intorno allo 0.5. Per calcolare il peso dell'**esposizione** dei singoli pixel andremo a sfruttare la funzione indicata nel paper, che traduciamo in python come "exp(-(([matrice dell'immagine] - 0.5)**2)/(2*(0.2**2)))", andando a calcolare il prodotto delle matrici ottenute per avere una matrice unica da poter sommare a "weight". Quando abbiamo calcolato per ogni immagine i suoi pesi, dividiamo le mappe pesate con l'importanza sommata delle varie mappe e normalizziamo moltiplicando per 255.

Andiamo ora a calcolare le piramide laplaniche e gaussiane come richiesto.

Per le prime partiamo calcolando le piramidi gaussiane delle singole immagini, dopo di che ad ogni livello di queste piramidi gaussiane andiamo ad applicare la funzione lowPassFilter, passandogli come parametro di exp False.

Prima di parlare dell'implementazione di questa funzione definiamo un kernel gaussiano come una matrice 2d e viene generalmente usato per realizzare un filtro passa basso dove i valori sono derivati dalla funzione "cv2.getGaussianKernel" che ci consente di ottenere i

coefficienti del filtro gaussiano, nel nostro caso useremo un'apertura di 5 e una deviazione standard gaussiana calcolata di default.
La funzione lowPassFilter accetta in input l'immagine da elaborare e un boolean exp che ne determina il funzionamento:

- se exp = True applicheremo la funzione "cv2.filter2D" che applicherà il filtro gaussiano all'immagine di input e ridurremo la dimensione dell'immagine attraverso "cv2.resize" ad un quarto della sua dimensione originale poiché usiamo dei coefficienti fx ed fy di valore 0.5, dimezzando cos' sia la dimensione dell'asse x che dell'asse y,
- se exp = False applicheremo la funzione "cv2.filter2D" come sopra e poi la funzione "cv2.resize" questa volta con coefficienti fx ed fy di valore 2, in modo da riportare l'immagine alla dimensione originale

Dopo di che calcoliamo la differenza tra l'ultima immagine scalata della piramide gaussiana e l'immagine appena elaborata e l'aggiungiamo alla lista che ritorneremo.
Per calcolare le piramidi gaussiane andiamo ad applicare la funzione lowPassFilter, questa volta con exp True in modo da applicare il filtro gaussiano e ridurre la dimensione in output.
A questo punto per ogni livello andremo a costruire la matrice "moreDetails" inizializzata tutta a 0, e per ogni immagine andremo prima a normalizzare le piramidi gaussiani, portando tutti i loro valori nel range [0,1], andiamo a impilare le gaussiane e moltiplichiamo le gaussiane con le immagini laplaniche, a questo punto sommo il risultato alle elaborazioni precedenti, in questo modo ad ogni iterazione andrò a catturare sempre più dettagli, salvando il tutto nella matrice "moreDetails". Alla fine aggiungiamo le immagini generate da ogni iterazione del ciclo for interno per i singoli livelli su tutte le immagini all'array pyr. Infine andiamo a sommare a fused che rappresenta l'ultima immagine presente in pyr, tutte le matrici sopra elaborate, applicando ad ogni iterazione la funzione lowPassFilter per applicare il filtro gaussiano in modo da ridurre ulteriormente "l'effetto cucitura" citato anche nel paper e ridimensionare correttamente l'immagine per la somma, l'immagine finale è ora pronta per essere ritornata e con la funzione cv2.imwrite salvata.

Il codice da me elaborato è stato sviluppato partendo da diversi articoli e implementazioni già presenti online, raccogliendo da ciascuno maggiori informazioni possibili sull'utilizzo di funzioni utili presenti nelle due librerie utilizzate nell'implementazione ed esempi di implementazioni e risultati finali utili per accertarmi della bontà delle immagini elaborate dal mio algoritmo

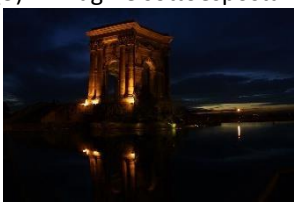
Esempi di esecuzione

Le 3 immagini di partenza usate come esempio hanno una dimensione di 1200x800

(1) Immagine sovraesposta

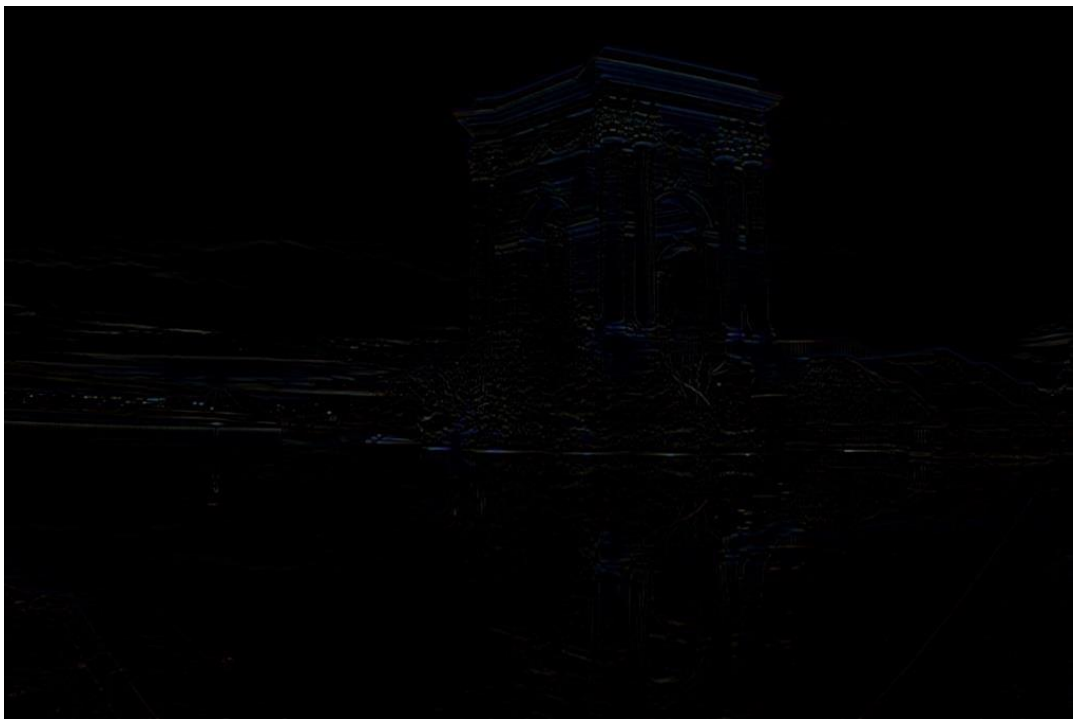
(2) Immagine con esposizione intermedia

(3) Immagine sottoesposta

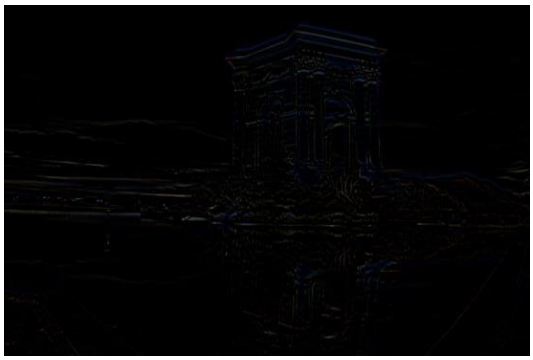


Piramidi laplaniche

Calcoliamo le piramidi laplaniche delle diverse foto usando 4 come numero di livelli (1)



La prima immagine che formerà la piramide laplaciana avrà dimensione 1200x800



La seconda immagine che formerà la piramide laplaciana avrà dimensione 600x400

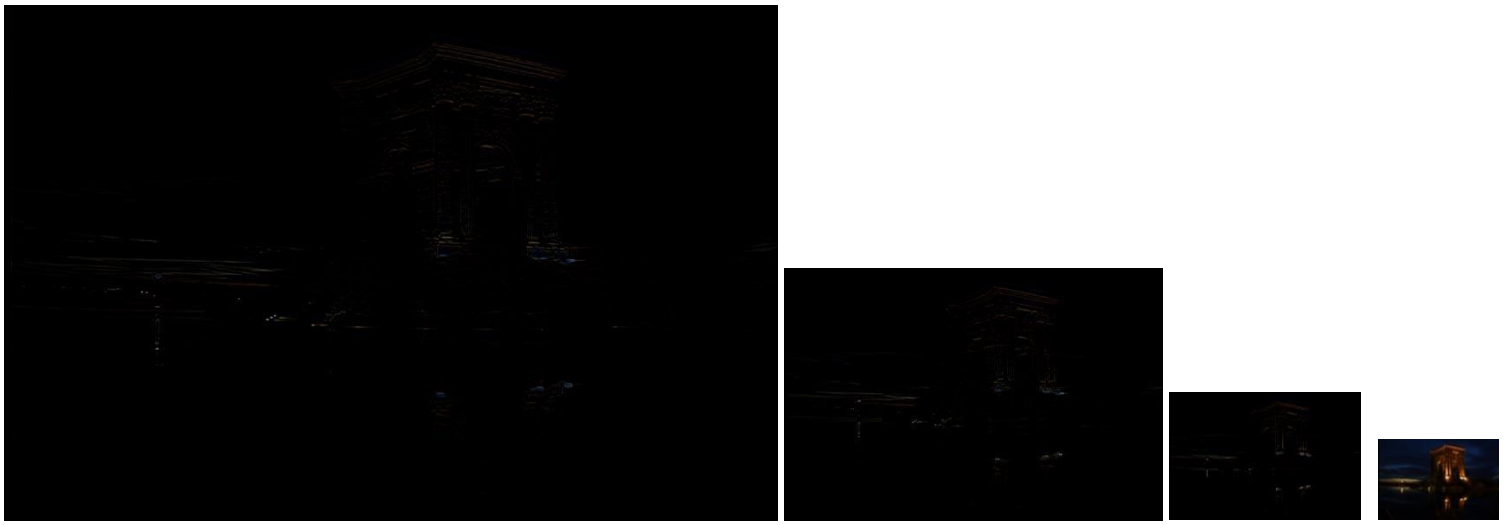
La terza immagine che formerà la piramide laplaciana avrà 300x200

La quarta immagine che formerà la piramide laplaciana avrà 150x100

(2)

1200x800 600x400 300x200 150x100

(3)



1200x800 600x400 300x200 150x100

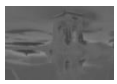
Dopo aver calcolato la piramide laplanica relativa ad ogni foto andiamo a calcolare le varie piramidi gaussiane.

Piramidi gaussiane

Calcoliamo le piramidi gaussiane delle diverse foto usando 4 come numero di livelli

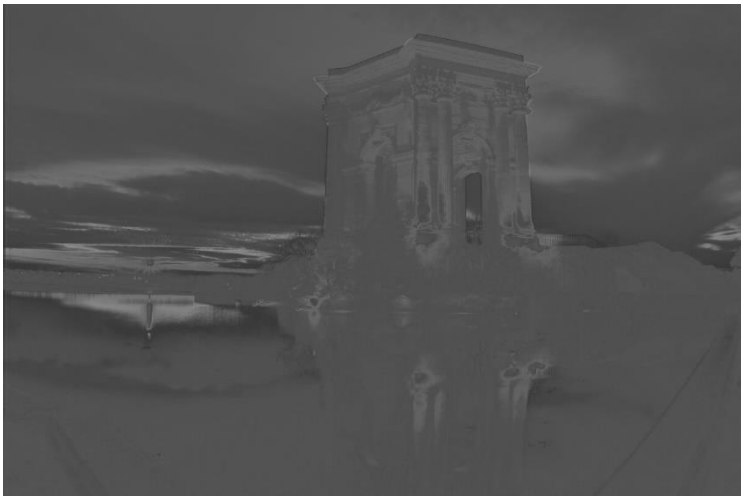
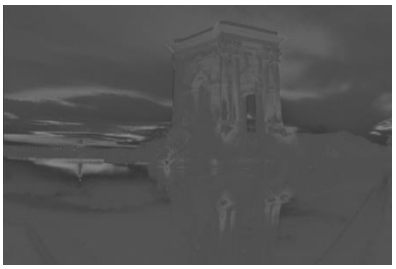
(1)

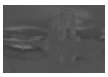


1200x800 600x400 300x200 150x100

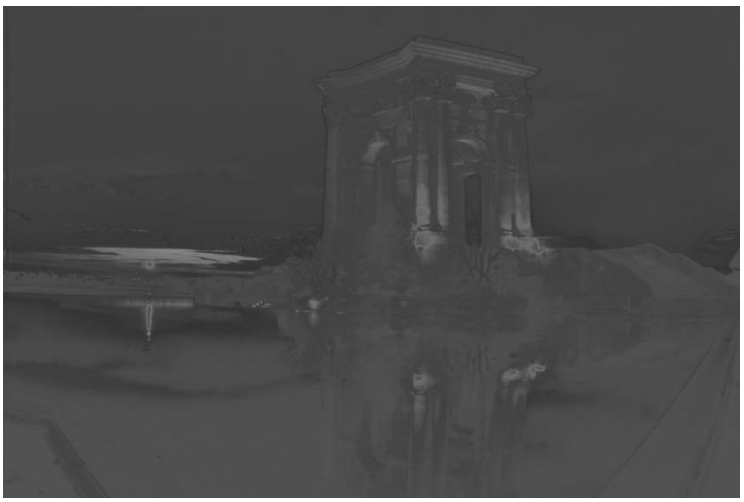
(2)



1200x800 600x400 300x200 150x100

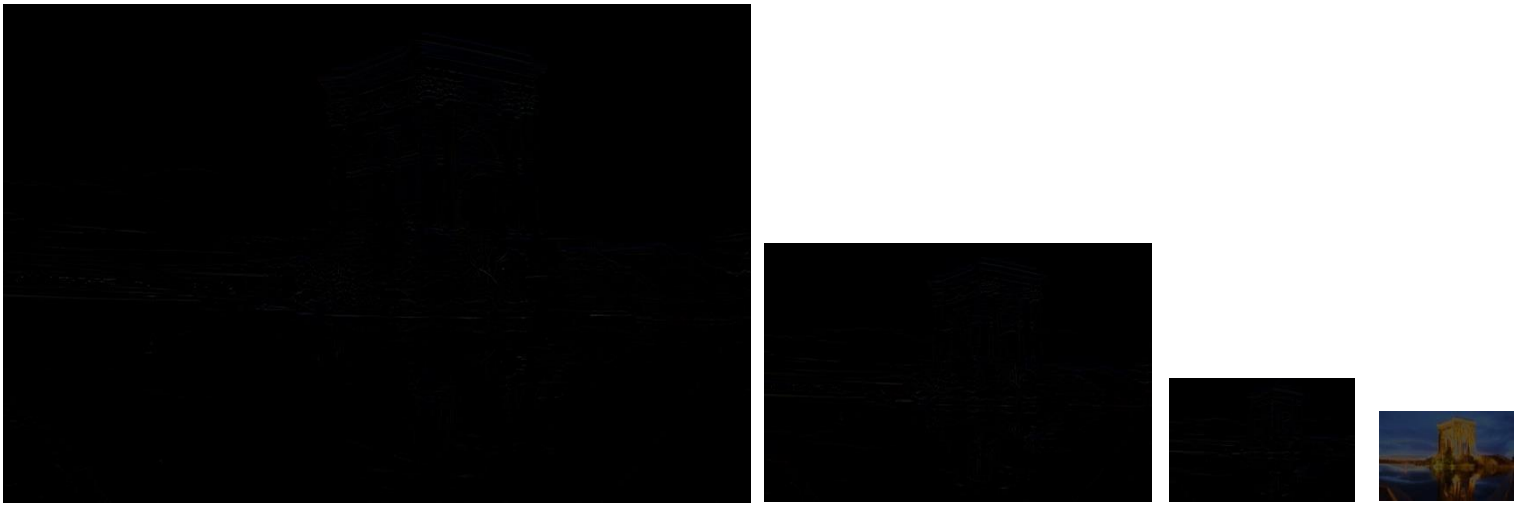
(3)




1200x800 600x400 300x200 150x100

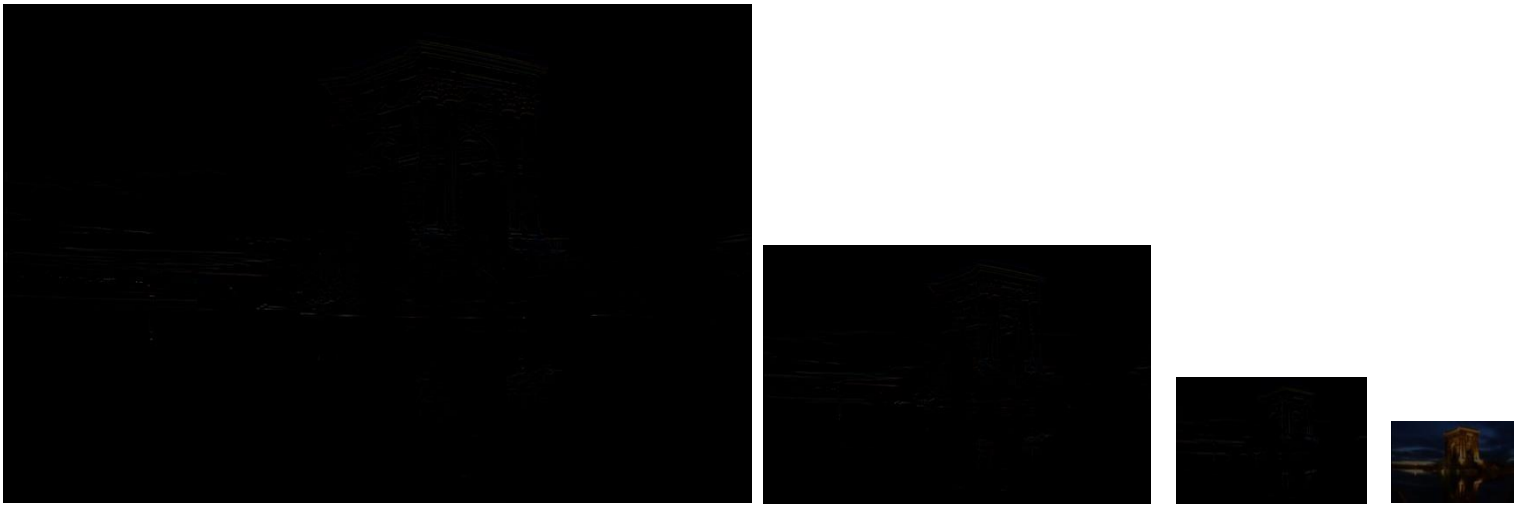
Otteniamo ora le immagini risultato della fusione dei singoli livelli delle piramidi gaussiane e laplaniche, attenute attraverso la funzione “cv2.multiply”.

(1)



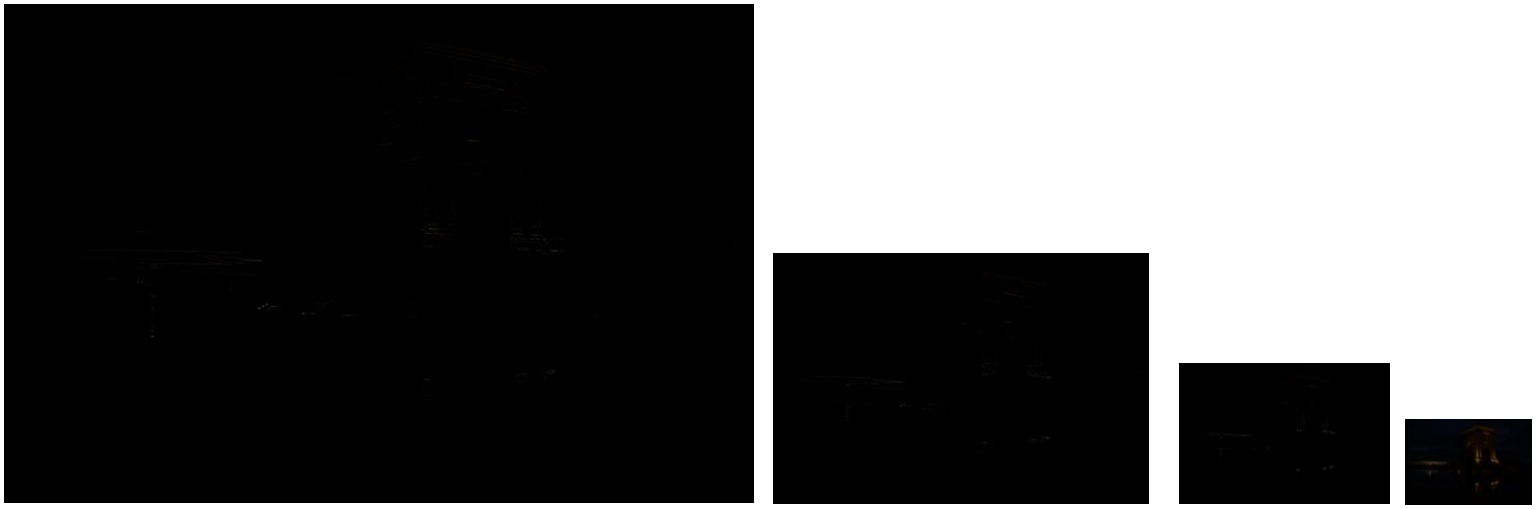
1200x800 600x400 300x200 150x100

(2)



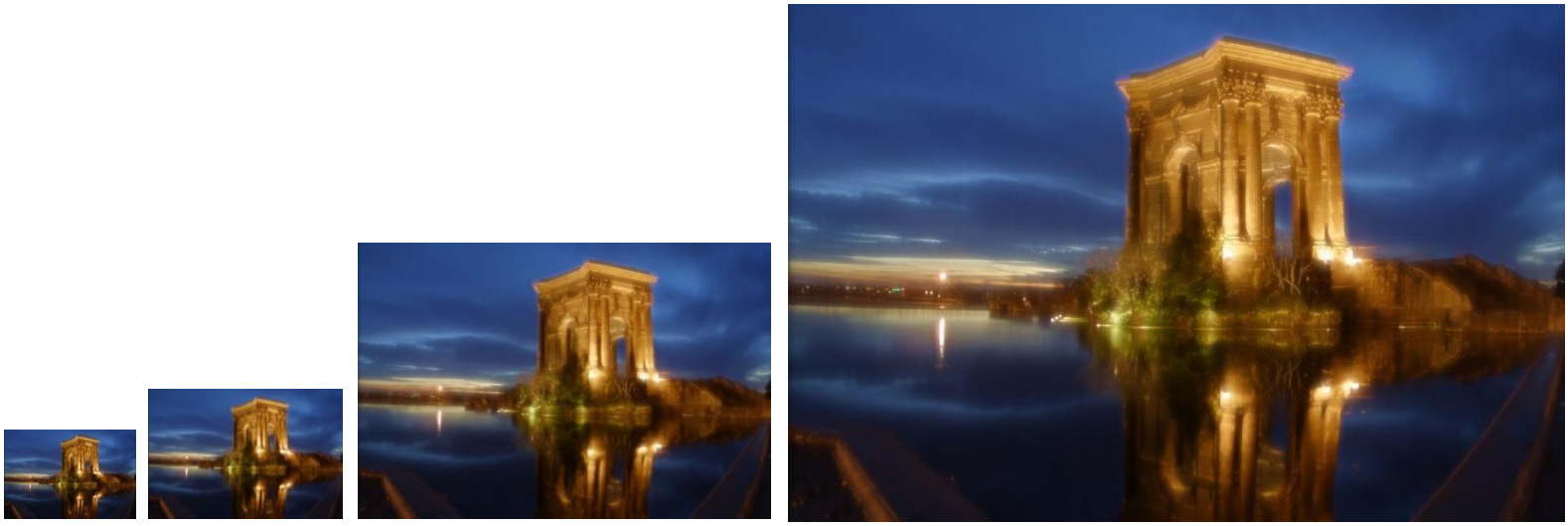
1200x800 600x400 300x200 150x100

(3)



1200x800 600x400 300x200 150x100

Sommiamo ora le varie piramidi ottenute per ottenere l'immagine finale, prima di sommarle gli applichiamo la funzione "lowPassFilter" per ridimensionarle correttamente

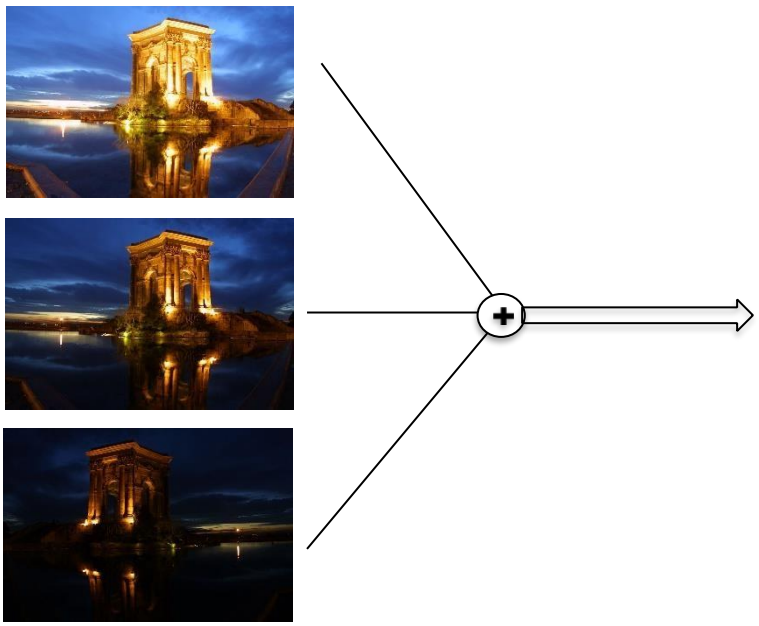


150x100 300x200 600x400 1200x800

L'immagine ottenuta e quindi ritornata dall'algoritmo sarà proprio l'ultima di risoluzione originale 1200x800.

Risultati finale

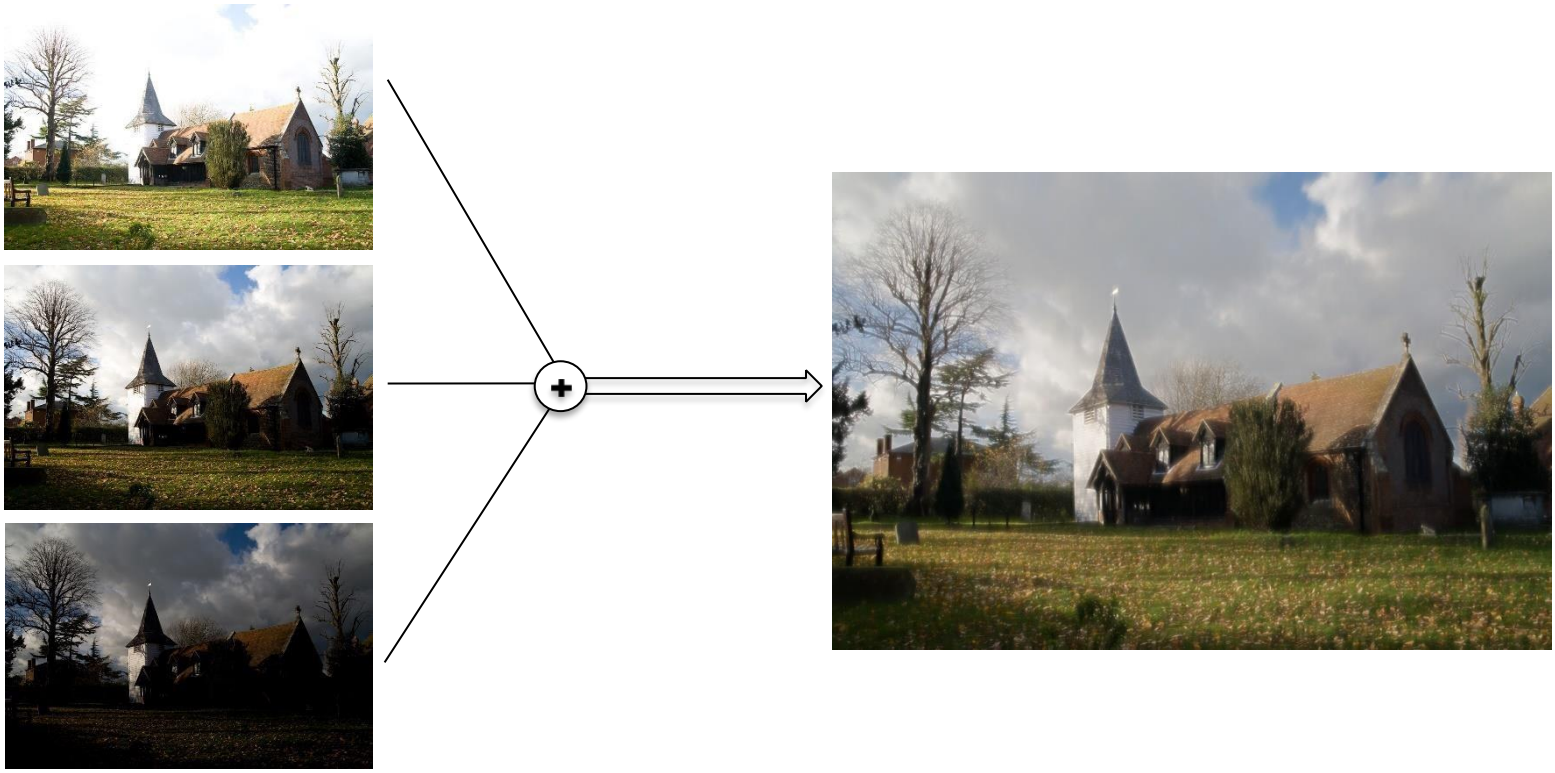
Monument



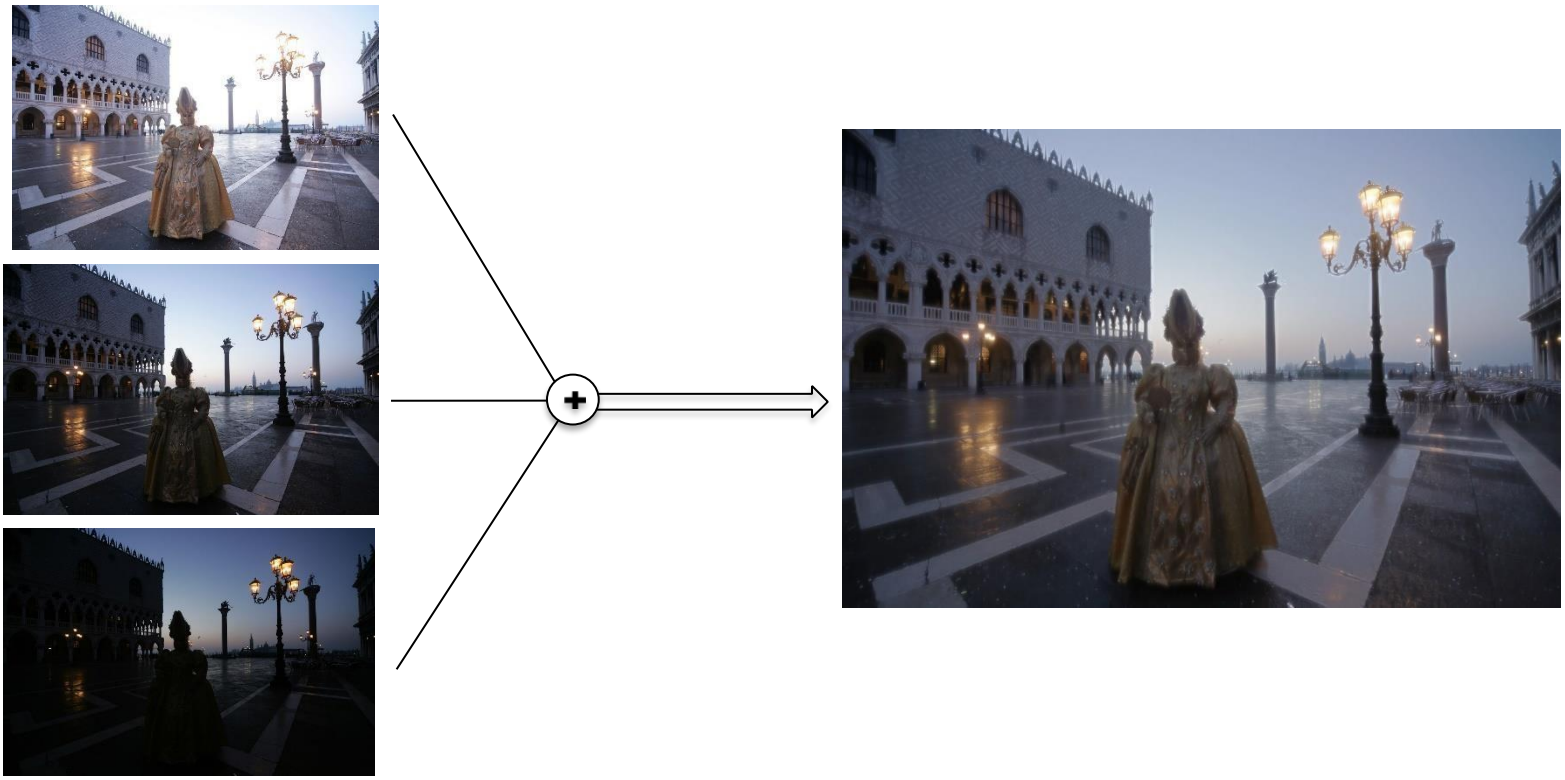
Utilizzando come numero livelli 4

Subito sotto troviamo altri esempi di risultati finali partendo da 3 immagini di input

Home



Venice



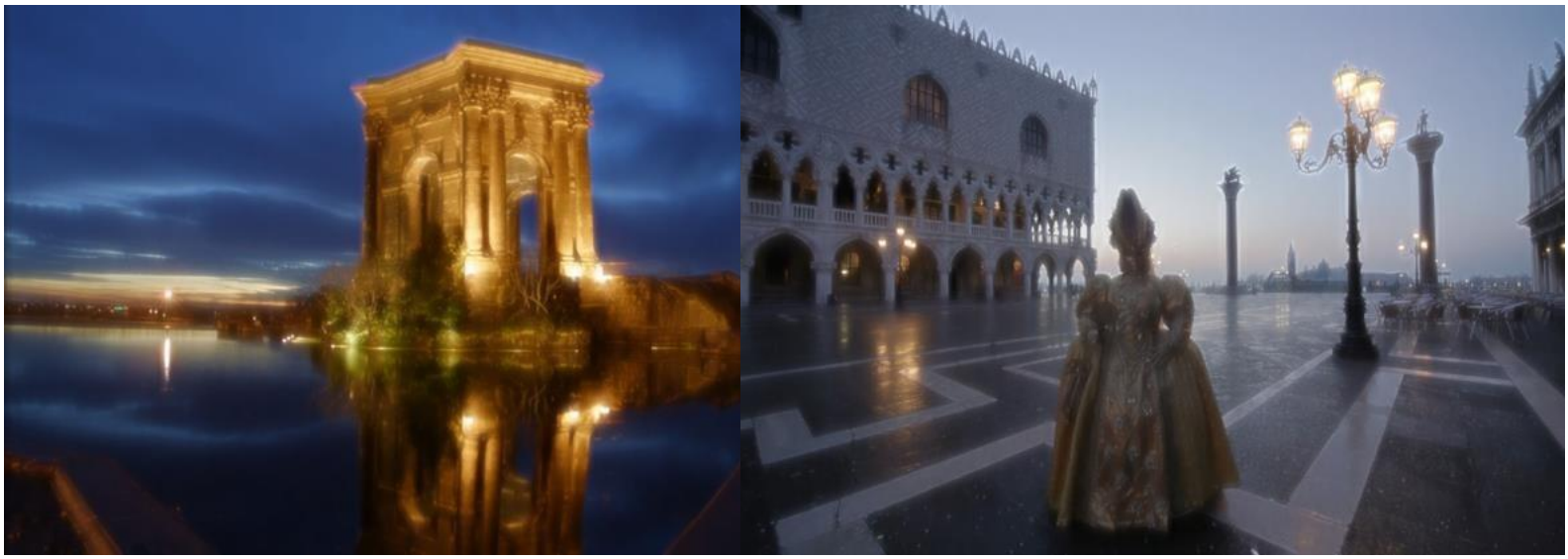
Per visualizzare correttamente i risultati finali eseguiti su varie immagini di test seguire il documento Test nella cartella Immagini che spiega dove trovare le immagini generate dalle varie fasi di esecuzione e come eseguire i test dell'algoritmo.

Considerazioni finali sui risultati

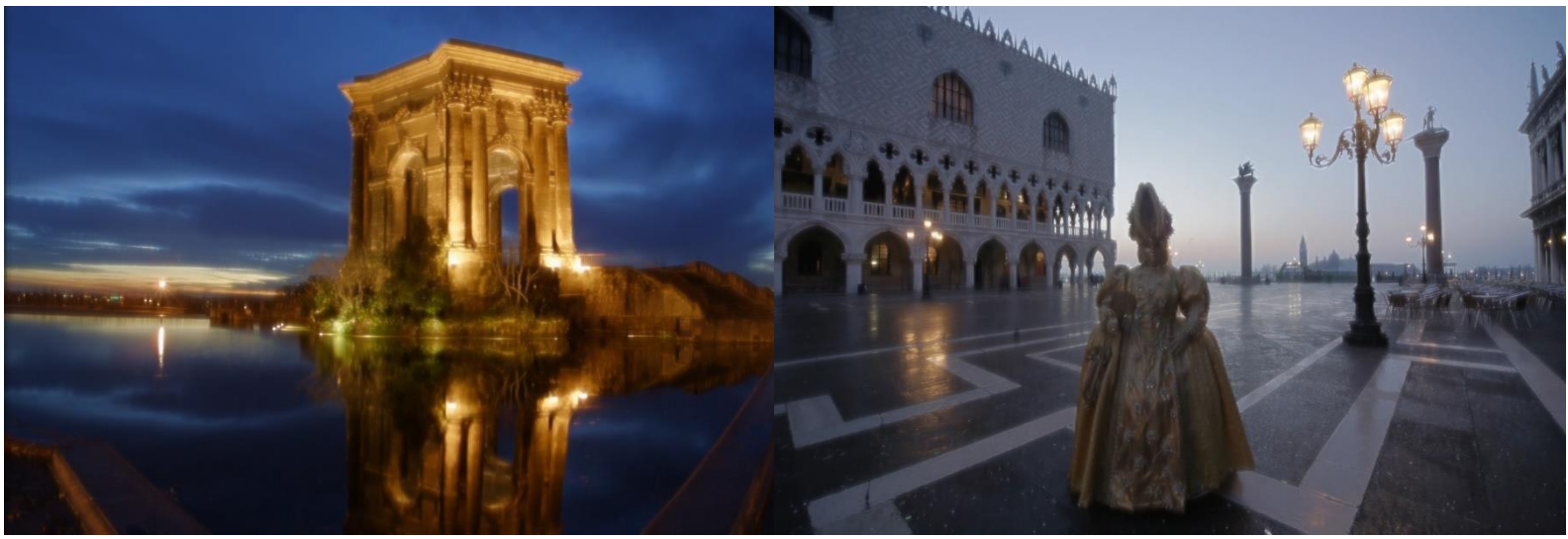
Possiamo notare come il risultato finale sia soddisfacente, ma abbiamo un particolare effetto "sfocatura", causato dalle diverse resize e le varie sovrapposizioni effettuate durante l'esecuzione, che aumenta con l'aumentare del numero di livelli in cui le immagini originali vengono scomposte.

Vediamo ora esempi di esecuzione con un numero di livelli differente notando il cambio di qualità.

Numero di livelli = 4



Numero di livelli = 3



Numero di livelli = 2