Universitàdegli Studi di Brescia
Facoltàdi Ingegneria elettronica
Dipartimento di Elettronica per l'Automazione
Corso di Robotica
A.A. 1999/2000

ELABORATO

LA SEGMENTAZIONE DI IMMAGINI PER IL CONTEGGIO DELLE MELE

Riva Walter Matr. 025020

INTRODUZIONE

Il presente elaborato si pone come obbiettivo l'analisi della segmentazione di immagine applicata ad un problema pratico: come contare in modo automatico un certo numero di elementi posti su di una superficie piana?

Gli oggetti dei quali si vuole contare il numero sono mele poste in una cassetta. Al fine di meglio comprendere tutte le problematiche relative all'applicazione si è scelto di effettuare anche la segmentazione di oggetti con forma e caratteristiche cromatiche più regolari rispetto a quella delle mele: palline da ping-pong e uova. Le palline da ping-pong sono state semplicemente appoggiate ad una superficie piana, mentre le uova sono state fotografate nel classico contenitore da 6 in materiale plastico.

COSA E' STATO FATTO

Per analizzare le differenti situazioni possibili sono state scattate fotografie digitali di mele gialle su sfondo bianco, mele gialle su sfondo nero, mele rosse su sfondo bianco e mele rosse su sfondo nero.
Sono state utilizzate cassette di cartone di mele Melinda (gialle) e Marlene(rosse).
Inoltre, al fine di valutare l'effetto di una diversa illuminazione, le stesse fotografie sono state fatte in ambiente interno con luce artificiale ed in ambiente esterno con luce naturale.
Per comprendere la possibilitàdi contare le mele sono state fotografate differenti situazioni, in particolare: presenza di tutte le mele, mancanza di una mela in posizione centrale, mancanza di una mela lungo al bordo della cassetta, mancanza di una mela che era parzialmente coperta dal bordo anteriore della cassetta, mancanza di due mele vicine, mancanza di due mele non vicine, mancanza di tre mele tutte vicine, mancanza di tre mele di cui due vicine, mancanza di tre mele tutte non vicine.
Una cosa analoga è stata fatta per le palline e le uova, l'unica differenza è che in questo caso sono stati impiegati uno sfondo bianco ed uno sfondo blu e gli elementi da fotografare non sono stati disposti in una cassetta ma su di una superficie libera.
In totale sono circa 9 situazione per ogni sfondo (2) e per ogni situazione di illuminazione (2); per un totale di 36 foto per tipo di elemento fotografato. Sono state scattate circa 150 fotografie.

Oltre a segmentare la fotografie così ottenute ci si è posti anche l'obbiettivo di valutare se alcuni tipi di pre-elaborazione dell'immagine potessero in qualche modo migliorare il risultato della segmentazione. Per questo motivo sono state prodotte altre quattro versioni delle immagini, ognuna delle quali è stata ricavata dell'originale tramite le seguenti elaborazioni:

a) filtro mediano su vicinato a 24
b) equalizzazione mediante una curva che è stata impostata in corrispondenza di ogni differente elemento, sfondo e luce.
c) agoritmo di color constancy "white-patch". Il principio sul quale si basa tale algoritmo è che in un'immagine ci deve senz'altro essere il colore bianco, quindi il colore più prossimo al bianco viene trasformato in bianco ed utilizzato per calcolare un coefficiente di moltiplicazione delle componenti di colore di tutti i pixels.

d) algoritmo di color constancy “grey-world”. Il principio sul quale si basa tale algoritmo è che in ogni immagine la media della componente di rosso estesa a tutti i pixels è uguale alla media estesa a tutti i pixels delle componenti di verde e di blu.

Le elaborazioni ”c” e “d” hanno l’obbiettivo di svincolare le caratteristiche dell’immagine alle caratteristiche spettrali dell’illuminante.

Per quanto riguarda l’elaborazione “b” (equalizzazione), le curve utilizzate sono tutte contenute nella galleria degli effetti denominata effetti che si trova nella directory principale di ogni CD ROM. Tale libreria è utilizzabile dal programma Photoimpact di Ulead Systems.
(Menu “finestra”, si seleziona “Tegola con facile tavolozza”, si procede con un clic su una delle 4 icone presentate. Nell’ambiente che appare saràpossibile importare una nuova galleria di effetti)

In totale le fotografie segmentate sono state circa 700.

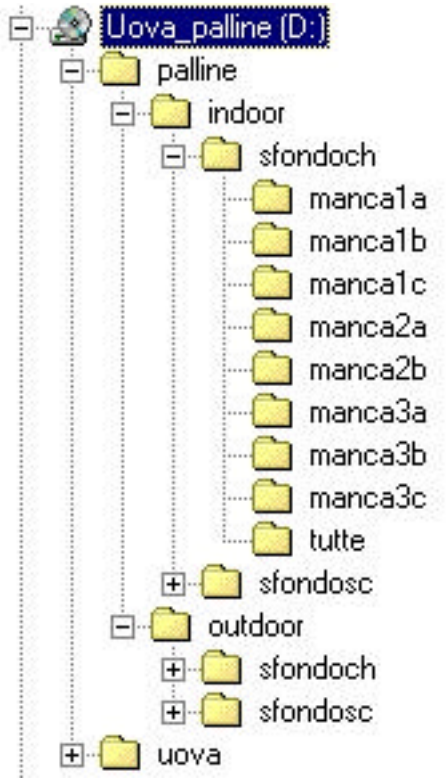
LA SEGMENTAZIONE

L’algoritmo di segmentazione utilizzato è stato trovato al sito internet www.caip.rutgers.edu.
Esso si basa sul mean shift algorithm, una tecnica che stima i gradienti di densità
Per un maggior approfondimento circa l’algoritmo impiegato si rimanda al codice sorgente allegato ed all’articolo ad esso relativo “ Robust Analysis of Feature Spaces: Color Image Segmentation”, Proceedings of the IEEE Conference of Computer Vision and Pattern Recognition, San Juan, Puerto Rico, June 1997, 750-755.
La versione trovata in Internet è stata parzialmente modificata per la presente applicazione ed è riportata nella directory “swsegm” del floppy disk allegato.
Il software utilizzato non richiede l’impostazione di alcun parametro. Al fine di ottenere una segmentazione più adeguata alla nostra applicazione, nella quale gli elementi da segmentare sono di grossa dimensione, è stato modificato il numero minimo di pixels per cluster. La versione originale del software prevedeva un minimo di 240 pixels; nel presente elaborato sono state utilizzate due opzioni: 600 ed 800 pixels. La dimensione minima di 800 pixels è stata utilizzata nella segmentazione delle uova (sia indoor che outdoor) e delle palline outdoor. La versione 600 pixels è invece stata utilizzata in tutti gli altri casi.

COME E’ STATA ORGANIZZATA LA RACCOLTA DI IMMAGINI

Le immagini sono state acquisite tramite una fotocamera digitale dotata di CCD da 1536X1024 pixels e sono state registrate in modalitàJPG alta qualità(400KB per immagine).
Successivamente tutte le immagini sono state scalate del 50% ottenendo quindi immagini da 768x512.

Le immagini sono state raccolte in due CD. Il primo dedicato unicamente alle mele, il secondo a palline e uova.
Per ogni elemento fotografato (mele gialle, mele rosse, palline e uova) le immagini sono state organizzate secondo la struttura di seguito riportata.



Legenda:	Indoor	Ambiente interno con illuminazione artificiale (lampada a risparmio energetico)
	Outdoor	Ambiente esterno con illuminazione naturale (giornata con cielo coperto)
	sfondoch	sfondo chiaro. Bianco
	sfondosc	sfondo scuro. Nero per le mele e blu per palline e uova
	manca1a:	manca un elemento in posizione centrale
	manca1b	manca un elemento vicino al bordo lungo
	manca1c	manca un elemento vicino al bordo corto (quello parzialmente coperto per le mele)
	manca2a	mancano due elementi contigui
	manca2b	mancano due elementi non contigui
	manca3a	mancano tre elementi contigui

manca3b	mancano tre elementi di cui due contigui ed uno no
manca3c	mancano tre elementi non contigui
tutte	non manca alcun elemento
10vicini	Solo per le palline outdoor sfondo scuro. 10 palline contigue.
12 vicini	Solo per le palline outdoor. 12 palline contigue.
7vicini	Solo per le palline outdoor sfondo chiaro. 7 palline contigue.
9vicini	Solo per palline outdoor sfondo chiaro

All'interno delle sottocartelle sono presenti i seguenti 10 files:

Nome del file	Descrizione
origin.tif	immagine originale non elaborata
sorigin.tif	risultato della segmentazione dell'immagine originale
equal.tif	immagine equalizzata
sequal.tif	segmentazione dell'immagine equalizzata
mediano.tif	immagine ottenuta dall'originale dopo il filtro mediano
smediano.tif	segmentazione dell'immagine filtrata
white.tif	immagine originale elaborata con l'algoritmo "white patch"
swhite.tif	segmentazione dell'immagine "white.tif"
grey.tif	immagine originale elaborata con l'algoritmo "grey-world"
sgrey.tif	segmentazione dell'immagine "grey.tif"

I RISULTATI

Di seguito sono riportati alcuni commenti circa le immagini ottenute dalla segmentazione per i vari elementi in corrispondenza delle differenti condizioni e tipi di elaborazione. Per completezza sono stati riportati i casi più significativi di ogni elemento analizzato.

Il risultato ottenuto dalla segmentazione delle mele in ambiente interno, dunque con luce artificiale, non è soddisfacente in a causa della no uniformitàdell'illuminazione che ha provocato la formazione di ombre e la conseguente elevata difficoltànell'individuare la presenza delle mele parzialmente coperte dal bordo di cartone della scaltola. Come si può notare dalle immagini sotto riportate nel caso di mancanza di due mele (una centrale ed una nel bordo in basso a destra) l'immagine originale, quella elaborata con il white path e quella con il filtro mediano, non consentono di vedere la mela che si trova nella seconda riga, parzialmente nascosta dal bordo. L'immagine con elaborazione grey world produce un risultato migliore, comunque inferiore a quello fornito dall'immagine equalizzata. Quest'ultima ha però lo svantaggio di ridurre il margine di separazione tra mele contigue,

aumentando quindi la probabilitàche due mele appartengano allo stesso cluster.

Immagine originale

Immagine segmentata

Immagine con filtro white patch

Immagine segmentata

Immagine con filtro mediano

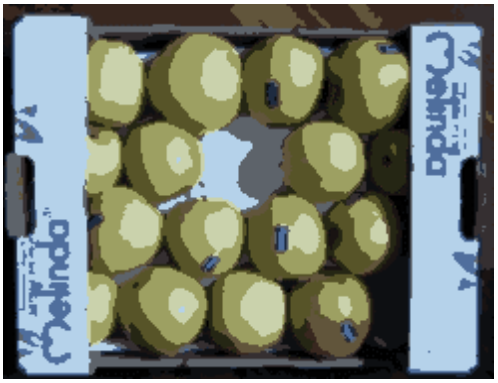


Immagine segmentata

Immagine con filtro grey world

Immagine segmentata

Immagine equalizzata

Immagine segmentata

Le immagini presentate si trovano nella cartella /mele/indoor/sfondoch/manca2a del CD “mele” e sono relative a mele gialle su sfondo chiaro. Le valutazioni fatte si applicano anche alle mele chiare in ambiente interno si sfondo scuro, anche se queste producono un risultato leggermente migliore.

Per qunto riguarda invece l’ambiente esterno, sempre per le mele gialle, i risultati sono migliori, anche se le fotografie sono state scattate in una giornata di cielo parzialmente coperto ed i raggi solari erano tali da produrre un’ ombra su un bordo della scatola. Di seguito sono riportate le immagini di mele, outdoor su sfondo chiaro. Si può notare come giàdalla segmentazione dell’immagine originale e quella con filtro white patch sia possibile distinguere, anche se non nettamente, le mele parzialmente nascoste dal bordo della scatola.
Anche in questo caso l’immagine equalizzata consente di individuare completamente tutte le mele ma riduce il margine di separazione tra le stesso.
Un buon compromesso è costituito dalla segmentazione dell’immagine con filtro grey world.

Immagine originale

Immagine segmentata

Immagine con filtro white patch

Immagine segmentata

Immagine con filtro mediano

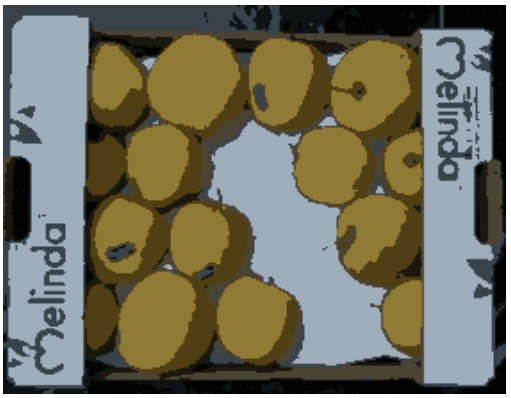


Immagine segmentata

Immagine con filtro grey world

Immagine segmentata

Immagine equalizzata

Immagini segmentata

Le immagini riportate si trovano nella cartella /mele/gialle/outdoor/sfondoch/manca3c.
Dalla immagini presentate si evince che per ottenere un ottimo risultato è opportuno fotografare la scena in condizioni di luce uniforme. In tale situazione, infatti, si ha l'assenza di ombre che possono nascondere le mele.
Si può inoltre osservare come il filtro grey world costituisca un buon algoritmo di pre elaborazione dell'immagine prima della segmentazione, esso consente infatti di segmentare buona parte della superficie della mela sotto lo stesso colore senza d'altronde ridurre eccessivamente la zona di separazione tra elementi contigui.

Si passa ora ad analizzare i risultati ottenuti dalla segmentazione delle mele rosse.
Le immagini riportate sono relative alle mele in ambiente interno. Come si può notare anche in questo caso la presenza di una luce decisamente non uniforme fa sì che le mele sul lato destro siano difficili da individuare. Nel caso delle mele rosse, inoltre, la segmentazione è meno uniforme a causa della disuniformitàdi colore delle mele stesse. Questo aspetto è ancora più marcato per le immagini acquisite in ambiente esterno.

Immagine originale

Immagine segmentata

Immagine con filtro white patch

Immagine segmentata

Immagine con filtro mediano

Immagine segmentata

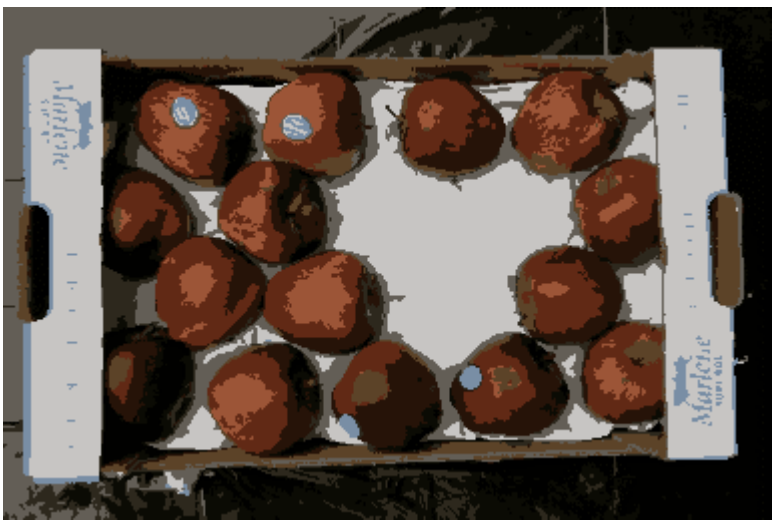
Immagine con filtro grey world

Immagine segmentata

Immagine equalizzata

Immagine segmentata

Le immagini presentate si trovano nella cartella /mele/rosse/indoor/sfondoch/manca3c.
Il risultato ottenuto dalla segmentazione delle mele rosse outdoor è analogo a quello ottenuto per le mele gialle: si ha un miglioramento rispetto alle mele indoor ma c'è un ulteriore margine di perfezionamento ottimizzando le condizioni di illuminazione. In ogni caso il risultato è buono.
Per le mele rosse outdoor, inoltre, si riscontra, come giàanticipato, una maggior irregolaritàdella segmentazione dovuta al fatto che le mele stesse hanno un colore irregolare, più di quanto non lo sia quello delle mele gialle. Questo asetto può essere verificato osservando l'immagine sottostante, relativa alla segmentazione di un'immagine equalizzata.



Rimangono ora da analizzare i risultati ottenuti dalla segmentazione delle palline e delle uova.
Per quanto riguarda le palline, anche se l'ambiente interno è caratterizzato da un'illuminazione scadente, la segmentazione su sfondo scuro ha dato ottimi risultati, decisamente meno brillanti sono invece stati i risultati su sfondo chiaro, a causa dello scarso contrasto tra le palline ed il fondo.

Immagine originale

Immagine segmentata

Immagine con filtro white patch

Immagine segmentata

Immagine con filtro mediano

Immagine segmentata

Immagine con filtro grey world

Immagine segmentata

Immagine con equalizzata

Immagine segmentata

L immagini presentate si trovano nella cartella /palline/indoor/sfondoch/manca1c del CD PALLINE&UOVA.
La segmentazione delle palline outdoor, pur presentando migliori risultati rispetto a quelli presentati, risente parecchio dello scarso contrasto tra il colore delle palline e dello sfondo. Nel caso in cui il fondo è scuro si ottiene invece un ottimo risultato, sia in ambiente interno che esterno. Di seguito sono riportate le immagini relative alle palline outdoor su sfondo scuro.

Immagine originale

Immagine segmentata

Immagine con filtro white patch

Immagine segmentata

Immagine con filtro mediano

Immagine segmentata

Immagine con filtro grey world

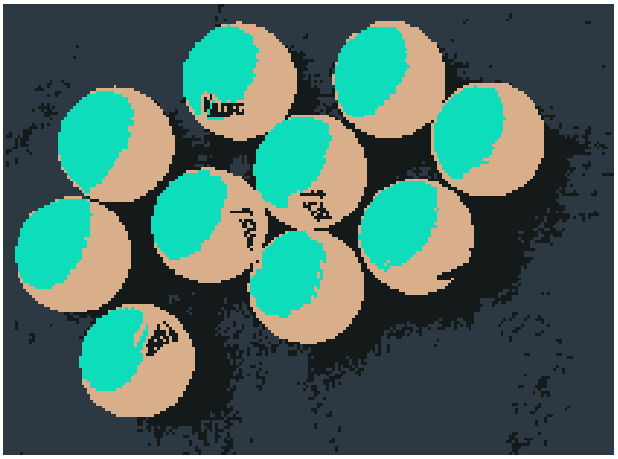


Immagine segmentata

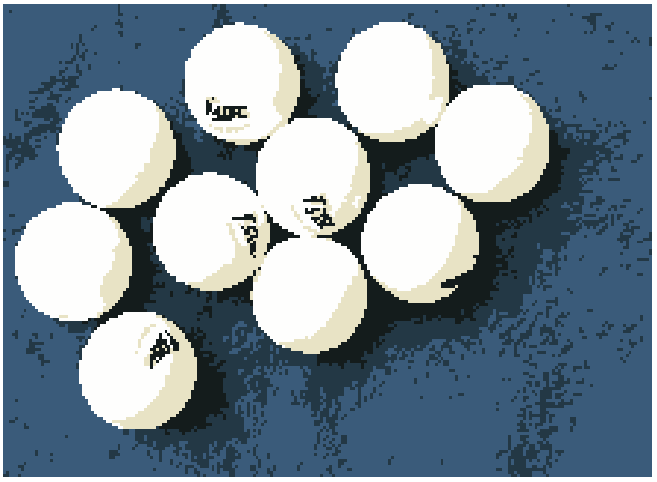


Immagine equalizzata

Immagine segmentata

La immagini riportate si trovano nella cartella /palline/outdoor/sfondosc/10sparse del CD PALLINE&UOVA.

In ultimo rimane da analizzare la segmentazione delle uova.
In questo caso, grazie all'uniformitàdel colore delle uova ed al buon contrasto con lo sfondo, sia chiaro che scuro, il risultato è stato buono sia in ambiente interno che esterno.
In ogni caso, in ambiente interno le cattive condizioni di luminositàportano ad un risultato meno brillante rispetto all'ambiente esterno.
Il risultato ottenuto con le uova è poco sensibile ai vari tipi di pre elaborazione dell'immagine o al colore dello sfondo, come può essere riscontrato nelle immagini seguenti.

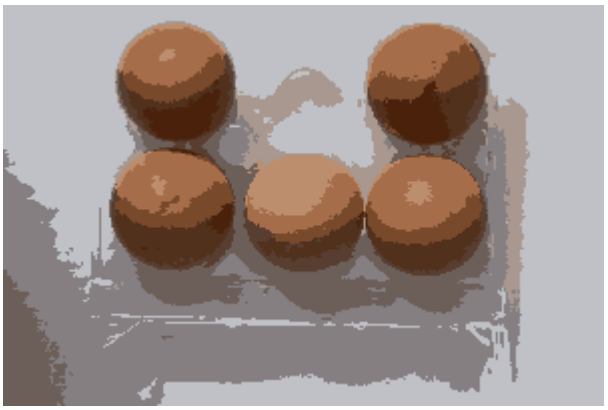


Immagine originale /uova/indoor/sfondoch/manca1b

Immagine segmentata

Immagine equalizzata
/uova/indoor/sfondoch/manca1b

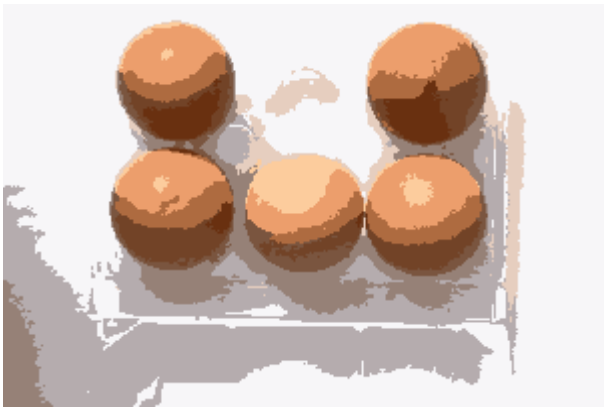


Immagine segmentata

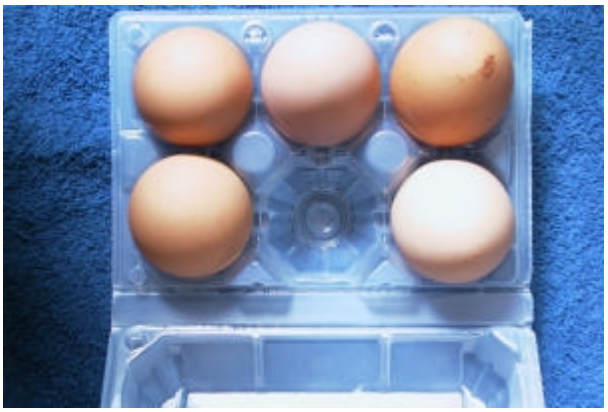


Immagine originale
/uova/indoor/sfondosc/manca1b

Immagine segmentata

Immagine equalizzata
/uova/indoor/sfondosc/manca1b

Immagine segmentata

Immagine originale

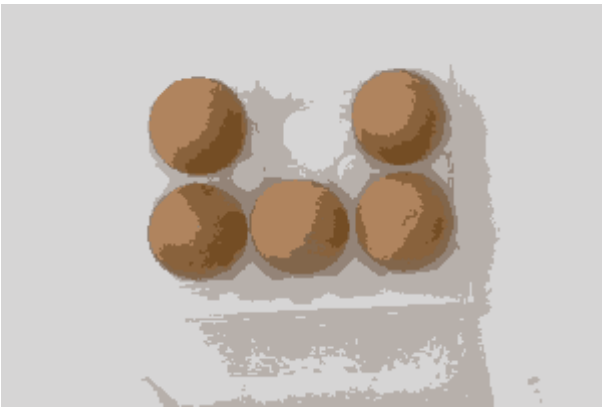


Immagine segmentata

Immagine equalizzata
/uova/outdoor/sfondoch/manca1a

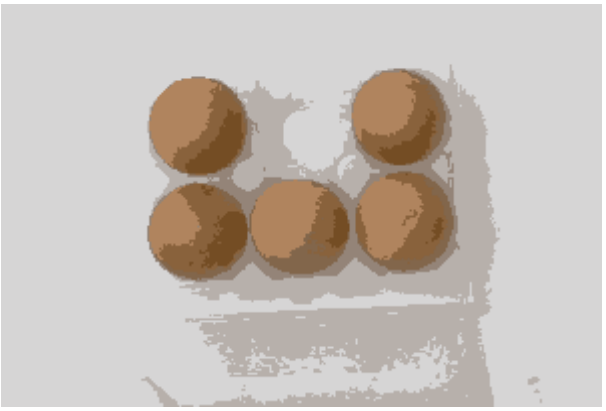


Immagine segmentata

Il risultato ottenuto con la segmentazione delle uova outdoor su sfondo scuro è analogo a quello ottenuto su sfondo chiaro.

CONCLUSIONI

Il lungo lavoro di acquisizione delle immagini e successiva elaborazione ha permesso di rilevare che la segmentazione di elementi caratterizzati da colore uniforme è un compito agevole anche in presenza di condizioni di illuminazione non uniforme, purchè si mantenga un buon contrasto tra lo sfondo e l'elemento di interesse. In tale situazione, la pre elaborazione dell'immagine non apporta un significativo miglioramento della segmentazione ai fini del conteggio degli elementi.
Quando l'oggetto da contare ha invece un colore non uniforme divengono più critiche le condizioni di illuminazione e giocano un ruolo significativo anche i differenti tipi di pre elaborazione dell'immagine.

ALLEGATI

```
//Color Image Segmentation
//This is the implementation of the algorithm described in
//D. Comaniciu, P. Meer,
//Robust Analysis of Feature Spaces: Color Image Segmentation,
//http://www.caip.rutgers.edu/~meer/RIUL/PAPERS/feature.ps.gz
//appeared in Proceedings of CVPR'97, San Juan, Puerto Rico.
// ===========================================================================
// =====      Module: segm_main.cc
// ===== ------------------------------------------------------------ ======
// =====      Version 01   Date: 04/22/97
// ===== ------------------------------------------------------------ ======
// ===========================================================================
// =====      Written by Dorin Comaniciu
// =====      e-mail:  comanici@caip.rutgers.edu
// ===========================================================================
// Permission to use, copy, or modify this software and its documentation
// for educational and research purposes only is hereby granted without
// fee, provided that this copyright notice appear on all copies and
// related documentation.  For any other uses of this software, in original
// or modified form, including but not limited to distribution in whole
// or in part, specific prior permission must be obtained from
// the author(s).
//
// THE SOFTWARE IS PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND,
// EXPRESS, IMPLIED OR OTHERWISE, INCLUDING WITHOUT LIMITATION, ANY
// WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.
//
// IN NO EVENT SHALL RUTGERS UNIVERSITY BE LIABLE FOR ANY SPECIAL,
// INCIDENTAL, INDIRECT OR CONSEQUENTIAL DAMAGES OF ANY KIND, OR ANY
// DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
// WHETHER OR NOT ADVISED OF THE POSSIBILITY OF DAMAGE, AND ON ANY
// THEORY OF LIABILITY, ARISING OUT OF OR IN CONNECTION WITH THE USE
// OR PERFORMANCE OF THIS SOFTWARE.
// ===========================================================================
//

#include    <stdlib.h>
#include    <math.h>
#include    <memory.h>
#include    <fstream.h>
#include    <strstream.h>
#include    "segm.hh"
#include    <string.h>


extern int option;

static RasterIpChannels* read_PPM_file( istream* fp );
Boolean write_PPM_file( char* fname, RasterIpChannels* signal );
Boolean my_write_PGM_file( char* fname, Octet* signal,int _rows, int _colms );

extern char pgmfile[50];

/*************************************************************/
/*                                                           */
/*    Run with:                                              */
/*         segm image_name.ppm                                  */
/*                                                        */
/*************************************************************/

int main( int argc, char **argv )
{
  int i;
```

```
  SegmenterMS::sRectangle rects[Max_rects];

  istream*  fp;
  fp = new ifstream( argv[1], ios::in );
  if ( fp->fail() )
      {
        cerr << __FILE__ << '(' << __LINE__ << "):  cannot open file " <<
          argv[1] << endl;
          cerr << "Usage: segm image_name.ppm output_image.ppm"<< endl;
        exit(1);
      }
  ostrstream     cmd;

  long      selects = 0L;
  selects |= Lightness; selects |= Ustar; selects |= Vstar;
  int p_dim=3;

  assert( p_dim <= p_max );   // must be since all array dimensions are such
  Boolean       block = false;
  unsigned int    seed  = 29254088; //random # generator



  int       n_rect = 0;
  Boolean   done = false;

 strcpy(pgmfile,argv[2]);
 pgmfile[strlen(pgmfile)-2]='g';         //nostre robe modifica

 cout <<"\n\nAlgoritmo di segmentazione di Comaniciu e Meer.\n\n";

 cout <<"File di input  : "<<argv[1]<<"\n";
 cout <<"File di output : "<<argv[2]<<"\n";


/*  cout << "Select:"<< endl
       <<"      w = undersegmentation inside a window"<< endl
       <<"      u = undersegmentation" << endl
       <<"      o = oversegmentation" << endl
       <<"      q = quantization" << endl;
  */
  char received='u';

  while(1)
   {
    // cin >> received;
     if((received =='w') || (received=='W'))
       {
         option=2; break;
       }
     else if((received =='u') || (received=='U'))
       {
         option=2; done=true; break;
       }
     else if((received =='o') || (received=='O'))
       {
         option=1; done=true; break;
       }
     else if((received =='q') || (received=='Q'))
       {
         option=0; done=true; break;
       }
     else
```

```
       {
         cout << endl <<"Please type w, u, o or q !" << endl;
       }
    }
  #ifdef ORG_SIZE
      cmd << "ee " << argv[1] << " &" << ends;
  #else
      cmd << "ee -geometry 120x120 " << argv[1] << " &" << ends;
  #endif
      system( cmd.str() );

  while ( !done )
    {
      cout << "***********  This is the " << n_rect+1 <<
        (n_rect==0 ? "st" : (n_rect==1 ? "nd" : (n_rect==2 ? "rd" : "th")))
        << " Window!  ************" << endl;
      cout << "The Upper_Left Corner Coords:" << endl;
      int   x1, y1;
      cin >> x1 >> y1;
      cout << endl;
      if((x1 < 0) || (y1 < 0))
        {
        cout << "********* " << n_rect << " Windows totally *******" << endl;
        done = true;
        break;
      }
      cout << "The Width and Height of window:" << endl;
      int   d1, d2;
      cin >> d1 >> d2;
      cout << endl;
      if ( (d1 == 0) || (d2 == 0) )
      {
        cout << "********* " << n_rect << " Windows totally *******" << endl;
        done = true;
        break;
      }

      rects[n_rect].x = x1; rects[n_rect].y = y1;
      rects[n_rect].width = d1; rects[n_rect].height = d2;
      n_rect++;
    }
  RasterIpChannels*     signal = read_PPM_file( fp );
  ((ifstream* )fp)->close();

  SegmenterMS     segmenter;
  segmenter.ms_segment( signal, rects, n_rect, selects, seed, block);
  RasterIpChannels*     sig_result = segmenter.result_ras_;

  write_PPM_file( argv[2], sig_result );

  ostrstream      cmd1;
  ostrstream      cmd2;

#ifdef ORG_SIZE
      cmd1 << "ee "<<argv[2]<<" &" << ends;
  if(option && !n_rect)
      cmd2 << "ee "<<pgmfile<<" &" << ends;
#else
      cmd1 << "xv -geometry 120x120 result.ppm &" << ends;
  if(option && !n_rect)
      cmd2 << "xv -geometry 120x120 result.pgm &" << ends;

#endif
```

```
      system( cmd1.str() );
  if(option && !n_rect)
      system( cmd2.str() );

  delete signal;
  delete sig_result;
  return( 0 );
}

static void skip( istream* fp )
{
  char      c1, c2;
  if ( (c1 = fp->get() ) != '#' ) {
    fp->putback( c1 );
    return;
  }
  while ( c1 == '#' )
    {
      while ( (c2 = fp->get()) != '\n' )
      ;
      c1 = fp->get();
      if ( c1 != '#' )
      fp->putback( c1 );
    }
  return;
}

static RasterIpChannels* read_PPM_file( istream* fp )
{
  int c1 = fp->get();
  int c2 = fp->get();
  int c3 = fp->get();

// test point

  if ( c1 == 'P' && c3 == '\n' )
    {
      Octet**     datain = new Octet*[p_max];
      int   w, h, m;

      switch ( c2 ) {
      case '3':
      {
        skip( fp );
        *fp >> w >> h;
        int i;
        for ( i = 0; i < p_max; i++ ) {
          datain[i] = new Octet[w * h];
        }
        fp->get();
        skip( fp );
        *fp >> m;

          //need a test for # comments !!!
        for ( int j = 0, idx = 0; j < h; j++ ) {
          for ( i = 0; i < w; i++, idx++ ) {
            *fp >> c1 >> c2 >> c3;   //ASCII decimal values
            datain[0][idx] = c1;
            datain[1][idx] = c2;
            datain[2][idx] = c3;
          }
        }
      }
```

```
      break;

      case '6':
      {
        skip( fp );
        *fp >> w >> h;

        for ( int i = 0; i < p_max; i++ ) {
          datain[i] = new Octet[w * h];
        }
        fp->get();
        skip( fp );
        *fp >> m;

// test point
        fp->get();
        skip( fp );

          cout << "Dimensioni : " << w << " x " << h << endl;
//          cout << "Max Value  m: " << m << endl;
          Octet *temp_buf = new Octet[h*w*3];
        fp->read(temp_buf, h*w*3);

          Octet *temp0 = datain[0];
        Octet *temp1 = datain[1];
        Octet *temp2 = datain[2];

        for ( register int j = 0, idx = 0; j < h*w; j++) {
            temp0[j] = temp_buf[idx++];
            temp1[j] = temp_buf[idx++];
            temp2[j] = temp_buf[idx++];
          }
          delete [] temp_buf;
      }
      break;

      default:
      cerr << "File is not the PPM format." << endl;
      return NULL;
      }

      XfRaster::Info    info;
      info.rows = h;
      info.columns = w;
      info.origin_x = 0;
      info.origin_y = 0;
      return (new RasterIpChannels( info, p_max, eDATA_OCTET,
                           datain, true ) );
    }
  cerr << "File is not the PPM format." << endl;
  return NULL;
}

Boolean my_write_PGM_file( char* fname, Octet* signal,int _rows, int _colms )
{
  ofstream fp( fname, ios::out );
  if ( fp.fail() )
    return false;

  fp << "P5\n" << _colms << ' ' << _rows<< "\n255" << endl;
  fp.write(signal,_rows*_colms);
  fp.close( );
  return true;
```

```
}

Boolean write_PPM_file( char* fname, RasterIpChannels* signal )
{
  ofstream fp( fname, ios::out );
  if ( fp.fail() )
    return false;

  fp << "P6\n" << signal->columns_ << ' ' << signal->rows_ << "\n255" << endl;

  assert( signal->dtype_ == eDATA_OCTET );
  Octet *temp_buf = new Octet[signal->rows_*signal->columns_*3];
  Octet *temp0 = signal->chdata_[0];
  Octet *temp1 = signal->chdata_[1];
  Octet *temp2 = signal->chdata_[2];
  for ( register int j = 0, idx = 0; j < signal->rows_*signal->columns_; j++) {
     temp_buf[idx++]=temp0[j];  //red
     temp_buf[idx++]=temp1[j];  //green
     temp_buf[idx++]=temp2[j];  //blue
   }
  fp.write(temp_buf,signal->rows_*signal->columns_*3);
  delete [] temp_buf;

  fp.close( );
  return true;
}

// Class constructor
// The `data' may be taken away from the caller in order to
// avoid time-consuming copying of the data.  However,
// the caller has to give explicit permission for this.
RasterIpChannels::RasterIpChannels(
    const XfRaster::Info& info, const int n_channels,
    const DataType dtype, Octet* data[], const Boolean do_take
) {
    rows_ = info.rows;
    columns_ = info.columns;
    dtype_ = dtype;
    clipf_ = false;
    n_channels_ = n_channels;
    if (n_channels_ == 0) {
      n_channels_ = 1;
    }
    size_t size = (size_t)(rows_ * columns_);
    chdata_ = new Octet*[n_channels_];
    for (int channel = 0; channel < n_channels_; channel++) {
      if ( do_take == true ) {     // take over the data from the caller
          chdata_[channel] = (Octet* )data[channel];
          data[channel] = nil;
      } else {
          if ( dtype_ == eDATA_FLOAT )       size *= sizeof(float);
          chdata_[channel] = new Octet[size];
          if (data != nil && data[channel] != nil) {
            memmove( chdata_[channel], data[channel], size );
          } else {
            memset( chdata_[channel], 0, size );
          }
      }
    }
  delete [] data;
}

RasterIpChannels::~RasterIpChannels() {
```

```
    for (int channel = 0; channel < n_channels_; channel++) {
      if (chdata_[channel])   delete [] chdata_[channel];
      chdata_[channel] = nil;
    }
    delete [] chdata_;
}

// RANGE forces `a' to be in the range [b..c] (inclusive)
inline void RANGE( int& a, const int b, const int c )
{
  if ( a < b ) {
      a = b;
  } else if ( a > c ) {
      a = c;
  }
}

void RasterIpChannels::raster_info(Info& i) {
    i.rows = rows_;
    i.columns = columns_;
    i.origin_x = 0;
    i.origin_y = 0;
}

// Move floating point array to Octet array, i.e. to [0..255]
// The result is either scaled to the range [0..255] or
// clipped to this range, depending on the flag `clipf'.
// Note:  Handles only 1-Octet per pixel pictures
// (i.e. mono/pseudo color pictures)
Octet** RasterIpChannels::raster_float_to_Octet(
    RasterIpChannels& ras
) {
    assert( ras.dtype() == eDATA_FLOAT );

    float   maxv = -1.0e38;
    XfRaster::Info      info;
    ras.raster_info(info);
    size_t  size = (size_t)(info.rows * info.columns);

    Octet** data = ras.chdata();
    int                 channels = ras.n_channels();
    Octet** picRes = new Octet*[channels];
    int                 i;
    for ( i = 0; i < channels; i++ )
      picRes[i] = new Octet[ size ];

    if ( ras.clipf() == true ) {  // clip the values outside the range [0..255]
      int   p;
      for ( i = 0; i < channels; i++ ) {
          register float*           ptr1 = (float* )data;
          register Octet*     ptr2 = picRes[i];
          for ( register int off = 0; off < size; off++, ptr1++, ptr2++ ) {
            p = int(*ptr1);
            RANGE( p, 0, 255 );
            *ptr2 = Octet(p);
          }
      }
    } else {                // scale the values to the range [0..255]
      for ( i = 0; i < channels; i++ ) {
          float               minv = (float) 1.e38;
          float               maxv = (float) -1.e38;
          register float*           ptr1 = (float* ) data[i];
          register Octet*     ptr2 = picRes[i];
```

```
          register int  off;
          for ( off = 0; off < size; off++, ptr1++ ) {
            if ( *ptr1 < minv )   minv = *ptr1;
            if ( *ptr1 > maxv )   maxv = *ptr1;
          }
          ptr1 = (float* ) data[i];
          float   ratio = (float) 255.0 / (maxv - minv);
          for ( off = 0; off < size; off++, ptr1++, ptr2++ )
              *ptr2 = Octet( (*ptr1 - minv) * ratio );
      }
    }
    return ( picRes );
}
```

```
static int act_threshold;

#define my_abs(a) ((a) > 0 ? (a): (-a))
#define SQRT2 1.4142
#define SQRT3 1.7321
static const float      BIG_NUM      =    1.0e+20;

// Coefficient matrix for xyz and rgb spaces
static const int    XYZ[3][3] = { { 4125, 3576, 1804 },
                                  { 2125, 7154,  721 },
                                  {  193, 1192, 9502 } };
static const float  RGB[3][3] = { { 3.2405, -1.5371, -0.4985 },
                                  {-0.9693,  1.8760,  0.0416 },
                                  { 0.0556, -0.2040,  1.0573 } };

// Constants for LUV transformation
static const float     Xn = 0.9505;
static const float     Yn = 1.0;
static const float     Zn = 1.0888;
static const float     Un_prime = 0.1978;
static const float     Vn_prime = 0.4683;
static const float     Lt = 0.008856;

// # of samples
static const int  Max_J      =      25;

// Limit of the number of failed trials
static const int       Max_trials    =  50;

// Defaults values for the parameters.
static const int  sam_max = 60;

// Few more trials at the end
static const int        MAX_TRIAL=10;

// Used in 3-D histogram computation
static const int FIRST_SIZE=262144; // 2^18
static const int SEC_SIZE=64;  // 2^6

// Make coordinate computation faster
// my_neigh is for auto_segm, my_neigh_r works with region
static int my_neigh[8];
static int my_neigh_r[8];


// Results
static int ORIG_COLORS;
static int SEGM_COLORS;



//Inizio funzioni misurazione tempo
mell_time mell_gettimeofday_get_time(void)
{
     struct timeval tv;
     gettimeofday(&tv, 0);
     return tv;
}

mell_time mell_gettimeofday_time_diff(mell_time t1, mell_time t2)
{
     mell_time diff;
```

```
     diff.tv_sec = t1.tv_sec - t2.tv_sec;
     diff.tv_usec = t1.tv_usec - t2.tv_usec;
     /* normalize */
     while (diff.tv_usec < 0)
       {
       diff.tv_usec += 1000000L;
       diff.tv_sec -= 1;
       }

     return diff;
}
//Fine funzioni misurazione tempo




Boolean my_write_PGM_file( char*, Octet*,int,int);
void    covariance_w(const int N, int M, const int p, int ** data,
                  int *w, float T[],  float C[p_max][p_max]);
void    mean_s(const int N, const int p, int J[], int **data, float T[]);
void    my_convert(int, float *, int *);
void    reverse_map(Octet *, Octet *, int *, Octet *, float T[][p_max]);

// Class constructor
SegmenterMS::SegmenterMS( )
{
  _p = 0;
  _p_ptr = 0;
  _rrows  = 0;
  _rcolms = 0;
  _data_all = nil;
  _data = nil;
  _NJ = Max_J;
}

// Class destructor
SegmenterMS::~SegmenterMS( )
{
  if ( _data_all ) {
    for ( register int i = 0; i < _p; i++ )
      if ( _data_all[i] )   delete [] _data_all[i];
    delete [] _data_all;
  }
  if ( _data ) {
    for ( register int i = 0; i < _p; i++ )
      if ( _data[i] )   delete [] _data[i];
    delete [] _data;
  }
}

// LUV (final_T[]) to RGB (TI[]) conversion
void my_convert(int selects, float *final_T, int *TI)
{
  // this condition is always true
  if ( selects & Lightness && selects & Ustar && selects & Vstar )
    {
      if(final_T[0]<0.1)
        {
        TI[0]=0;TI[1]=0;TI[2]=0;
       }
       else
       {
```

```
          float my_x, my_y, my_z;
        if(final_T[0]< 8.0)
            my_y = Yn * final_T[0] / 903.3;
          else
            my_y = Yn * pow((final_T[0] + 16.0) / 116.0, 3);

          float u_prime = final_T[1] / (13 * final_T[0]) + Un_prime;
          float v_prime = final_T[2] / (13 * final_T[0]) + Vn_prime;

          my_x = 9 * u_prime * my_y / (4 * v_prime);
          my_z = (12 - 3 * u_prime - 20 * v_prime) * my_y / (4 * v_prime);

          TI[0] =int((RGB[0][0]*my_x + RGB[0][1]*my_y + RGB[0][2]*my_z)*255.0);
          TI[1] =int((RGB[1][0]*my_x + RGB[1][1]*my_y + RGB[1][2]*my_z)*255.0);
          TI[2] =int((RGB[2][0]*my_x + RGB[2][1]*my_y + RGB[2][2]*my_z)*255.0);

          if(TI[0]>255) TI[0]=255;
          else if(TI[0]<0) TI[0]=0;

        if(TI[1]>255) TI[1]=255;
          else if(TI[1]<0) TI[1]=0;

        if(TI[2]>255) TI[2]=255;
        else if(TI[2]<0) TI[2]=0;
      }
    }
  else
    {
       TI[0]=(int)final_T[0];
       TI[1]=(int)final_T[1];
       TI[2]=(int)final_T[2];
    }
}

// RGB to LUV conversion
// To gain speed the conversion works on a table of colors (_col_RGB[])
// rather than on the whole image
void SegmenterMS::convert_RGB_LUV( RasterIpChannels* signal, long selects )
{
  int x, y, z, my_temp;

  float l_star, u_star, v_star;
  float u_prime, v_prime;
  register int temp_col, temp_index, temp_ind;
  register int j,k;

  int a00=XYZ[0][0], a01=XYZ[0][1], a02=XYZ[0][2];
  int a10=XYZ[1][0], a11=XYZ[1][1], a12=XYZ[1][2];
  int a20=XYZ[2][0], a21=XYZ[2][1], a22=XYZ[2][2];

  int *A00 = new int[MAXV]; int *A01 = new int[MAXV]; int *A02 = new int[MAXV];
  int *A10 = new int[MAXV]; int *A11 = new int[MAXV]; int *A12 = new int[MAXV];
  int *A20 = new int[MAXV]; int *A21 = new int[MAXV]; int *A22 = new int[MAXV];
  for(j=0; j<MAXV;j++)
    {
      A00[j]=a00*j; A01[j]=a01*j; A02[j]=a02*j;
      A10[j]=a10*j; A11[j]=a11*j; A12[j]=a12*j;
      A20[j]=a20*j; A21[j]=a21*j; A22[j]=a22*j;
    }

  float *my_pow = new float[MAXV];
  for(j=0; j<MAXV;j++)
    my_pow[j]= 116.0 * pow(j/255.0, 0.3333333) - 16;
```

```
Octet* temp_ch0 = signal->chdata(0);
Octet* temp_ch1 = signal->chdata(1);
Octet* temp_ch2 = signal->chdata(2);

int pp;
int *temp0, *temp1, *temp2;
pp = _p_ptr;
if ( selects & Lightness ) temp0 = _data_all[pp++];
if ( selects & Ustar ) temp1 = _data_all[pp++];
if ( selects & Vstar ) temp2 = _data_all[pp++];
_p_ptr=pp;

for ( j = 0; j < _n_colors; j++)
  {
      temp_col=_col_RGB[j];
    int R=temp_col>>16; int G=(temp_col>>8) & 255; int B=temp_col & 255;

      x = A00[R] + A01[G] + A02[B];
      y = A10[R] + A11[G] + A12[B];
      z = A20[R] + A21[G] + A22[B];

      float  tval = y / 2550000.0; //Yn==1
    if ( tval >  Lt)  l_star = my_pow[(int)(tval*255+0.5)];
      else  l_star = 903.3 * tval;

      my_temp = x + 15 * y + 3 * z;
    if(my_temp)
      {
        u_prime = (float)(x << 2) / (float)(my_temp);
        v_prime = (float)(9 * y) / (float)(my_temp);
      }
    else
      {
        u_prime=4.0;
        v_prime=9.0/15.0;
      }

    tval=13*l_star;
      u_star = tval* (u_prime - Un_prime); // Un_prime = 0.1978
      v_star = tval* (v_prime - Vn_prime); // Vn_prime = 0.4683

    _col0[j] = (int)(l_star+0.5);
    if(u_star>0) _col1[j] = (int)(u_star+0.5);
      else _col1[j] = (int)(u_star-0.5);

    if(v_star>0) _col2[j] = (int)(v_star+0.5);
      else _col2[j] = (int)(v_star-0.5);
  }
for(j=0;j<_ro_col;j++)
  {
    temp_col=(((((int)temp_ch0[j])<<8)+(int)temp_ch1[j])<<8)+(int)temp_ch2[j];
    temp_ind=_col_misc[temp_col>>6];
    for(k=temp_ind;k<temp_ind+SEC_SIZE;k++)
    if(_col_RGB[k]==temp_col)
      {
         temp_index=_col_index[j]=k;
         break;
     }
    temp0[j]=_col0[temp_index];
    temp1[j]=_col1[temp_index];
    temp2[j]=_col2[temp_index];
  }
```

```
  delete [] my_pow;
  delete [] A22; delete [] A21; delete [] A20;
  delete [] A12; delete [] A11; delete [] A10;
  delete [] A02; delete [] A01; delete [] A00;
  delete [] _col_misc;
  delete [] _col_RGB;
}

// 3-D Histogram computation
// Implement a trade-off between speed and required memory
void SegmenterMS::my_histogram(RasterIpChannels* signal, long selects)
{
  int *first_tab= new int[FIRST_SIZE];
  _col_misc= new int[FIRST_SIZE];
  int **third_tab;
  int *fourth_tab;
  int *fifth_tab=new int[SEC_SIZE];
  _n_colors=0;

  register int k,j,p,r;
  int temp_ind, sec_ind, third_ind;

  int first_contor=0, third_contor=0;

  memset(first_tab,0,sizeof(int)*FIRST_SIZE);
  memset(_col_misc,0,sizeof(int)*FIRST_SIZE);

  register Octet* ch0 = signal->chdata(0);
  register Octet* ch1 = signal->chdata(1);
  register Octet* ch2 = signal->chdata(2);

  //first_tab -> how many
  for(k=0;k<_ro_col;k++)
    {
      temp_ind=(((ch0[k]<<8)+ch1[k])<<2)+(ch2[k]>>6);
      first_tab[temp_ind]++;
    }
  //_col_misc -> memo position
  for(k=0;k<FIRST_SIZE;k++)
    if(first_tab[k])
      {
        _col_misc[k]=first_contor;
      first_contor++;
      }
  //contors
  fourth_tab=new int[first_contor];
  memset(fourth_tab,0,sizeof(int)*first_contor);
  //tab of pointers to reduced colors
  third_tab=new int *[first_contor];
  first_contor=0;
  for(k=0;k<FIRST_SIZE;k++)
    if(first_tab[k])
      {
        third_tab[first_contor]=new int[first_tab[k]];
      first_contor++;
      }

  for(k=0;k<_ro_col;k++)
    {
      temp_ind=(((ch0[k]<<8)+ch1[k])<<2)+(ch2[k]>>6);
      sec_ind=ch2[k] & 63;
      third_ind=_col_misc[temp_ind];
      third_tab[third_ind][fourth_tab[third_ind]]=sec_ind;
```

```
      fourth_tab[third_ind]++;
    }
  for(k=0;k<first_contor;k++)
    {
      memset(fifth_tab,0,sizeof(int)*SEC_SIZE);
      for(j=0;j<fourth_tab[k];j++)
        fifth_tab[third_tab[k][j]]++;
      for(j=0;j<SEC_SIZE;j++)
        if(fifth_tab[j])
         _n_colors++;
      }
  _col_RGB=new int[_n_colors];
  _m_colors=new int[_n_colors];

  k=0;p=0;
  for(r=0;r<FIRST_SIZE;r++)
    if(first_tab[r])
     {
       memset(fifth_tab,0,sizeof(int)*SEC_SIZE);
       for(j=0;j<fourth_tab[k];j++)
         fifth_tab[third_tab[k][j]]++;
       _col_misc[r]=p;
       for(j=0;j<SEC_SIZE;j++)
         if(fifth_tab[j])
         {
           _col_RGB[p]=(r<<6)+j;
           _m_colors[p]=fifth_tab[j];
             p++;
         }
       delete [] third_tab[k];
       k++;
     }
  delete [] third_tab;
  delete [] fourth_tab;
  delete [] fifth_tab;
  delete [] first_tab;

  _col_all = new int*[3];
  _col0=_col_all[0] = new int[_n_colors];
  _col1=_col_all[1] = new int[_n_colors];
  _col2=_col_all[2] = new int[_n_colors];
  _col_index = new int[_ro_col];
}

// Update _col_remain[], _m_col_remain, and _n_col_remain
void SegmenterMS::my_actual(Octet *my_class)
{
  register int i;
  int temp_contor=n_remain;
  register int *temp_rem= new int[_ro_col];
  memcpy(temp_rem,gen_remain,sizeof(int)*temp_contor);
  n_remain=0;
  for(i=0;i<temp_contor;i++)
    if(!my_class[temp_rem[i]])
      gen_remain[n_remain++]=temp_rem[i];
  delete [] temp_rem;
  memset(_col_remain,0,sizeof(int)*_n_col_remain);
  memset(_m_col_remain,0,sizeof(int)*_n_colors);
  _n_col_remain=0;
  for(i=0;i<n_remain;i++)
    _m_col_remain[_col_index[gen_remain[i]]]++;
  for(i=0;i<_n_colors;i++)
     if(_m_col_remain[i])
```

```
       {
       _col_remain[_n_col_remain]=i;
         _n_col_remain++;
       }
}

// if more than "how_many"  neighbors, consider the point
void SegmenterMS::test_neigh(Octet* my_class, int *selected, int* my_contor, int
how_many)
{
  register int i,j,p,k;
  register Octet* local_class=my_class;
  register int temp_contor=*my_contor;
  register int my_index;
  if(auto_segm) my_index=n_remain;
  else          my_index=_ro_col;
  for ( p = 0, i; p < my_index; p++ )
    {
      if(auto_segm) i=gen_remain[p];
      else          i=p;
      if(!local_class[i])
      {
          int neigh_contor=0, no_neigh=1;
          for(j=0;j<8;j++)
            {
              k=i+my_neigh[j];
              if(k>=0 && k<_ro_col && local_class[k])
              {
                  if(auto_segm  && gen_class[k]!=255) continue;
                neigh_contor++;
                  if(neigh_contor>how_many)
                {
                  no_neigh=0;
                      break;
                }
              }
            }
        if(!no_neigh)
        {
            if(auto_segm) selected[*my_contor]=i;
          *my_contor=*my_contor+1;
        }
      }
    }
  for(i=temp_contor;i<*my_contor;i++)
    local_class[selected[i]]=1;
}


// Find the feature vectors inside the given window
// Use Improved Absolute Error Inequality Criterion
// when computing Euclidean distance
// See J.S.Pan el al, Fast Clustering Alg. for VQ, Pattern Recognition,
// Vol. 29, No. 3, pp. 511-518, 1996

void SegmenterMS::new_auto_loop(float *final_T, Octet *sel_col)
{
  float L,U,V,RAD2,R;
  register int TT0=0, TT1=0, TT2=0;
  register int local_contor=0;
  float final_T0=final_T[0], final_T1=final_T[1], final_T2=final_T[2];
  float RADIUS_S2=SQRT2*RADIUS, RADIUS_S3=SQRT3*RADIUS;
```

```
  for ( register int p = 0, k; p < _n_col_remain; p++ )
        {
            k=_col_remain[p];
              L=_col0[k]-final_T0; if((L=my_abs(L))>=RADIUS) continue;
              U=_col1[k]-final_T1; if((R=my_abs(U)+L)>=RADIUS_S2) continue;
            V=_col2[k]-final_T2; if(R+my_abs(V)>=RADIUS_S3) continue;
              RAD2=L*L+U*U+V*V;
              if(RAD2<RADIUS2)
            {
                  register int r=_m_col_remain[k];
                  TT0+=_col0[k]*r; TT1+=_col1[k]*r; TT2+=_col2[k]*r;
              local_contor+=r;
                  sel_col[k]=1;
            }
         }
  final_T[0]=(float)TT0/(float)local_contor;
  final_T[1]=(float)TT1/(float)local_contor;
  final_T[2]=(float)TT2/(float)local_contor;
}

// The same as above, but for non auto_segmentation
void SegmenterMS::nauto_loop(float *final_T, int *selected,
                             Octet *my_class, int *my_contor)
{
  float L,U,V,RAD2,R;
  register int local_contor=*my_contor;
  float final_T0=final_T[0], final_T1=final_T[1], final_T2=final_T[2];
  float RADIUS_S2=SQRT2*RADIUS, RADIUS_S3=SQRT3*RADIUS;

  for ( register int k = 0; k < _n_points; k++ )
          {
              L=_data[0][k]-final_T0; if((L=my_abs(L))>=RADIUS)  continue;
              U=_data[1][k]-final_T1; if((R=my_abs(U)+L)>=RADIUS_S2)  continue;
            V=_data[2][k]-final_T2; if(R+my_abs(V)>=RADIUS_S3)  continue;
              RAD2=L*L+U*U+V*V;
              if(RAD2<RADIUS2)
            {
              selected[local_contor++]=k;
                  my_class[k]=1;
            }
          }
  *my_contor=local_contor;
}

// Set the Radius of the window
void set_RADIUS(int gen_gen_contor, int final)
{
  if(final==2)      RADIUS=final_RADIUS*1.26;
  else if(final==1) RADIUS=final_RADIUS;
  else              RADIUS=fix_RADIUS[gen_gen_contor];
  RADIUS2=RADIUS*RADIUS;
}

// Test if the clusters have the same mean
int test_same_cluster(int rect, float T[][p_max])
{
  float L,U,V,RAD2;
  for(register int k=0; k<rect;k++)
    {
      L=T[k][0]-T[rect][0]; U=T[k][1]-T[rect][1]; V=T[k][2]-T[rect][2];
      RAD2=L*L+U*U+V*V;
      if(RAD2<1)
        return 1;
```

```
    }
  return 0;
}

// First take only pixels inside the search windows at their final locations
// Then inflate windows to double volume and retain only pixels which are
// neighbors with the previous
void SegmenterMS::get_codeblock1(float T[][p_max], int n_rects)
{
  float L,U,V,RAD2, R, min_RAD2;
  int min_ind;
  register int i,k,u;
  register int pres_class, my_flag;
  register float *ptr;

  if(auto_segm) set_RADIUS(0,0);
  else          set_RADIUS(2,0);

  for(k=0;k<_ro_col;k++)
    {
      min_RAD2=BIG_NUM; min_ind=0;
      for(i=0;i<n_rects;i++)
      {
          ptr=T[i];
          L=_data0[k]-ptr[0]; if(my_abs(L)>=RADIUS)  continue;
          U=_data1[k]-ptr[1]; if(my_abs(U)>=RADIUS)  continue;
        V=_data2[k]-ptr[2]; if(my_abs(V)>=RADIUS)  continue;
          RAD2=L*L+U*U+V*V;
          if(RAD2<min_RAD2)
          {
            min_RAD2=RAD2;
            min_ind=i;
          }
      }
      if(min_RAD2<RADIUS2) gen_class[k]=min_ind;
      else                 gen_class[k]=n_rects;
    }

  if(auto_segm) set_RADIUS(0,1);
  else          set_RADIUS(0,0);

  for(k=0;k<_ro_col;k++)
    if(gen_class[k]==n_rects)
      for(i=0;i<8;i++)
      {
          u=k+my_neigh[i];
          if(u>=0 && u<_ro_col)
            if((pres_class=gen_class[u])!=n_rects)
            {
                ptr=T[pres_class];
                L=_data0[k]-ptr[0]; if(my_abs(L)>=RADIUS)  continue;
                U=_data1[k]-ptr[1]; if(my_abs(U)>=RADIUS)  continue;
                V=_data2[k]-ptr[2]; if(my_abs(V)>=RADIUS)  continue;
                RAD2=L*L+U*U+V*V;
              if(RAD2<RADIUS2) gen_class[k]=pres_class;
            }
        }
}

// Final allocation
void SegmenterMS::get_codeblock(float T[][p_max], int n_rects)
{
  float L,U,V,RAD2, min_RAD2;
```

```
  register int min_ind;
  register int i,k;
  register float *ptr;

  for(k=0;k<_ro_col;k++)
    {
      min_RAD2=BIG_NUM;
      min_ind=0;
      for(i=0;i<n_rects;i++)
      {
          ptr=T[i];
        L=_data0[k]-ptr[0]; U=_data1[k]-ptr[1]; V=_data2[k]-ptr[2];
          RAD2=L*L+U*U+V*V;
          if(RAD2<min_RAD2)
          {
            min_RAD2=RAD2;
            min_ind=i;
          }
      }
      gen_class[k]=min_ind;
     }
}

// Compute the mean of feature vectors mapped into the same color
void SegmenterMS::new_codebook(float T[][p_max], int n_rects)
{
  register int i,k;
  register int *tab_contor = new int[n_rects];
  register int prez_class;
  register float *ptr;

  memset(tab_contor,0,sizeof(int)*n_rects);
  for(i=0;i<n_rects;i++)
    {
      T[i][0]=0.0; T[i][1]=0.0; T[i][2]=0.0;
    }
  for(k=0;k<_ro_col;k++)
   if((prez_class=gen_class[k])!=n_rects)
    {
      ptr=T[prez_class];
      ptr[0]+=_data0[k]; ptr[1]+=_data1[k]; ptr[2]+=_data2[k];
      tab_contor[prez_class]++;
    }
   for(i=0;i<n_rects;i++)
    {
      T[i][0]/=tab_contor[i]; T[i][1]/=tab_contor[i]; T[i][2]/=tab_contor[i];
    }
  delete [] tab_contor;
}

// Determine the final feature palette
void SegmenterMS::optimize(float T[][p_max], int n_rects)
{
  get_codeblock1(T,n_rects);
  new_codebook(T,n_rects);
  if(auto_segm)
    get_codeblock(T,n_rects);
 // cerr<<":";
}


// Inverse of the mapping array used in color elimination
```

```
void reverse_map(Octet *inv_map, Octet *my_map, int *n_rects, Octet
*valid_class, float T[][p_max])
{
  float sec_T[Max_rects][p_max];
  register int u=0, k, j;
  for(j=0;j<*n_rects;j++)
    {
      if(valid_class[j])
      {
          for(k=0;k<3;k++)
            sec_T[u][k]=T[j][k];
          my_map[j]=u;
          inv_map[u]=j;
          u++;
        }
    }
  my_map[*n_rects]=u;
  inv_map[u]=*n_rects;
  *n_rects=u;
  for(j=0;j<*n_rects;j++)
    for(k=0;k<3;k++)
      T[j][k]=sec_T[j][k];
}

// Eliminate colors that have less than "my_lim" connected pixels
void SegmenterMS::eliminate_class(Octet *my_class,int *my_max_region, int
*n_rects,int my_lim, Octet* inv_map, float T[][p_max], REGION *first_region)
{
  register int j, k;
  register Octet *valid_class;
  register REGION *current_region=first_region;

  valid_class=new Octet[*n_rects];
  for(j=0;j<*n_rects;j++)
    {
      if(my_max_region[j]<my_lim) valid_class[j]=0;
      else                        valid_class[j]=1;
    }
  while(1)
    {
      if((current_region->my_class<*n_rects &&
          !valid_class[current_region->my_class]))
        for(k=0;k<current_region->my_contor;k++)
           gen_class[current_region->my_region[k]]=*n_rects;
      if(current_region->next_region_str)
        current_region=current_region->next_region_str;
      else break;
    }
  Octet my_map[Max_rects];
  reverse_map(inv_map,my_map,n_rects,valid_class,T);
  for(k=0;k<_ro_col;k++)
    my_class[k]=my_map[gen_class[k]];
  delete [] valid_class;
  memcpy(gen_class,my_class,_ro_col);
}

// Eliminate regions with less than "my_lim" pixels
void SegmenterMS::eliminate_region(int *n_rects,int my_lim, float T[][p_max],
REGION* first_region)
{
  register int j,u,k,p, pres_class, min_ind;
  register REGION *current_region=first_region;
  register int* region;
```

```
  float *ptr;
  float L,U,V,RAD2,minRAD2;
  int increm;

  while(1)
    {
      if(current_region->my_contor<my_lim)
      {
          set_RADIUS(0,0); increm=4;
        region=current_region->my_region;
          for(k=0;k<current_region->my_contor;k++)
            gen_class[region[k]]=*n_rects;
        while(1)
            {
              Boolean my_flag=0;
              RADIUS+=increm; RADIUS2=RADIUS*RADIUS; increm+=4;
              for(k=0;k<current_region->my_contor;k++)
              if(gen_class[p=region[k]]==(*n_rects))
                {
                  minRAD2=RADIUS2;
                    for(j=1;j<8;j+=2)
                      {
                        u=p+my_neigh[j];
                        if(u>=0 && u<_ro_col)
                        if((pres_class=gen_class[u])!=(*n_rects))
                      {
                            ptr=T[pres_class];
                        L=_data0[p]-ptr[0]; U=_data1[p]-ptr[1];
                            V=_data2[p]-ptr[2]; RAD2=L*L+U*U+V*V;
                        if(RAD2<minRAD2)
                          {
                                minRAD2=RAD2; min_ind=pres_class;
                        }
                      }
                    }
                 if(minRAD2<RADIUS2) gen_class[p]=min_ind;
                   my_flag=1;
                }
             if(!my_flag) break;
          }
      }
      if(current_region->next_region_str)
        current_region=current_region->next_region_str;
      else break;
    }
}

// Destroy the region list
void SegmenterMS::destroy_region_list(REGION *first_region)
{
  register REGION *current_region=first_region;
  while(1)
    {
      delete [] current_region->my_region;
      first_region=current_region;
      if(current_region->next_region_str)
        {
          current_region=current_region->next_region_str;
          delete first_region;
      }
      else
        {
          delete first_region;
```

```
          break;
      }
    }
}

// Connected component main routine
void SegmenterMS::find_other_neigh(int k, int *my_ptr_tab, REGION
*current_region)
{
  register int *ptr_tab=my_ptr_tab;
  register int i,u, j=k, sec_signal;
  register int contor=0;
  register int region_contor=current_region->my_contor;
  register int region_class=current_region->my_class;
  ptr_tab[contor]=j;

  while(1)
    {
      sec_signal=0;
      for(i=1;i<9;i+=2)
      {
        u=j+my_neigh[i];
          if(u>=0 && u<_ro_col)
          if(gen_class[u]==region_class && !taken[u])
          {
              sec_signal=1;
            conn_selected[region_contor++]=u;
            taken[u]=1;
              ptr_tab[++contor]=u;
            }
      }
      if(sec_signal) j=ptr_tab[contor];
      else
       {
           if(contor>1) j=ptr_tab[--contor];
         else break;
       }
    }
  current_region->my_contor=region_contor;
}

// Create the region list
REGION *SegmenterMS::create_region_list(int *my_max_region, int change_type)
{
  register int k, local_label=0;
  register REGION *first_region, *prev_region, *current_region;
  taken = new Octet[_ro_col];
  memset(taken,0,_ro_col);
  conn_selected = new int[_ro_col];
  int *ptr_tab=new int[_ro_col];

  for(k=0;k<_ro_col;k++)
    if(!taken[k])
      {
        current_region=new REGION;
        current_region->my_contor=0;
        current_region->my_class=gen_class[k];
        current_region->my_label=local_label;
        if(k!=0) prev_region->next_region_str=current_region;
      if(k==0){ first_region=current_region;}

      local_label++;
      conn_selected[current_region->my_contor++]=k;
```

```
        taken[k]=1;
        find_other_neigh(k,ptr_tab,current_region);
        if(change_type==0)
        if(my_max_region[current_region->my_class]<current_region->my_contor)
            my_max_region[current_region->my_class]=current_region->my_contor;
        current_region->my_region=new int[current_region->my_contor];

        memcpy(current_region-
>my_region,conn_selected,sizeof(int)*current_region->my_contor);
        prev_region=current_region;
       }
  current_region->next_region_str=0;

  delete [] ptr_tab; delete [] taken; delete [] conn_selected;
  return first_region;
}

// Find connected components and remove small regions of classes
// with small regions
void SegmenterMS::conn_comp(Octet *my_class, int *n_rects, Octet *inv_map, float
T[][p_max],int my_lim, int change_type)
{
  REGION *first_region;
  int *my_max_region;
  if(change_type==0)
    {
      my_max_region = new int[(*n_rects)+1];
      memset(my_max_region,0,sizeof(int)*((*n_rects)+1));
    }
  first_region=create_region_list(my_max_region, change_type);
  if(change_type==0)  //elliminate classes with small regions

eliminate_class(my_class,my_max_region,n_rects,my_lim,inv_map,T,first_region);
  else if(change_type==1)  //elliminate small regions
    eliminate_region(n_rects,my_lim,T,first_region);
  destroy_region_list(first_region);
  if(change_type==0) delete [] my_max_region;
 // cerr<<":";
}


// Cut a rectangle from the entire input data
// Deletes the previous rectangle, if any
void SegmenterMS::cut_rectangle( sRectangle* rect )
{
  if ( _data ) {
    for ( register int i = 0; i < _p; i++ )
      if ( _data[i] )   delete [] _data[i];
    delete [] _data;
  }

  // Set the dimensions of the currently processed region.
  _rrows  = rect->height;
  _rcolms = rect->width;
  _data = new int*[_p];

  register int my_x = rect->x;
  register int my_y = rect->y;
  register int i, j, d;
  for ( i = 0; i < _p; i++ )
      _data[i] = new int[_rcolms*_rrows];

  if(auto_segm)
```

```
    for ( d = 0; d < _p; d++ )
      memcpy(_data[d], _data_all[d],sizeof(int)*_ro_col);
  else
    {
      int   idx1 = my_y * _colms + my_x;
      int   idx2 = 0;
      for ( j = my_y, d;
            j < my_y + _rrows; j++, idx1 += _colms - _rcolms )
        for ( i = my_x; i < my_x + _rcolms; i++, idx1++, idx2++ )
          {
            for ( d = 0; d < _p; d++ )
            _data[d][idx2] = _data_all[d][idx1];
          }
    }
  //cerr<<":";
}

// Compute the mean of N points given by J[]
void mean_s(const int N, const int p, int J[], int **data, float T[])
{
  int TT[p_max];
  register int k, i, j;
  for ( i = 0; i < p; i++ )
    TT[i] = 0;
  for ( i = 0; i < N; i++ )
    {
      k = J[i];
      for ( j = 0; j < p; j++ )
        TT[j] += data[j][k];
     }
   for ( i = 0; i < p; i++ )
      T[i] = (float)TT[i] / (float)N;
}

// Build a subsample set of 9 points
int SegmenterMS::subsample(float *Xmean )
{
  int J[9];
  register int my_contor=0, uj, i0;
  if(auto_segm)
  {
    i0=J[my_contor]=
       gen_remain[int(float(n_remain)*float(rand())/float(RAND_MAX))];
  }
  else
  {
    i0=J[my_contor]=int(float(_n_points)*float(rand())/float(RAND_MAX));
  }

  my_contor++;

  for(register i=0;i<8;i++){
    uj=i0 + my_neigh_r[i];
    if(uj>=0 && uj<_n_points)
      {
        if((auto_segm && gen_class[uj]!=255)) break;
      else
          {
            J[my_contor] = uj;
            my_contor++;
        }
      }
    }
```

```
  mean_s(my_contor, _p, J, _data, Xmean);
  return 1;
}

// Sampling routine with all needed tests
float SegmenterMS::my_sampling( int rect, float T[Max_rects][p_max])
{
  register int  k, c;
  register float L,U,V,Res;
  register float my_dist=max_dist, my_sqrt_dist=fix_RADIUS[0];
  float     TJ[Max_J][p_max];
  int l =  0;       //contor of number of subsample sets
  int ll = 0;       //contor of trials
  float     Xmean[p_max];
  float     Obj_fct[Max_J];

  //Max_trials = max number of failed trials
  //_NJ = max number of subsample sets

  while ( (ll < Max_trials) && (l < _NJ ) )
    {
      if ( subsample(Xmean) )    // the subsample procedure succeeded
      {
            ll = 0; c=0;

            // Save the mean
            for ( k = 0; k < _p; k++ ) TJ[l][k] = Xmean[k];

            // Compute the square residuals (Euclid dist.)
            if(auto_segm)
                  for ( register int p = 0; p < _n_col_remain; p++ )
                {
                 k=_col_remain[p];
                      L=_col0[k]-Xmean[0]; if(my_abs(L)>=my_sqrt_dist) continue;
                 U=_col1[k]-Xmean[1]; if(my_abs(U)>=my_sqrt_dist) continue;
                 V=_col2[k]-Xmean[2]; if(my_abs(V)>=my_sqrt_dist) continue;
                 if(L*L+U*U+V*V<my_dist) c+=_m_col_remain[k];
                }
            else
                for ( k = 0; k < _n_points; k++ )
                {
                      L=_data[0][k]-Xmean[0];
                      if(my_abs(L)>=my_sqrt_dist) continue;
                 U=_data[1][k]-Xmean[1];
                      if(my_abs(U)>=my_sqrt_dist) continue;
                 V=_data[2][k]-Xmean[2];
                      if(my_abs(V)>=my_sqrt_dist) continue;
                      if(L*L+U*U+V*V<my_dist) c++;
                }

              //  Objective functions
            Obj_fct[l]=c;
              l++;
          }
      else ++ll;
    }
  if ( ll == Max_trials && l < 1) return( BIG_NUM ); // Cannot find a kernel

  //  Choose the highest density
  L = -BIG_NUM; c=0;
  for ( k = 0; k < _NJ; k++ )
      if ( Obj_fct[k] > L)
```

```
      {
        L = Obj_fct[k];
        c = k;
      }
  if(Obj_fct[c]>0)
    for(k=0;k<_p;k++)
       T[rect][k]=TJ[c][k];
  else return -BIG_NUM; // Not enough points
  return ( 0 );
}

// Compute the weighted covariance of N points
void covariance_w(const int N, int M, const int p,  int **data,
               int *w, float T[], float C[p_max][p_max])
{
  register int i, j, k, l;
  int TT[p_max];
  for ( i = 0; i < p; i++ )
    TT[i] = 0;
  for ( i = 0; i < M; i++ )
    for ( j = 0; j < p; j++ )
      TT[j] += w[i]*data[j][i];
  for ( i = 0; i < p; i++ )
      T[i] = (float) TT[i] / (float)N;

  for ( i = 0; i < p; i++ )
    for ( j = i; j < p; j++ )
      C[i][j] = 0.0;
  for ( i = 0; i < M; i++ )
    {
      for ( k = 0; k < p; k++ )
      for ( l = k; l < p; l++ )
         C[k][l]+=w[i]*(data[k][i]-T[k])*(data[l][i]-T[l]);
    }
  for ( k = 0; k < p; k++ )
    {
    for ( l = k; l < p; l++ )
      C[k][l] /= (float)(N-1);
    for ( l = 0; l < k; l++ )
      C[k][l] = C[l][k];
    }
}

// initialization
void SegmenterMS::init_neigh(void)
{
  my_neigh[0]= -_colms-1;  my_neigh[1]= -_colms;
  my_neigh[2]= -_colms+1;  my_neigh[3]= +1;
  my_neigh[4]= +_colms+1;  my_neigh[5]= +_colms;
  my_neigh[6]= +_colms-1;  my_neigh[7]= -1;

  my_neigh_r[0]= -_rcolms-1;  my_neigh_r[1]= -_rcolms;
  my_neigh_r[2]= -_rcolms+1;  my_neigh_r[3]= +1;
  my_neigh_r[4]= +_rcolms+1;  my_neigh_r[5]= +_rcolms;
  my_neigh_r[6]= +_rcolms-1;  my_neigh_r[7]= -1;
}

// Init matrices parameters
void SegmenterMS::init_matr(void)
{
  // General statistic parameters for X.
  float     Mg[p_max];         //sample mean of X
  float     C[p_max][p_max];   //sample covariance matrix of X
```

```
  covariance_w(_ro_col, _n_colors, _p, _col_all, _m_colors, Mg, C);

  // Adaptation
  float my_th=C[0][0]+C[1][1]+C[2][2];
  int active_gen_contor=1;

  if(auto_segm)
     fix_RADIUS[0]=gen_RADIUS[option]*sqrt(my_th/100);
  else
    {
      active_gen_contor=rect_gen_contor;
      for(int i=0;i<active_gen_contor;i++)
        fix_RADIUS[i]=rect_RADIUS[i]*sqrt(my_th/100);
    }
  final_RADIUS=fix_RADIUS[active_gen_contor-1]*1.26;
  max_dist=fix_RADIUS[0]*fix_RADIUS[0];
#ifdef TRACE
  printf("\n %.2f %.2f ", fix_RADIUS[0], final_RADIUS);
#endif
  act_threshold=(int)((my_threshold[option]>sqrt(_ro_col)/my_rap[option])?
                 my_threshold[option]:sqrt(_ro_col)/my_rap[option]);
 // cerr<<":";
}

// Init
void SegmenterMS::initializations(RasterIpChannels* pic, sRectangle rects[], int
*n_rects, long selects, int *active_gen_contor)
{
  register int i;
  XfRaster::Info        info;
  pic->raster_info(info);
  _colms = info.columns; _rows  = info.rows; _ro_col = _rows * _colms;

  _data_all = new int*[_p];
  for ( i = 0; i < _p; i++ )
      _data_all[i] = new int[_ro_col];
  _data0=_data_all[0]; _data1=_data_all[1]; _data2=_data_all[2];
  init_neigh();
  my_histogram(pic, selects);
  convert_RGB_LUV( pic, selects );
  gen_class = new Octet[_ro_col];
  memset(gen_class,255,_ro_col);
  if(!(*n_rects))
    {
      auto_segm=1;
      *n_rects=Max_rects;
      n_remain=_ro_col;
      _n_col_remain=_n_colors;
      gen_remain = new int[_ro_col];
      _col_remain = new int[_n_colors];
      _m_col_remain = new int[_n_colors];
      for ( i = 0; i< _ro_col ; i++ )
        gen_remain[i] = i;
      for ( i = 0; i< _n_colors ; i++ )
        _col_remain[i] = i;
      memcpy(_m_col_remain,_m_colors,sizeof(int)*_n_colors);

      for ( i = 0; i < Max_rects ; i++ )
        {
          rects[i].width=_colms; rects[i].height=_rows;
          rects[i].x = 0;        rects[i].y = 0;
        }
      *active_gen_contor=1;
```

```
    }
  else
    {
      auto_segm=0;
      n_remain=0;
      *active_gen_contor=rect_gen_contor;
      option=2;
    }
  init_matr();
  delete [] _m_colors;
 }

// Mean shift segmentation applied on the selected region(s) of an image or
// on the whole image
Boolean SegmenterMS::ms_segment( RasterIpChannels* pic, sRectangle rects[],
                            int n_rects, long selects ,
                                    unsigned int seed_default, Boolean block)
{
      mell_time end0,begin0;
      double t;
      begin0=mell_get_time();

  if (n_rects > Max_rects) return false;
  if ( selects & Lightness || selects & Ustar || selects & Vstar )
    _p=3;
  else return false;

  int contor_trials=0, active_gen_contor;
  float     T[Max_rects][p_max];
  int TI[Max_rects][p_max];
  float L,U,V,RAD2,q;
  register int i,k;

  srand( seed_default ); //cerr<<":";
  initializations(pic, rects, &n_rects, selects, &active_gen_contor);

  // Mean shift algorithm and removal of the detected feature
  int rect;
  //IL PROBLEMA è in questo ciclo for
  for ( rect = 0; rect < n_rects; rect++ )
    {
      _n_points = rects[rect].width * rects[rect].height;
      cut_rectangle( &rects[rect] );
    MS:
      if(auto_segm && contor_trials) _NJ=1;

      q =  my_sampling( rect, T);

      if(q == -BIG_NUM || q == BIG_NUM)
        if(contor_trials++<MAX_TRIAL)  goto MS;
        else                           break;


      float final_T[p_max];
      for ( i = 0; i < _p; i++ ) final_T[i]=T[rect][i];

      int *selected = new int[_ro_col];
      Octet *sel_col = new Octet[_n_colors];
      Octet *my_class = new Octet[_ro_col];

      int my_contor=0, gen_gen_contor=-1, how_many=10;
      while(gen_gen_contor++<active_gen_contor-1)
        {
```

```
        set_RADIUS(gen_gen_contor, 0);
      int gen_contor=0;
        Boolean last_loop=0;

        while(gen_contor++<how_many)
          {
            if(auto_segm)
          {
                memset(sel_col,0,_n_colors);
                new_auto_loop(final_T,sel_col);
                L=T[rect][0]-final_T[0]; U=T[rect][1]-final_T[1];
                V=T[rect][2]-final_T[2]; RAD2=L*L+U*U+V*V;
                if(RAD2<0.1){
                  my_contor=0;
                  memset(my_class,0,_ro_col);
                  for(k=0;k<n_remain;k++)
                {
                      register int p=gen_remain[k];
                if(sel_col[_col_index[p]])
                  {
                    selected[my_contor++]=p;
                          my_class[p]=1;
                  }
                }
                  break;
              }
             else
                 {
                T[rect][0]=final_T[0];T[rect][1]=final_T[1];
                  T[rect][2]=final_T[2];
                 }
              }
             else
             {
                my_contor=0;
                    memset(my_class,0,_ro_col);
                  nauto_loop(final_T, selected, my_class,&my_contor);
                    mean_s(my_contor, _p, selected, _data, final_T);
                    L=T[rect][0]-final_T[0]; U=T[rect][1]-final_T[1];
                    V=T[rect][2]-final_T[2]; RAD2=L*L+U*U+V*V;

                    if(RAD2<0.1)
                      break;
                    else
                     {
                     T[rect][0]=final_T[0];T[rect][1]=final_T[1];
                       T[rect][2]=final_T[2];
                     }
              }
          }
      }
      if(auto_segm)
      {
          if(option==0) test_neigh(my_class, selected, &my_contor,2);
          else if(option==1) test_neigh(my_class, selected, &my_contor,1);
          else if(option==2) test_neigh(my_class, selected, &my_contor,0);
      }
#ifdef TRACE
      printf("my_contor = %d contor_trials = %d",my_contor,contor_trials);
#endif
      if(!auto_segm)
        if(test_same_cluster(rect,T))
        {
```

```
            delete [] my_class; delete [] sel_col; delete [] selected;
            continue;
        }
      if(auto_segm && my_contor<act_threshold)
        {
          delete [] my_class; delete [] sel_col; delete [] selected;
          if(contor_trials++<MAX_TRIAL) goto MS;
          else                          break;
        }

      if(auto_segm) my_actual(my_class);

      my_convert(selects, final_T, TI[rect]);
      for ( k = 0; k < _ro_col; k++ )
        if(my_class[k])
          gen_class[k]=rect;

      delete [] my_class; delete [] sel_col; delete [] selected;
    }
  n_rects=rect;
  Octet** isegment = new Octet*[3];
  register int j;
  for ( j = 0; j < 3; j++ )
    {
      isegment[j] = new Octet[_ro_col];
      memset( isegment[j], 0, _ro_col );
    }
  Octet *isegment0=isegment[0]; Octet *isegment1=isegment[1];
  Octet *isegment2=isegment[2];

  if(auto_segm)
    {
      Octet *my_class = new Octet[_ro_col];
      for ( k = 0; k < _ro_col; k++ )
         if(gen_class[k]==255) gen_class[k]=n_rects;

      Octet inv_map[Max_rects];
      for(k=0;k<n_rects;k++) inv_map[k]=k;

      if(option==0)  conn_comp(my_class,&n_rects,inv_map,T,10,0);
      else if(option==1) conn_comp(my_class,&n_rects,inv_map,T,20,0);
      else conn_comp(my_class,&n_rects,inv_map,T,act_threshold,0);
      optimize(T,n_rects);
      for(k=0;k<n_rects;k++) my_convert(selects, T[k], TI[k]);

      // Postprocessing
      if(option==1 || option ==2)
          for(k=2;k<10;k+=2) conn_comp(my_class,&n_rects,inv_map,T,k,1);

      end0 = mell_get_time();
      t = mell_time_to_sec(mell_time_diff(end0,begin0));

      register Octet my_max_ind;
      for ( k = 0; k < _ro_col; k++ )
        {
          if((my_max_ind=gen_class[k])!=n_rects)
          {
              int *ti=TI[my_max_ind];
            isegment0[k]=ti[0]; isegment1[k]=ti[1]; isegment2[k]=ti[2];
          }
      }
```

```
//Nostra aggiunta per la stampa a video dei cluster
      long numpixel = 0; int rr,gg,bb;
      cout <<"\n\n";
      for (k=0;k<n_rects;k++)
         {
                numpixel = 0;
            for (long kk=0;kk<_ro_col;kk++)
            {
                  if (gen_class[kk]==k)
            {
                        numpixel++;
                  rr = isegment0[kk];
                  gg = isegment1[kk];
                  bb = isegment2[kk];
            }
            }
            printf ("Cluster %3d: r=%3d; g=%3d; b=%3d; number of
pixels=%6ld\n",k+1,rr,gg,bb,numpixel);

             }
//FINE aggiunta

//Stampa dei tempi
printf("\n");

printf("Tempo impiegato     : %10.4f s\n",t);
printf("Throughput          : %10.0f pixel/s\n",_ro_col/t);
printf("frames/sec (300x300): %10.6f frames/s\n",_ro_col/t/300/300);

printf("\n");
//Fine stampa tempi


      // Save the borders
      memset(my_class,255,_ro_col);
      for ( k = 0; k < _ro_col; k++ )
      {
          int pres_class=gen_class[k];
          int pos_colms=k%_colms;

          if(pos_colms==0 || pos_colms==(_colms-1) ||
             k<(_colms-1) || k>(_ro_col-_colms))
             my_class[k] = 0;
          else
             for(j=1;j<8;j+=2)
               {
                 int u=k+my_neigh[j];
                 if(u>=0 && u<_ro_col &&  (pres_class<gen_class[u]))
               {
                 my_class[k] = 0; break;
               }
              }
      }
      my_write_PGM_file( pgmfile, my_class,_rows,_colms);
      delete [] my_class;
    }
  else   // if not auto_segmentation
    {
      optimize(T,n_rects);
      for(k=0;k<n_rects;k++)
        my_convert(selects, T[k], TI[k]);
      register int temp_class;
      for (k=0;k<_ro_col;k++)
```

```
        {
            if((temp_class=gen_class[k])<n_rects)
            {
                 isegment0[k] = /*TI[temp_class][0];*/pic->chdata(0)[k];
               isegment1[k] = /*TI[temp_class][1];*/pic->chdata(1)[k];
               isegment2[k] = /*TI[temp_class][2];*/pic->chdata(2)[k];
             }
          }
      }
  if(auto_segm){
    delete [] _m_col_remain;
    delete [] _col_remain;
    delete [] gen_remain;
  }
  delete [] gen_class;
  delete [] _col_index;
  delete [] _col0; delete [] _col1; delete [] _col2;
  delete [] _col_all;

  XfRaster::Info  info;
  pic->raster_info(info);
  result_ras_ = new RasterIpChannels( info, 3, eDATA_OCTET,
                              isegment, true );
  cout << endl << "Original Colors: " << _n_colors << endl;
  cout << "Numero regioni: " << n_rects << endl;

  return true;
}
```

```
  SegmenterMS::sRectangle rects[Max_rects];

  istream*  fp;
  fp = new ifstream( argv[1], ios::in );
  if ( fp->fail() )
      {
        cerr << __FILE__ << '(' << __LINE__ << "):  cannot open file " <<
          argv[1] << endl;
          cerr << "Usage: segm image_name.ppm output_image.ppm"<< endl;
        exit(1);
      }
  ostrstream     cmd;

  long      selects = 0L;
  selects |= Lightness; selects |= Ustar; selects |= Vstar;
  int p_dim=3;

  assert( p_dim <= p_max );   // must be since all array dimensions are such
  Boolean       block = false;
  unsigned int    seed  = 29254088; //random # generator



  int       n_rect = 0;
  Boolean   done = false;

 strcpy(pgmfile,argv[2]);
 pgmfile[strlen(pgmfile)-2]='g';         //nostre robe modifica

 cout <<"\n\nAlgoritmo di segmentazione di Comaniciu e Meer.\n\n";

 cout <<"File di input  : "<<argv[1]<<"\n";
 cout <<"File di output : "<<argv[2]<<"\n";


/*  cout << "Select:"<< endl
       <<"      w = undersegmentation inside a window"<< endl
       <<"      u = undersegmentation" << endl
       <<"      o = oversegmentation" << endl
       <<"      q = quantization" << endl;
  */
  char received='u';

  while(1)
   {
    // cin >> received;
     if((received =='w') || (received=='W'))
       {
         option=2; break;
       }
     else if((received =='u') || (received=='U'))
       {
         option=2; done=true; break;
       }
     else if((received =='o') || (received=='O'))
       {
         option=1; done=true; break;
       }
     else if((received =='q') || (received=='Q'))
       {
         option=0; done=true; break;
       }
     else
```

```
       {
         cout << endl <<"Please type w, u, o or q !" << endl;
       }
    }
  #ifdef ORG_SIZE
      cmd << "ee " << argv[1] << " &" << ends;
  #else
      cmd << "ee -geometry 120x120 " << argv[1] << " &" << ends;
  #endif
      system( cmd.str() );

  while ( !done )
    {
      cout << "***********  This is the " << n_rect+1 <<
        (n_rect==0 ? "st" : (n_rect==1 ? "nd" : (n_rect==2 ? "rd" : "th")))
        << " Window!  ************" << endl;
      cout << "The Upper_Left Corner Coords:" << endl;
      int   x1, y1;
      cin >> x1 >> y1;
      cout << endl;
      if((x1 < 0) || (y1 < 0))
        {
        cout << "********* " << n_rect << " Windows totally *******" << endl;
        done = true;
        break;
      }
      cout << "The Width and Height of window:" << endl;
      int   d1, d2;
      cin >> d1 >> d2;
      cout << endl;
      if ( (d1 == 0) || (d2 == 0) )
      {
        cout << "********* " << n_rect << " Windows totally *******" << endl;
        done = true;
        break;
      }

      rects[n_rect].x = x1; rects[n_rect].y = y1;
      rects[n_rect].width = d1; rects[n_rect].height = d2;
      n_rect++;
    }
  RasterIpChannels*     signal = read_PPM_file( fp );
  ((ifstream* )fp)->close();

  SegmenterMS     segmenter;
  segmenter.ms_segment( signal, rects, n_rect, selects, seed, block);
  RasterIpChannels*     sig_result = segmenter.result_ras_;

  write_PPM_file( argv[2], sig_result );

  ostrstream      cmd1;
  ostrstream      cmd2;

#ifdef ORG_SIZE
      cmd1 << "ee "<<argv[2]<<" &" << ends;
  if(option && !n_rect)
      cmd2 << "ee "<<pgmfile<<" &" << ends;
#else
      cmd1 << "xv -geometry 120x120 result.ppm &" << ends;
  if(option && !n_rect)
      cmd2 << "xv -geometry 120x120 result.pgm &" << ends;

#endif
```

```
      system( cmd1.str() );
  if(option && !n_rect)
      system( cmd2.str() );

  delete signal;
  delete sig_result;
  return( 0 );
}

static void skip( istream* fp )
{
  char      c1, c2;
  if ( (c1 = fp->get() ) != '#' ) {
    fp->putback( c1 );
    return;
  }
  while ( c1 == '#' )
    {
      while ( (c2 = fp->get()) != '\n' )
      ;
      c1 = fp->get();
      if ( c1 != '#' )
      fp->putback( c1 );
    }
  return;
}

static RasterIpChannels* read_PPM_file( istream* fp )
{
  int c1 = fp->get();
  int c2 = fp->get();
  int c3 = fp->get();

// test point

  if ( c1 == 'P' && c3 == '\n' )
    {
      Octet**     datain = new Octet*[p_max];
      int   w, h, m;

      switch ( c2 ) {
      case '3':
      {
        skip( fp );
        *fp >> w >> h;
        int i;
        for ( i = 0; i < p_max; i++ ) {
          datain[i] = new Octet[w * h];
        }
        fp->get();
        skip( fp );
        *fp >> m;

          //need a test for # comments !!!
        for ( int j = 0, idx = 0; j < h; j++ ) {
          for ( i = 0; i < w; i++, idx++ ) {
            *fp >> c1 >> c2 >> c3;   //ASCII decimal values
            datain[0][idx] = c1;
            datain[1][idx] = c2;
            datain[2][idx] = c3;
          }
        }
      }
```

```
      break;

      case '6':
      {
        skip( fp );
        *fp >> w >> h;

        for ( int i = 0; i < p_max; i++ ) {
          datain[i] = new Octet[w * h];
        }
        fp->get();
        skip( fp );
        *fp >> m;

// test point
        fp->get();
        skip( fp );

          cout << "Dimensioni : " << w << " x " << h << endl;
//          cout << "Max Value  m: " << m << endl;
          Octet *temp_buf = new Octet[h*w*3];
        fp->read(temp_buf, h*w*3);

          Octet *temp0 = datain[0];
        Octet *temp1 = datain[1];
        Octet *temp2 = datain[2];

        for ( register int j = 0, idx = 0; j < h*w; j++) {
            temp0[j] = temp_buf[idx++];
            temp1[j] = temp_buf[idx++];
            temp2[j] = temp_buf[idx++];
          }
          delete [] temp_buf;
      }
      break;

      default:
      cerr << "File is not the PPM format." << endl;
      return NULL;
      }

      XfRaster::Info    info;
      info.rows = h;
      info.columns = w;
      info.origin_x = 0;
      info.origin_y = 0;
      return (new RasterIpChannels( info, p_max, eDATA_OCTET,
                           datain, true ) );
    }
  cerr << "File is not the PPM format." << endl;
  return NULL;
}

Boolean my_write_PGM_file( char* fname, Octet* signal,int _rows, int _colms )
{
  ofstream fp( fname, ios::out );
  if ( fp.fail() )
    return false;

  fp << "P5\n" << _colms << ' ' << _rows<< "\n255" << endl;
  fp.write(signal,_rows*_colms);
  fp.close( );
  return true;
```

```
}

Boolean write_PPM_file( char* fname, RasterIpChannels* signal )
{
  ofstream fp( fname, ios::out );
  if ( fp.fail() )
    return false;

  fp << "P6\n" << signal->columns_ << ' ' << signal->rows_ << "\n255" << endl;

  assert( signal->dtype_ == eDATA_OCTET );
  Octet *temp_buf = new Octet[signal->rows_*signal->columns_*3];
  Octet *temp0 = signal->chdata_[0];
  Octet *temp1 = signal->chdata_[1];
  Octet *temp2 = signal->chdata_[2];
  for ( register int j = 0, idx = 0; j < signal->rows_*signal->columns_; j++) {
     temp_buf[idx++]=temp0[j];  //red
     temp_buf[idx++]=temp1[j];  //green
     temp_buf[idx++]=temp2[j];  //blue
   }
  fp.write(temp_buf,signal->rows_*signal->columns_*3);
  delete [] temp_buf;

  fp.close( );
  return true;
}

// Class constructor
// The `data' may be taken away from the caller in order to
// avoid time-consuming copying of the data.  However,
// the caller has to give explicit permission for this.
RasterIpChannels::RasterIpChannels(
    const XfRaster::Info& info, const int n_channels,
    const DataType dtype, Octet* data[], const Boolean do_take
) {
    rows_ = info.rows;
    columns_ = info.columns;
    dtype_ = dtype;
    clipf_ = false;
    n_channels_ = n_channels;
    if (n_channels_ == 0) {
      n_channels_ = 1;
    }
    size_t size = (size_t)(rows_ * columns_);
    chdata_ = new Octet*[n_channels_];
    for (int channel = 0; channel < n_channels_; channel++) {
      if ( do_take == true ) {     // take over the data from the caller
          chdata_[channel] = (Octet* )data[channel];
          data[channel] = nil;
      } else {
          if ( dtype_ == eDATA_FLOAT )       size *= sizeof(float);
          chdata_[channel] = new Octet[size];
          if (data != nil && data[channel] != nil) {
            memmove( chdata_[channel], data[channel], size );
          } else {
            memset( chdata_[channel], 0, size );
          }
      }
    }
  delete [] data;
}

RasterIpChannels::~RasterIpChannels() {
```

```
    for (int channel = 0; channel < n_channels_; channel++) {
      if (chdata_[channel])   delete [] chdata_[channel];
      chdata_[channel] = nil;
    }
    delete [] chdata_;
}

// RANGE forces `a' to be in the range [b..c] (inclusive)
inline void RANGE( int& a, const int b, const int c )
{
  if ( a < b ) {
      a = b;
  } else if ( a > c ) {
      a = c;
  }
}

void RasterIpChannels::raster_info(Info& i) {
    i.rows = rows_;
    i.columns = columns_;
    i.origin_x = 0;
    i.origin_y = 0;
}

// Move floating point array to Octet array, i.e. to [0..255]
// The result is either scaled to the range [0..255] or
// clipped to this range, depending on the flag `clipf'.
// Note:  Handles only 1-Octet per pixel pictures
// (i.e. mono/pseudo color pictures)
Octet** RasterIpChannels::raster_float_to_Octet(
    RasterIpChannels& ras
) {
    assert( ras.dtype() == eDATA_FLOAT );

    float   maxv = -1.0e38;
    XfRaster::Info      info;
    ras.raster_info(info);
    size_t  size = (size_t)(info.rows * info.columns);

    Octet** data = ras.chdata();
    int                 channels = ras.n_channels();
    Octet** picRes = new Octet*[channels];
    int                 i;
    for ( i = 0; i < channels; i++ )
      picRes[i] = new Octet[ size ];

    if ( ras.clipf() == true ) {  // clip the values outside the range [0..255]
      int   p;
      for ( i = 0; i < channels; i++ ) {
          register float*           ptr1 = (float* )data;
          register Octet*     ptr2 = picRes[i];
          for ( register int off = 0; off < size; off++, ptr1++, ptr2++ ) {
            p = int(*ptr1);
            RANGE( p, 0, 255 );
            *ptr2 = Octet(p);
          }
      }
    } else {                // scale the values to the range [0..255]
      for ( i = 0; i < channels; i++ ) {
          float               minv = (float) 1.e38;
          float               maxv = (float) -1.e38;
          register float*           ptr1 = (float* ) data[i];
          register Octet*     ptr2 = picRes[i];
```

```
          register int  off;
          for ( off = 0; off < size; off++, ptr1++ ) {
            if ( *ptr1 < minv )   minv = *ptr1;
            if ( *ptr1 > maxv )   maxv = *ptr1;
          }
          ptr1 = (float* ) data[i];
          float   ratio = (float) 255.0 / (maxv - minv);
          for ( off = 0; off < size; off++, ptr1++, ptr2++ )
              *ptr2 = Octet( (*ptr1 - minv) * ratio );
      }
    }
    return ( picRes );
}
```

```
//Color Image Segmentation
//This is the implementation of the algorithm described in
//D. Comaniciu, P. Meer,
//Robust Analysis of Feature Spaces: Color Image Segmentation,
//http://www.caip.rutgers.edu/~meer/RIUL/PAPERS/feature.ps.gz
//appeared in Proceedings of CVPR'97, San Juan, Puerto Rico.
// ===========================================================================
// =====      Module: segm.cc
// ===== ------------------------------------------------------------ ======
// =====      Version 01   Date: 04/22/97
// ===== ------------------------------------------------------------ ======
// ===========================================================================
// =====      Written by Dorin Comaniciu
// =====      e-mail:  comanici@caip.rutgers.edu
// ===========================================================================
// Permission to use, copy, or modify this software and its documentation
// for educational and research purposes only is hereby granted without
// fee, provided that this copyright notice appear on all copies and
// related documentation.  For any other uses of this software, in original
// or modified form, including but not limited to distribution in whole
// or in part, specific prior permission must be obtained from
// the author(s).
//
// THE SOFTWARE IS PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND,
// EXPRESS, IMPLIED OR OTHERWISE, INCLUDING WITHOUT LIMITATION, ANY
// WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.
//
// IN NO EVENT SHALL RUTGERS UNIVERSITY BE LIABLE FOR ANY SPECIAL,
// INCIDENTAL, INDIRECT OR CONSEQUENTIAL DAMAGES OF ANY KIND, OR ANY
// DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
// WHETHER OR NOT ADVISED OF THE POSSIBILITY OF DAMAGE, AND ON ANY
// THEORY OF LIABILITY, ARISING OUT OF OR IN CONNECTION WITH THE USE
// OR PERFORMANCE OF THIS SOFTWARE.
// ===========================================================================
//

#include <math.h>
#include <stdlib.h>
#include <limits.h>
#include <fstream.h>
#include <strstream.h>
#include <memory.h>
#include "timer.h"
#include "segm.hh"


int option = 2;
char pgmfile[50];

static const int rect_gen_contor=3;

// Radius of the searching window
static float gen_RADIUS[3]={2, 3, 4};
static float rect_RADIUS[rect_gen_contor]={8, 6, 4};
static float fix_RADIUS[rect_gen_contor];
static float final_RADIUS;
static float RADIUS2;
static float RADIUS;

static float max_dist;

static int my_threshold[3]={50, 100, 400};
static int my_rap[3]={4, 2, 1};
```

```
static int act_threshold;

#define my_abs(a) ((a) > 0 ? (a): (-a))
#define SQRT2 1.4142
#define SQRT3 1.7321
static const float      BIG_NUM      =    1.0e+20;

// Coefficient matrix for xyz and rgb spaces
static const int    XYZ[3][3] = { { 4125, 3576, 1804 },
                                  { 2125, 7154,  721 },
                                  {  193, 1192, 9502 } };
static const float  RGB[3][3] = { { 3.2405, -1.5371, -0.4985 },
                                  {-0.9693,  1.8760,  0.0416 },
                                  { 0.0556, -0.2040,  1.0573 } };

// Constants for LUV transformation
static const float     Xn = 0.9505;
static const float     Yn = 1.0;
static const float     Zn = 1.0888;
static const float     Un_prime = 0.1978;
static const float     Vn_prime = 0.4683;
static const float     Lt = 0.008856;

// # of samples
static const int  Max_J      =      25;

// Limit of the number of failed trials
static const int       Max_trials    =  50;

// Defaults values for the parameters.
static const int  sam_max = 60;

// Few more trials at the end
static const int        MAX_TRIAL=10;

// Used in 3-D histogram computation
static const int FIRST_SIZE=262144; // 2^18
static const int SEC_SIZE=64;  // 2^6

// Make coordinate computation faster
// my_neigh is for auto_segm, my_neigh_r works with region
static int my_neigh[8];
static int my_neigh_r[8];


// Results
static int ORIG_COLORS;
static int SEGM_COLORS;



//Inizio funzioni misurazione tempo
mell_time mell_gettimeofday_get_time(void)
{
     struct timeval tv;
     gettimeofday(&tv, 0);
     return tv;
}

mell_time mell_gettimeofday_time_diff(mell_time t1, mell_time t2)
{
     mell_time diff;
```

```
     diff.tv_sec = t1.tv_sec - t2.tv_sec;
     diff.tv_usec = t1.tv_usec - t2.tv_usec;
     /* normalize */
     while (diff.tv_usec < 0)
       {
       diff.tv_usec += 1000000L;
       diff.tv_sec -= 1;
       }

     return diff;
}
//Fine funzioni misurazione tempo




Boolean my_write_PGM_file( char*, Octet*,int,int);
void    covariance_w(const int N, int M, const int p, int ** data,
                  int *w, float T[],  float C[p_max][p_max]);
void    mean_s(const int N, const int p, int J[], int **data, float T[]);
void    my_convert(int, float *, int *);
void    reverse_map(Octet *, Octet *, int *, Octet *, float T[][p_max]);

// Class constructor
SegmenterMS::SegmenterMS( )
{
  _p = 0;
  _p_ptr = 0;
  _rrows  = 0;
  _rcolms = 0;
  _data_all = nil;
  _data = nil;
  _NJ = Max_J;
}

// Class destructor
SegmenterMS::~SegmenterMS( )
{
  if ( _data_all ) {
    for ( register int i = 0; i < _p; i++ )
      if ( _data_all[i] )   delete [] _data_all[i];
    delete [] _data_all;
  }
  if ( _data ) {
    for ( register int i = 0; i < _p; i++ )
      if ( _data[i] )   delete [] _data[i];
    delete [] _data;
  }
}

// LUV (final_T[]) to RGB (TI[]) conversion
void my_convert(int selects, float *final_T, int *TI)
{
  // this condition is always true
  if ( selects & Lightness && selects & Ustar && selects & Vstar )
    {
      if(final_T[0]<0.1)
        {
        TI[0]=0;TI[1]=0;TI[2]=0;
       }
       else
       {
```

```
          float my_x, my_y, my_z;
        if(final_T[0]< 8.0)
            my_y = Yn * final_T[0] / 903.3;
          else
            my_y = Yn * pow((final_T[0] + 16.0) / 116.0, 3);

          float u_prime = final_T[1] / (13 * final_T[0]) + Un_prime;
          float v_prime = final_T[2] / (13 * final_T[0]) + Vn_prime;

          my_x = 9 * u_prime * my_y / (4 * v_prime);
          my_z = (12 - 3 * u_prime - 20 * v_prime) * my_y / (4 * v_prime);

          TI[0] =int((RGB[0][0]*my_x + RGB[0][1]*my_y + RGB[0][2]*my_z)*255.0);
          TI[1] =int((RGB[1][0]*my_x + RGB[1][1]*my_y + RGB[1][2]*my_z)*255.0);
          TI[2] =int((RGB[2][0]*my_x + RGB[2][1]*my_y + RGB[2][2]*my_z)*255.0);

          if(TI[0]>255) TI[0]=255;
          else if(TI[0]<0) TI[0]=0;

        if(TI[1]>255) TI[1]=255;
          else if(TI[1]<0) TI[1]=0;

        if(TI[2]>255) TI[2]=255;
        else if(TI[2]<0) TI[2]=0;
      }
    }
  else
    {
       TI[0]=(int)final_T[0];
       TI[1]=(int)final_T[1];
       TI[2]=(int)final_T[2];
    }
}

// RGB to LUV conversion
// To gain speed the conversion works on a table of colors (_col_RGB[])
// rather than on the whole image
void SegmenterMS::convert_RGB_LUV( RasterIpChannels* signal, long selects )
{
  int x, y, z, my_temp;

  float l_star, u_star, v_star;
  float u_prime, v_prime;
  register int temp_col, temp_index, temp_ind;
  register int j,k;

  int a00=XYZ[0][0], a01=XYZ[0][1], a02=XYZ[0][2];
  int a10=XYZ[1][0], a11=XYZ[1][1], a12=XYZ[1][2];
  int a20=XYZ[2][0], a21=XYZ[2][1], a22=XYZ[2][2];

  int *A00 = new int[MAXV]; int *A01 = new int[MAXV]; int *A02 = new int[MAXV];
  int *A10 = new int[MAXV]; int *A11 = new int[MAXV]; int *A12 = new int[MAXV];
  int *A20 = new int[MAXV]; int *A21 = new int[MAXV]; int *A22 = new int[MAXV];
  for(j=0; j<MAXV;j++)
    {
      A00[j]=a00*j; A01[j]=a01*j; A02[j]=a02*j;
      A10[j]=a10*j; A11[j]=a11*j; A12[j]=a12*j;
      A20[j]=a20*j; A21[j]=a21*j; A22[j]=a22*j;
    }

  float *my_pow = new float[MAXV];
  for(j=0; j<MAXV;j++)
    my_pow[j]= 116.0 * pow(j/255.0, 0.3333333) - 16;
```

```
Octet* temp_ch0 = signal->chdata(0);
Octet* temp_ch1 = signal->chdata(1);
Octet* temp_ch2 = signal->chdata(2);

int pp;
int *temp0, *temp1, *temp2;
pp = _p_ptr;
if ( selects & Lightness ) temp0 = _data_all[pp++];
if ( selects & Ustar ) temp1 = _data_all[pp++];
if ( selects & Vstar ) temp2 = _data_all[pp++];
_p_ptr=pp;

for ( j = 0; j < _n_colors; j++)
  {
      temp_col=_col_RGB[j];
    int R=temp_col>>16; int G=(temp_col>>8) & 255; int B=temp_col & 255;

      x = A00[R] + A01[G] + A02[B];
      y = A10[R] + A11[G] + A12[B];
      z = A20[R] + A21[G] + A22[B];

      float  tval = y / 2550000.0; //Yn==1
    if ( tval >  Lt)  l_star = my_pow[(int)(tval*255+0.5)];
      else  l_star = 903.3 * tval;

      my_temp = x + 15 * y + 3 * z;
    if(my_temp)
      {
        u_prime = (float)(x << 2) / (float)(my_temp);
        v_prime = (float)(9 * y) / (float)(my_temp);
      }
    else
      {
        u_prime=4.0;
        v_prime=9.0/15.0;
      }

    tval=13*l_star;
      u_star = tval* (u_prime - Un_prime); // Un_prime = 0.1978
      v_star = tval* (v_prime - Vn_prime); // Vn_prime = 0.4683

    _col0[j] = (int)(l_star+0.5);
    if(u_star>0) _col1[j] = (int)(u_star+0.5);
      else _col1[j] = (int)(u_star-0.5);

    if(v_star>0) _col2[j] = (int)(v_star+0.5);
      else _col2[j] = (int)(v_star-0.5);
  }
for(j=0;j<_ro_col;j++)
  {
    temp_col=(((((int)temp_ch0[j])<<8)+(int)temp_ch1[j])<<8)+(int)temp_ch2[j];
    temp_ind=_col_misc[temp_col>>6];
    for(k=temp_ind;k<temp_ind+SEC_SIZE;k++)
    if(_col_RGB[k]==temp_col)
      {
         temp_index=_col_index[j]=k;
         break;
     }
    temp0[j]=_col0[temp_index];
    temp1[j]=_col1[temp_index];
    temp2[j]=_col2[temp_index];
  }
```

```
  delete [] my_pow;
  delete [] A22; delete [] A21; delete [] A20;
  delete [] A12; delete [] A11; delete [] A10;
  delete [] A02; delete [] A01; delete [] A00;
  delete [] _col_misc;
  delete [] _col_RGB;
}

// 3-D Histogram computation
// Implement a trade-off between speed and required memory
void SegmenterMS::my_histogram(RasterIpChannels* signal, long selects)
{
  int *first_tab= new int[FIRST_SIZE];
  _col_misc= new int[FIRST_SIZE];
  int **third_tab;
  int *fourth_tab;
  int *fifth_tab=new int[SEC_SIZE];
  _n_colors=0;

  register int k,j,p,r;
  int temp_ind, sec_ind, third_ind;

  int first_contor=0, third_contor=0;

  memset(first_tab,0,sizeof(int)*FIRST_SIZE);
  memset(_col_misc,0,sizeof(int)*FIRST_SIZE);

  register Octet* ch0 = signal->chdata(0);
  register Octet* ch1 = signal->chdata(1);
  register Octet* ch2 = signal->chdata(2);

  //first_tab -> how many
  for(k=0;k<_ro_col;k++)
    {
      temp_ind=(((ch0[k]<<8)+ch1[k])<<2)+(ch2[k]>>6);
      first_tab[temp_ind]++;
    }
  //_col_misc -> memo position
  for(k=0;k<FIRST_SIZE;k++)
    if(first_tab[k])
      {
        _col_misc[k]=first_contor;
      first_contor++;
      }
  //contors
  fourth_tab=new int[first_contor];
  memset(fourth_tab,0,sizeof(int)*first_contor);
  //tab of pointers to reduced colors
  third_tab=new int *[first_contor];
  first_contor=0;
  for(k=0;k<FIRST_SIZE;k++)
    if(first_tab[k])
      {
        third_tab[first_contor]=new int[first_tab[k]];
      first_contor++;
      }

  for(k=0;k<_ro_col;k++)
    {
      temp_ind=(((ch0[k]<<8)+ch1[k])<<2)+(ch2[k]>>6);
      sec_ind=ch2[k] & 63;
      third_ind=_col_misc[temp_ind];
      third_tab[third_ind][fourth_tab[third_ind]]=sec_ind;
```

```
      fourth_tab[third_ind]++;
    }
  for(k=0;k<first_contor;k++)
    {
      memset(fifth_tab,0,sizeof(int)*SEC_SIZE);
      for(j=0;j<fourth_tab[k];j++)
        fifth_tab[third_tab[k][j]]++;
      for(j=0;j<SEC_SIZE;j++)
        if(fifth_tab[j])
         _n_colors++;
      }
  _col_RGB=new int[_n_colors];
  _m_colors=new int[_n_colors];

  k=0;p=0;
  for(r=0;r<FIRST_SIZE;r++)
    if(first_tab[r])
     {
       memset(fifth_tab,0,sizeof(int)*SEC_SIZE);
       for(j=0;j<fourth_tab[k];j++)
         fifth_tab[third_tab[k][j]]++;
       _col_misc[r]=p;
       for(j=0;j<SEC_SIZE;j++)
         if(fifth_tab[j])
         {
           _col_RGB[p]=(r<<6)+j;
           _m_colors[p]=fifth_tab[j];
             p++;
         }
       delete [] third_tab[k];
       k++;
     }
  delete [] third_tab;
  delete [] fourth_tab;
  delete [] fifth_tab;
  delete [] first_tab;

  _col_all = new int*[3];
  _col0=_col_all[0] = new int[_n_colors];
  _col1=_col_all[1] = new int[_n_colors];
  _col2=_col_all[2] = new int[_n_colors];
  _col_index = new int[_ro_col];
}

// Update _col_remain[], _m_col_remain, and _n_col_remain
void SegmenterMS::my_actual(Octet *my_class)
{
  register int i;
  int temp_contor=n_remain;
  register int *temp_rem= new int[_ro_col];
  memcpy(temp_rem,gen_remain,sizeof(int)*temp_contor);
  n_remain=0;
  for(i=0;i<temp_contor;i++)
    if(!my_class[temp_rem[i]])
      gen_remain[n_remain++]=temp_rem[i];
  delete [] temp_rem;
  memset(_col_remain,0,sizeof(int)*_n_col_remain);
  memset(_m_col_remain,0,sizeof(int)*_n_colors);
  _n_col_remain=0;
  for(i=0;i<n_remain;i++)
    _m_col_remain[_col_index[gen_remain[i]]]++;
  for(i=0;i<_n_colors;i++)
     if(_m_col_remain[i])
```

```
       {
       _col_remain[_n_col_remain]=i;
         _n_col_remain++;
       }
}

// if more than "how_many"  neighbors, consider the point
void SegmenterMS::test_neigh(Octet* my_class, int *selected, int* my_contor, int
how_many)
{
  register int i,j,p,k;
  register Octet* local_class=my_class;
  register int temp_contor=*my_contor;
  register int my_index;
  if(auto_segm) my_index=n_remain;
  else          my_index=_ro_col;
  for ( p = 0, i; p < my_index; p++ )
    {
      if(auto_segm) i=gen_remain[p];
      else          i=p;
      if(!local_class[i])
      {
          int neigh_contor=0, no_neigh=1;
          for(j=0;j<8;j++)
            {
              k=i+my_neigh[j];
              if(k>=0 && k<_ro_col && local_class[k])
              {
                  if(auto_segm  && gen_class[k]!=255) continue;
                neigh_contor++;
                  if(neigh_contor>how_many)
                {
                  no_neigh=0;
                      break;
                }
              }
            }
        if(!no_neigh)
        {
            if(auto_segm) selected[*my_contor]=i;
          *my_contor=*my_contor+1;
        }
      }
    }
  for(i=temp_contor;i<*my_contor;i++)
    local_class[selected[i]]=1;
}


// Find the feature vectors inside the given window
// Use Improved Absolute Error Inequality Criterion
// when computing Euclidean distance
// See J.S.Pan el al, Fast Clustering Alg. for VQ, Pattern Recognition,
// Vol. 29, No. 3, pp. 511-518, 1996

void SegmenterMS::new_auto_loop(float *final_T, Octet *sel_col)
{
  float L,U,V,RAD2,R;
  register int TT0=0, TT1=0, TT2=0;
  register int local_contor=0;
  float final_T0=final_T[0], final_T1=final_T[1], final_T2=final_T[2];
  float RADIUS_S2=SQRT2*RADIUS, RADIUS_S3=SQRT3*RADIUS;
```

```
  for ( register int p = 0, k; p < _n_col_remain; p++ )
        {
            k=_col_remain[p];
              L=_col0[k]-final_T0; if((L=my_abs(L))>=RADIUS) continue;
              U=_col1[k]-final_T1; if((R=my_abs(U)+L)>=RADIUS_S2) continue;
            V=_col2[k]-final_T2; if(R+my_abs(V)>=RADIUS_S3) continue;
              RAD2=L*L+U*U+V*V;
              if(RAD2<RADIUS2)
            {
                  register int r=_m_col_remain[k];
                  TT0+=_col0[k]*r; TT1+=_col1[k]*r; TT2+=_col2[k]*r;
              local_contor+=r;
                  sel_col[k]=1;
            }
         }
  final_T[0]=(float)TT0/(float)local_contor;
  final_T[1]=(float)TT1/(float)local_contor;
  final_T[2]=(float)TT2/(float)local_contor;
}

// The same as above, but for non auto_segmentation
void SegmenterMS::nauto_loop(float *final_T, int *selected,
                             Octet *my_class, int *my_contor)
{
  float L,U,V,RAD2,R;
  register int local_contor=*my_contor;
  float final_T0=final_T[0], final_T1=final_T[1], final_T2=final_T[2];
  float RADIUS_S2=SQRT2*RADIUS, RADIUS_S3=SQRT3*RADIUS;

  for ( register int k = 0; k < _n_points; k++ )
          {
              L=_data[0][k]-final_T0; if((L=my_abs(L))>=RADIUS)  continue;
              U=_data[1][k]-final_T1; if((R=my_abs(U)+L)>=RADIUS_S2)  continue;
            V=_data[2][k]-final_T2; if(R+my_abs(V)>=RADIUS_S3)  continue;
              RAD2=L*L+U*U+V*V;
              if(RAD2<RADIUS2)
            {
              selected[local_contor++]=k;
                  my_class[k]=1;
            }
          }
  *my_contor=local_contor;
}

// Set the Radius of the window
void set_RADIUS(int gen_gen_contor, int final)
{
  if(final==2)      RADIUS=final_RADIUS*1.26;
  else if(final==1) RADIUS=final_RADIUS;
  else              RADIUS=fix_RADIUS[gen_gen_contor];
  RADIUS2=RADIUS*RADIUS;
}

// Test if the clusters have the same mean
int test_same_cluster(int rect, float T[][p_max])
{
  float L,U,V,RAD2;
  for(register int k=0; k<rect;k++)
    {
      L=T[k][0]-T[rect][0]; U=T[k][1]-T[rect][1]; V=T[k][2]-T[rect][2];
      RAD2=L*L+U*U+V*V;
      if(RAD2<1)
        return 1;
```

```
    }
  return 0;
}

// First take only pixels inside the search windows at their final locations
// Then inflate windows to double volume and retain only pixels which are
// neighbors with the previous
void SegmenterMS::get_codeblock1(float T[][p_max], int n_rects)
{
  float L,U,V,RAD2, R, min_RAD2;
  int min_ind;
  register int i,k,u;
  register int pres_class, my_flag;
  register float *ptr;

  if(auto_segm) set_RADIUS(0,0);
  else          set_RADIUS(2,0);

  for(k=0;k<_ro_col;k++)
    {
      min_RAD2=BIG_NUM; min_ind=0;
      for(i=0;i<n_rects;i++)
      {
          ptr=T[i];
          L=_data0[k]-ptr[0]; if(my_abs(L)>=RADIUS)  continue;
          U=_data1[k]-ptr[1]; if(my_abs(U)>=RADIUS)  continue;
        V=_data2[k]-ptr[2]; if(my_abs(V)>=RADIUS)  continue;
          RAD2=L*L+U*U+V*V;
          if(RAD2<min_RAD2)
          {
            min_RAD2=RAD2;
            min_ind=i;
          }
      }
      if(min_RAD2<RADIUS2) gen_class[k]=min_ind;
      else                 gen_class[k]=n_rects;
    }

  if(auto_segm) set_RADIUS(0,1);
  else          set_RADIUS(0,0);

  for(k=0;k<_ro_col;k++)
    if(gen_class[k]==n_rects)
      for(i=0;i<8;i++)
      {
          u=k+my_neigh[i];
          if(u>=0 && u<_ro_col)
            if((pres_class=gen_class[u])!=n_rects)
            {
                ptr=T[pres_class];
                L=_data0[k]-ptr[0]; if(my_abs(L)>=RADIUS)  continue;
                U=_data1[k]-ptr[1]; if(my_abs(U)>=RADIUS)  continue;
                V=_data2[k]-ptr[2]; if(my_abs(V)>=RADIUS)  continue;
                RAD2=L*L+U*U+V*V;
              if(RAD2<RADIUS2) gen_class[k]=pres_class;
            }
        }
}

// Final allocation
void SegmenterMS::get_codeblock(float T[][p_max], int n_rects)
{
  float L,U,V,RAD2, min_RAD2;
```

```
  register int min_ind;
  register int i,k;
  register float *ptr;

  for(k=0;k<_ro_col;k++)
    {
      min_RAD2=BIG_NUM;
      min_ind=0;
      for(i=0;i<n_rects;i++)
      {
          ptr=T[i];
        L=_data0[k]-ptr[0]; U=_data1[k]-ptr[1]; V=_data2[k]-ptr[2];
          RAD2=L*L+U*U+V*V;
          if(RAD2<min_RAD2)
          {
            min_RAD2=RAD2;
            min_ind=i;
          }
      }
      gen_class[k]=min_ind;
     }
}

// Compute the mean of feature vectors mapped into the same color
void SegmenterMS::new_codebook(float T[][p_max], int n_rects)
{
  register int i,k;
  register int *tab_contor = new int[n_rects];
  register int prez_class;
  register float *ptr;

  memset(tab_contor,0,sizeof(int)*n_rects);
  for(i=0;i<n_rects;i++)
    {
      T[i][0]=0.0; T[i][1]=0.0; T[i][2]=0.0;
    }
  for(k=0;k<_ro_col;k++)
   if((prez_class=gen_class[k])!=n_rects)
    {
      ptr=T[prez_class];
      ptr[0]+=_data0[k]; ptr[1]+=_data1[k]; ptr[2]+=_data2[k];
      tab_contor[prez_class]++;
    }
   for(i=0;i<n_rects;i++)
    {
      T[i][0]/=tab_contor[i]; T[i][1]/=tab_contor[i]; T[i][2]/=tab_contor[i];
    }
  delete [] tab_contor;
}

// Determine the final feature palette
void SegmenterMS::optimize(float T[][p_max], int n_rects)
{
  get_codeblock1(T,n_rects);
  new_codebook(T,n_rects);
  if(auto_segm)
    get_codeblock(T,n_rects);
 // cerr<<":";
}


// Inverse of the mapping array used in color elimination
```

```
void reverse_map(Octet *inv_map, Octet *my_map, int *n_rects, Octet
*valid_class, float T[][p_max])
{
  float sec_T[Max_rects][p_max];
  register int u=0, k, j;
  for(j=0;j<*n_rects;j++)
    {
      if(valid_class[j])
      {
          for(k=0;k<3;k++)
            sec_T[u][k]=T[j][k];
          my_map[j]=u;
          inv_map[u]=j;
          u++;
        }
    }
  my_map[*n_rects]=u;
  inv_map[u]=*n_rects;
  *n_rects=u;
  for(j=0;j<*n_rects;j++)
    for(k=0;k<3;k++)
      T[j][k]=sec_T[j][k];
}

// Eliminate colors that have less than "my_lim" connected pixels
void SegmenterMS::eliminate_class(Octet *my_class,int *my_max_region, int
*n_rects,int my_lim, Octet* inv_map, float T[][p_max], REGION *first_region)
{
  register int j, k;
  register Octet *valid_class;
  register REGION *current_region=first_region;

  valid_class=new Octet[*n_rects];
  for(j=0;j<*n_rects;j++)
    {
      if(my_max_region[j]<my_lim) valid_class[j]=0;
      else                        valid_class[j]=1;
    }
  while(1)
    {
      if((current_region->my_class<*n_rects &&
          !valid_class[current_region->my_class]))
        for(k=0;k<current_region->my_contor;k++)
           gen_class[current_region->my_region[k]]=*n_rects;
      if(current_region->next_region_str)
        current_region=current_region->next_region_str;
      else break;
    }
  Octet my_map[Max_rects];
  reverse_map(inv_map,my_map,n_rects,valid_class,T);
  for(k=0;k<_ro_col;k++)
    my_class[k]=my_map[gen_class[k]];
  delete [] valid_class;
  memcpy(gen_class,my_class,_ro_col);
}

// Eliminate regions with less than "my_lim" pixels
void SegmenterMS::eliminate_region(int *n_rects,int my_lim, float T[][p_max],
REGION* first_region)
{
  register int j,u,k,p, pres_class, min_ind;
  register REGION *current_region=first_region;
  register int* region;
```

```
  float *ptr;
  float L,U,V,RAD2,minRAD2;
  int increm;

  while(1)
    {
      if(current_region->my_contor<my_lim)
      {
          set_RADIUS(0,0); increm=4;
        region=current_region->my_region;
          for(k=0;k<current_region->my_contor;k++)
            gen_class[region[k]]=*n_rects;
        while(1)
            {
              Boolean my_flag=0;
              RADIUS+=increm; RADIUS2=RADIUS*RADIUS; increm+=4;
              for(k=0;k<current_region->my_contor;k++)
              if(gen_class[p=region[k]]==(*n_rects))
                {
                  minRAD2=RADIUS2;
                    for(j=1;j<8;j+=2)
                      {
                        u=p+my_neigh[j];
                        if(u>=0 && u<_ro_col)
                        if((pres_class=gen_class[u])!=(*n_rects))
                      {
                            ptr=T[pres_class];
                        L=_data0[p]-ptr[0]; U=_data1[p]-ptr[1];
                            V=_data2[p]-ptr[2]; RAD2=L*L+U*U+V*V;
                        if(RAD2<minRAD2)
                          {
                                minRAD2=RAD2; min_ind=pres_class;
                        }
                      }
                    }
                 if(minRAD2<RADIUS2) gen_class[p]=min_ind;
                   my_flag=1;
                }
             if(!my_flag) break;
          }
      }
      if(current_region->next_region_str)
        current_region=current_region->next_region_str;
      else break;
    }
}

// Destroy the region list
void SegmenterMS::destroy_region_list(REGION *first_region)
{
  register REGION *current_region=first_region;
  while(1)
    {
      delete [] current_region->my_region;
      first_region=current_region;
      if(current_region->next_region_str)
        {
          current_region=current_region->next_region_str;
          delete first_region;
      }
      else
        {
          delete first_region;
```

```
          break;
      }
    }
}

// Connected component main routine
void SegmenterMS::find_other_neigh(int k, int *my_ptr_tab, REGION
*current_region)
{
  register int *ptr_tab=my_ptr_tab;
  register int i,u, j=k, sec_signal;
  register int contor=0;
  register int region_contor=current_region->my_contor;
  register int region_class=current_region->my_class;
  ptr_tab[contor]=j;

  while(1)
    {
      sec_signal=0;
      for(i=1;i<9;i+=2)
      {
        u=j+my_neigh[i];
          if(u>=0 && u<_ro_col)
          if(gen_class[u]==region_class && !taken[u])
          {
              sec_signal=1;
            conn_selected[region_contor++]=u;
            taken[u]=1;
              ptr_tab[++contor]=u;
            }
      }
      if(sec_signal) j=ptr_tab[contor];
      else
       {
           if(contor>1) j=ptr_tab[--contor];
         else break;
       }
    }
  current_region->my_contor=region_contor;
}

// Create the region list
REGION *SegmenterMS::create_region_list(int *my_max_region, int change_type)
{
  register int k, local_label=0;
  register REGION *first_region, *prev_region, *current_region;
  taken = new Octet[_ro_col];
  memset(taken,0,_ro_col);
  conn_selected = new int[_ro_col];
  int *ptr_tab=new int[_ro_col];

  for(k=0;k<_ro_col;k++)
    if(!taken[k])
      {
        current_region=new REGION;
        current_region->my_contor=0;
        current_region->my_class=gen_class[k];
        current_region->my_label=local_label;
        if(k!=0) prev_region->next_region_str=current_region;
      if(k==0){ first_region=current_region;}

      local_label++;
      conn_selected[current_region->my_contor++]=k;
```

```
        taken[k]=1;
        find_other_neigh(k,ptr_tab,current_region);
        if(change_type==0)
        if(my_max_region[current_region->my_class]<current_region->my_contor)
            my_max_region[current_region->my_class]=current_region->my_contor;
        current_region->my_region=new int[current_region->my_contor];

        memcpy(current_region-
>my_region,conn_selected,sizeof(int)*current_region->my_contor);
        prev_region=current_region;
       }
  current_region->next_region_str=0;

  delete [] ptr_tab; delete [] taken; delete [] conn_selected;
  return first_region;
}

// Find connected components and remove small regions of classes
// with small regions
void SegmenterMS::conn_comp(Octet *my_class, int *n_rects, Octet *inv_map, float
T[][p_max],int my_lim, int change_type)
{
  REGION *first_region;
  int *my_max_region;
  if(change_type==0)
    {
      my_max_region = new int[(*n_rects)+1];
      memset(my_max_region,0,sizeof(int)*((*n_rects)+1));
    }
  first_region=create_region_list(my_max_region, change_type);
  if(change_type==0)  //elliminate classes with small regions

eliminate_class(my_class,my_max_region,n_rects,my_lim,inv_map,T,first_region);
  else if(change_type==1)  //elliminate small regions
    eliminate_region(n_rects,my_lim,T,first_region);
  destroy_region_list(first_region);
  if(change_type==0) delete [] my_max_region;
 // cerr<<":";
}


// Cut a rectangle from the entire input data
// Deletes the previous rectangle, if any
void SegmenterMS::cut_rectangle( sRectangle* rect )
{
  if ( _data ) {
    for ( register int i = 0; i < _p; i++ )
      if ( _data[i] )   delete [] _data[i];
    delete [] _data;
  }

  // Set the dimensions of the currently processed region.
  _rrows  = rect->height;
  _rcolms = rect->width;
  _data = new int*[_p];

  register int my_x = rect->x;
  register int my_y = rect->y;
  register int i, j, d;
  for ( i = 0; i < _p; i++ )
      _data[i] = new int[_rcolms*_rrows];

  if(auto_segm)
```

```
    for ( d = 0; d < _p; d++ )
      memcpy(_data[d], _data_all[d],sizeof(int)*_ro_col);
  else
    {
      int   idx1 = my_y * _colms + my_x;
      int   idx2 = 0;
      for ( j = my_y, d;
            j < my_y + _rrows; j++, idx1 += _colms - _rcolms )
        for ( i = my_x; i < my_x + _rcolms; i++, idx1++, idx2++ )
          {
            for ( d = 0; d < _p; d++ )
            _data[d][idx2] = _data_all[d][idx1];
          }
    }
  //cerr<<":";
}

// Compute the mean of N points given by J[]
void mean_s(const int N, const int p, int J[], int **data, float T[])
{
  int TT[p_max];
  register int k, i, j;
  for ( i = 0; i < p; i++ )
    TT[i] = 0;
  for ( i = 0; i < N; i++ )
    {
      k = J[i];
      for ( j = 0; j < p; j++ )
        TT[j] += data[j][k];
     }
   for ( i = 0; i < p; i++ )
      T[i] = (float)TT[i] / (float)N;
}

// Build a subsample set of 9 points
int SegmenterMS::subsample(float *Xmean )
{
  int J[9];
  register int my_contor=0, uj, i0;
  if(auto_segm)
  {
    i0=J[my_contor]=
       gen_remain[int(float(n_remain)*float(rand())/float(RAND_MAX))];
  }
  else
  {
    i0=J[my_contor]=int(float(_n_points)*float(rand())/float(RAND_MAX));
  }

  my_contor++;

  for(register i=0;i<8;i++){
    uj=i0 + my_neigh_r[i];
    if(uj>=0 && uj<_n_points)
      {
        if((auto_segm && gen_class[uj]!=255)) break;
      else
          {
            J[my_contor] = uj;
            my_contor++;
        }
      }
    }
```

```
  mean_s(my_contor, _p, J, _data, Xmean);
  return 1;
}

// Sampling routine with all needed tests
float SegmenterMS::my_sampling( int rect, float T[Max_rects][p_max])
{
  register int  k, c;
  register float L,U,V,Res;
  register float my_dist=max_dist, my_sqrt_dist=fix_RADIUS[0];
  float     TJ[Max_J][p_max];
  int l =  0;       //contor of number of subsample sets
  int ll = 0;       //contor of trials
  float     Xmean[p_max];
  float     Obj_fct[Max_J];

  //Max_trials = max number of failed trials
  //_NJ = max number of subsample sets

  while ( (ll < Max_trials) && (l < _NJ ) )
    {
      if ( subsample(Xmean) )    // the subsample procedure succeeded
      {
            ll = 0; c=0;

            // Save the mean
            for ( k = 0; k < _p; k++ ) TJ[l][k] = Xmean[k];

            // Compute the square residuals (Euclid dist.)
            if(auto_segm)
                  for ( register int p = 0; p < _n_col_remain; p++ )
                {
                 k=_col_remain[p];
                      L=_col0[k]-Xmean[0]; if(my_abs(L)>=my_sqrt_dist) continue;
                 U=_col1[k]-Xmean[1]; if(my_abs(U)>=my_sqrt_dist) continue;
                 V=_col2[k]-Xmean[2]; if(my_abs(V)>=my_sqrt_dist) continue;
                 if(L*L+U*U+V*V<my_dist) c+=_m_col_remain[k];
                }
            else
                for ( k = 0; k < _n_points; k++ )
                {
                      L=_data[0][k]-Xmean[0];
                      if(my_abs(L)>=my_sqrt_dist) continue;
                 U=_data[1][k]-Xmean[1];
                      if(my_abs(U)>=my_sqrt_dist) continue;
                 V=_data[2][k]-Xmean[2];
                      if(my_abs(V)>=my_sqrt_dist) continue;
                      if(L*L+U*U+V*V<my_dist) c++;
                }

              //  Objective functions
            Obj_fct[l]=c;
              l++;
          }
      else ++ll;
    }
  if ( ll == Max_trials && l < 1) return( BIG_NUM ); // Cannot find a kernel

  //  Choose the highest density
  L = -BIG_NUM; c=0;
  for ( k = 0; k < _NJ; k++ )
      if ( Obj_fct[k] > L)
```

```
      {
        L = Obj_fct[k];
        c = k;
      }
  if(Obj_fct[c]>0)
    for(k=0;k<_p;k++)
       T[rect][k]=TJ[c][k];
  else return -BIG_NUM; // Not enough points
  return ( 0 );
}

// Compute the weighted covariance of N points
void covariance_w(const int N, int M, const int p,  int **data,
               int *w, float T[], float C[p_max][p_max])
{
  register int i, j, k, l;
  int TT[p_max];
  for ( i = 0; i < p; i++ )
    TT[i] = 0;
  for ( i = 0; i < M; i++ )
    for ( j = 0; j < p; j++ )
      TT[j] += w[i]*data[j][i];
  for ( i = 0; i < p; i++ )
      T[i] = (float) TT[i] / (float)N;

  for ( i = 0; i < p; i++ )
    for ( j = i; j < p; j++ )
      C[i][j] = 0.0;
  for ( i = 0; i < M; i++ )
    {
      for ( k = 0; k < p; k++ )
      for ( l = k; l < p; l++ )
         C[k][l]+=w[i]*(data[k][i]-T[k])*(data[l][i]-T[l]);
    }
  for ( k = 0; k < p; k++ )
    {
    for ( l = k; l < p; l++ )
      C[k][l] /= (float)(N-1);
    for ( l = 0; l < k; l++ )
      C[k][l] = C[l][k];
    }
}

// initialization
void SegmenterMS::init_neigh(void)
{
  my_neigh[0]= -_colms-1;  my_neigh[1]= -_colms;
  my_neigh[2]= -_colms+1;  my_neigh[3]= +1;
  my_neigh[4]= +_colms+1;  my_neigh[5]= +_colms;
  my_neigh[6]= +_colms-1;  my_neigh[7]= -1;

  my_neigh_r[0]= -_rcolms-1;  my_neigh_r[1]= -_rcolms;
  my_neigh_r[2]= -_rcolms+1;  my_neigh_r[3]= +1;
  my_neigh_r[4]= +_rcolms+1;  my_neigh_r[5]= +_rcolms;
  my_neigh_r[6]= +_rcolms-1;  my_neigh_r[7]= -1;
}

// Init matrices parameters
void SegmenterMS::init_matr(void)
{
  // General statistic parameters for X.
  float     Mg[p_max];         //sample mean of X
  float     C[p_max][p_max];   //sample covariance matrix of X
```

```
  covariance_w(_ro_col, _n_colors, _p, _col_all, _m_colors, Mg, C);

  // Adaptation
  float my_th=C[0][0]+C[1][1]+C[2][2];
  int active_gen_contor=1;

  if(auto_segm)
     fix_RADIUS[0]=gen_RADIUS[option]*sqrt(my_th/100);
  else
    {
      active_gen_contor=rect_gen_contor;
      for(int i=0;i<active_gen_contor;i++)
        fix_RADIUS[i]=rect_RADIUS[i]*sqrt(my_th/100);
    }
  final_RADIUS=fix_RADIUS[active_gen_contor-1]*1.26;
  max_dist=fix_RADIUS[0]*fix_RADIUS[0];
#ifdef TRACE
  printf("\n %.2f %.2f ", fix_RADIUS[0], final_RADIUS);
#endif
  act_threshold=(int)((my_threshold[option]>sqrt(_ro_col)/my_rap[option])?
                 my_threshold[option]:sqrt(_ro_col)/my_rap[option]);
 // cerr<<":";
}

// Init
void SegmenterMS::initializations(RasterIpChannels* pic, sRectangle rects[], int
*n_rects, long selects, int *active_gen_contor)
{
  register int i;
  XfRaster::Info        info;
  pic->raster_info(info);
  _colms = info.columns; _rows  = info.rows; _ro_col = _rows * _colms;

  _data_all = new int*[_p];
  for ( i = 0; i < _p; i++ )
      _data_all[i] = new int[_ro_col];
  _data0=_data_all[0]; _data1=_data_all[1]; _data2=_data_all[2];
  init_neigh();
  my_histogram(pic, selects);
  convert_RGB_LUV( pic, selects );
  gen_class = new Octet[_ro_col];
  memset(gen_class,255,_ro_col);
  if(!(*n_rects))
    {
      auto_segm=1;
      *n_rects=Max_rects;
      n_remain=_ro_col;
      _n_col_remain=_n_colors;
      gen_remain = new int[_ro_col];
      _col_remain = new int[_n_colors];
      _m_col_remain = new int[_n_colors];
      for ( i = 0; i< _ro_col ; i++ )
        gen_remain[i] = i;
      for ( i = 0; i< _n_colors ; i++ )
        _col_remain[i] = i;
      memcpy(_m_col_remain,_m_colors,sizeof(int)*_n_colors);

      for ( i = 0; i < Max_rects ; i++ )
        {
          rects[i].width=_colms; rects[i].height=_rows;
          rects[i].x = 0;        rects[i].y = 0;
        }
      *active_gen_contor=1;
```

```
    }
  else
    {
      auto_segm=0;
      n_remain=0;
      *active_gen_contor=rect_gen_contor;
      option=2;
    }
  init_matr();
  delete [] _m_colors;
 }

// Mean shift segmentation applied on the selected region(s) of an image or
// on the whole image
Boolean SegmenterMS::ms_segment( RasterIpChannels* pic, sRectangle rects[],
                            int n_rects, long selects ,
                                    unsigned int seed_default, Boolean block)
{
      mell_time end0,begin0;
      double t;
      begin0=mell_get_time();

  if (n_rects > Max_rects) return false;
  if ( selects & Lightness || selects & Ustar || selects & Vstar )
    _p=3;
  else return false;

  int contor_trials=0, active_gen_contor;
  float     T[Max_rects][p_max];
  int TI[Max_rects][p_max];
  float L,U,V,RAD2,q;
  register int i,k;

  srand( seed_default ); //cerr<<":";
  initializations(pic, rects, &n_rects, selects, &active_gen_contor);

  // Mean shift algorithm and removal of the detected feature
  int rect;
  //IL PROBLEMA è in questo ciclo for
  for ( rect = 0; rect < n_rects; rect++ )
    {
      _n_points = rects[rect].width * rects[rect].height;
      cut_rectangle( &rects[rect] );
    MS:
      if(auto_segm && contor_trials) _NJ=1;

      q =  my_sampling( rect, T);

      if(q == -BIG_NUM || q == BIG_NUM)
        if(contor_trials++<MAX_TRIAL)  goto MS;
        else                           break;


      float final_T[p_max];
      for ( i = 0; i < _p; i++ ) final_T[i]=T[rect][i];

      int *selected = new int[_ro_col];
      Octet *sel_col = new Octet[_n_colors];
      Octet *my_class = new Octet[_ro_col];

      int my_contor=0, gen_gen_contor=-1, how_many=10;
      while(gen_gen_contor++<active_gen_contor-1)
        {
```

```
        set_RADIUS(gen_gen_contor, 0);
      int gen_contor=0;
        Boolean last_loop=0;

        while(gen_contor++<how_many)
          {
            if(auto_segm)
          {
                memset(sel_col,0,_n_colors);
                new_auto_loop(final_T,sel_col);
                L=T[rect][0]-final_T[0]; U=T[rect][1]-final_T[1];
                V=T[rect][2]-final_T[2]; RAD2=L*L+U*U+V*V;
                if(RAD2<0.1){
                  my_contor=0;
                  memset(my_class,0,_ro_col);
                  for(k=0;k<n_remain;k++)
                {
                      register int p=gen_remain[k];
                if(sel_col[_col_index[p]])
                  {
                    selected[my_contor++]=p;
                          my_class[p]=1;
                  }
                }
                  break;
              }
             else
                 {
                T[rect][0]=final_T[0];T[rect][1]=final_T[1];
                  T[rect][2]=final_T[2];
                 }
              }
             else
             {
                my_contor=0;
                    memset(my_class,0,_ro_col);
                  nauto_loop(final_T, selected, my_class,&my_contor);
                    mean_s(my_contor, _p, selected, _data, final_T);
                    L=T[rect][0]-final_T[0]; U=T[rect][1]-final_T[1];
                    V=T[rect][2]-final_T[2]; RAD2=L*L+U*U+V*V;

                    if(RAD2<0.1)
                      break;
                    else
                     {
                     T[rect][0]=final_T[0];T[rect][1]=final_T[1];
                       T[rect][2]=final_T[2];
                     }
              }
          }
      }
      if(auto_segm)
      {
          if(option==0) test_neigh(my_class, selected, &my_contor,2);
          else if(option==1) test_neigh(my_class, selected, &my_contor,1);
          else if(option==2) test_neigh(my_class, selected, &my_contor,0);
      }
#ifdef TRACE
      printf("my_contor = %d contor_trials = %d",my_contor,contor_trials);
#endif
      if(!auto_segm)
        if(test_same_cluster(rect,T))
        {
```

```
            delete [] my_class; delete [] sel_col; delete [] selected;
            continue;
        }
      if(auto_segm && my_contor<act_threshold)
        {
          delete [] my_class; delete [] sel_col; delete [] selected;
          if(contor_trials++<MAX_TRIAL) goto MS;
          else                          break;
        }

      if(auto_segm) my_actual(my_class);

      my_convert(selects, final_T, TI[rect]);
      for ( k = 0; k < _ro_col; k++ )
        if(my_class[k])
          gen_class[k]=rect;

      delete [] my_class; delete [] sel_col; delete [] selected;
    }
  n_rects=rect;
  Octet** isegment = new Octet*[3];
  register int j;
  for ( j = 0; j < 3; j++ )
    {
      isegment[j] = new Octet[_ro_col];
      memset( isegment[j], 0, _ro_col );
    }
  Octet *isegment0=isegment[0]; Octet *isegment1=isegment[1];
  Octet *isegment2=isegment[2];

  if(auto_segm)
    {
      Octet *my_class = new Octet[_ro_col];
      for ( k = 0; k < _ro_col; k++ )
         if(gen_class[k]==255) gen_class[k]=n_rects;

      Octet inv_map[Max_rects];
      for(k=0;k<n_rects;k++) inv_map[k]=k;

      if(option==0)  conn_comp(my_class,&n_rects,inv_map,T,10,0);
      else if(option==1) conn_comp(my_class,&n_rects,inv_map,T,20,0);
      else conn_comp(my_class,&n_rects,inv_map,T,act_threshold,0);
      optimize(T,n_rects);
      for(k=0;k<n_rects;k++) my_convert(selects, T[k], TI[k]);

      // Postprocessing
      if(option==1 || option ==2)
          for(k=2;k<10;k+=2) conn_comp(my_class,&n_rects,inv_map,T,k,1);

      end0 = mell_get_time();
      t = mell_time_to_sec(mell_time_diff(end0,begin0));

      register Octet my_max_ind;
      for ( k = 0; k < _ro_col; k++ )
        {
          if((my_max_ind=gen_class[k])!=n_rects)
          {
              int *ti=TI[my_max_ind];
            isegment0[k]=ti[0]; isegment1[k]=ti[1]; isegment2[k]=ti[2];
          }
      }
```

```
//Nostra aggiunta per la stampa a video dei cluster
      long numpixel = 0; int rr,gg,bb;
      cout <<"\n\n";
      for (k=0;k<n_rects;k++)
         {
                numpixel = 0;
            for (long kk=0;kk<_ro_col;kk++)
            {
                  if (gen_class[kk]==k)
            {
                        numpixel++;
                  rr = isegment0[kk];
                  gg = isegment1[kk];
                  bb = isegment2[kk];
            }
            }
            printf ("Cluster %3d: r=%3d; g=%3d; b=%3d; number of
pixels=%6ld\n",k+1,rr,gg,bb,numpixel);

             }
//FINE aggiunta

//Stampa dei tempi
printf("\n");

printf("Tempo impiegato     : %10.4f s\n",t);
printf("Throughput          : %10.0f pixel/s\n",_ro_col/t);
printf("frames/sec (300x300): %10.6f frames/s\n",_ro_col/t/300/300);

printf("\n");
//Fine stampa tempi


      // Save the borders
      memset(my_class,255,_ro_col);
      for ( k = 0; k < _ro_col; k++ )
      {
          int pres_class=gen_class[k];
          int pos_colms=k%_colms;

          if(pos_colms==0 || pos_colms==(_colms-1) ||
             k<(_colms-1) || k>(_ro_col-_colms))
             my_class[k] = 0;
          else
             for(j=1;j<8;j+=2)
               {
                 int u=k+my_neigh[j];
                 if(u>=0 && u<_ro_col &&  (pres_class<gen_class[u]))
               {
                 my_class[k] = 0; break;
               }
              }
      }
      my_write_PGM_file( pgmfile, my_class,_rows,_colms);
      delete [] my_class;
    }
  else   // if not auto_segmentation
    {
      optimize(T,n_rects);
      for(k=0;k<n_rects;k++)
        my_convert(selects, T[k], TI[k]);
      register int temp_class;
      for (k=0;k<_ro_col;k++)
```

```
        {
            if((temp_class=gen_class[k])<n_rects)
            {
                 isegment0[k] = /*TI[temp_class][0];*/pic->chdata(0)[k];
               isegment1[k] = /*TI[temp_class][1];*/pic->chdata(1)[k];
               isegment2[k] = /*TI[temp_class][2];*/pic->chdata(2)[k];
             }
          }
      }
  if(auto_segm){
    delete [] _m_col_remain;
    delete [] _col_remain;
    delete [] gen_remain;
  }
  delete [] gen_class;
  delete [] _col_index;
  delete [] _col0; delete [] _col1; delete [] _col2;
  delete [] _col_all;

  XfRaster::Info  info;
  pic->raster_info(info);
  result_ras_ = new RasterIpChannels( info, 3, eDATA_OCTET,
                              isegment, true );
  cout << endl << "Original Colors: " << _n_colors << endl;
  cout << "Numero regioni: " << n_rects << endl;

  return true;
}
```

```
//Color Image Segmentation
//This is the implementation of the algorithm described in
//D. Comaniciu, P. Meer,
//Robust Analysis of Feature Spaces: Color Image Segmentation,
//http://www.caip.rutgers.edu/~meer/RIUL/PAPERS/feature.ps.gz
//appeared in Proceedings of CVPR'97, San Juan, Puerto Rico.
// ============================================================================
// =====      Module: segm_hh
// ===== ------------------------------------------------------------ ======
// =====      Version 01   Date: 04/22/97
// ===== ------------------------------------------------------------ ======
// ============================================================================
// =====      Written by Dorin Comaniciu
// =====      e-mail:  comanici@caip.rutgers.edu
// ============================================================================
// Permission to use, copy, or modify this software and its documentation
// for educational and research purposes only is hereby granted without
// fee, provided that this copyright notice appear on all copies and
// related documentation.  For any other uses of this software, in original
// or modified form, including but not limited to distribution in whole
// or in part, specific prior permission must be obtained from
// the author(s).
//
// THE SOFTWARE IS PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND,
// EXPRESS, IMPLIED OR OTHERWISE, INCLUDING WITHOUT LIMITATION, ANY
// WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.
//
// IN NO EVENT SHALL RUTGERS UNIVERSITY BE LIABLE FOR ANY SPECIAL,
// INCIDENTAL, INDIRECT OR CONSEQUENTIAL DAMAGES OF ANY KIND, OR ANY
// DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
// WHETHER OR NOT ADVISED OF THE POSSIBILITY OF DAMAGE, AND ON ANY
// THEORY OF LIABILITY, ARISING OUT OF OR IN CONNECTION WITH THE USE
// OR PERFORMANCE OF THIS SOFTWARE.
// ============================================================================
//


#include <assert.h>
#define         ORG_SIZE

//#define         DEBUG

//#define   TRACE

#define         Max_rects       50

#define MIN(x,y)         ((x) <= (y) ? (x) : (y))
#define Min3(x,y,z)      (MIN((x),(y)) <= (z) ? MIN((x),(y)) : (z))

#  include <memory.h>
#  define       bzero(b,len)            memset(b,'\0',len)
#  define       bcopy(b1,b2,len)        memmove(b2,b1,len)

typedef char Boolean;         // copied from <X11/Intrinsic.h>
typedef unsigned char Octet;

#define false 0
#define true !false
#define nil 0

class XfRaster {
 public:
    XfRaster() { };
```

```
    ~XfRaster() { };

    struct Info {
        int rows;
        int columns;
        int origin_x;
        int origin_y;
    };
};

enum DataType {              // Values of data type:
        eDATA_OCTET  =  10,  // unsigned char data
        eDATA_FLOAT          // floating point data
    };


class RasterIpChannels : public XfRaster {
 public:

    RasterIpChannels(
      const XfRaster::Info& info, const int n_channels,
      const DataType, Octet* data[], const Boolean
    );
    ~RasterIpChannels();

    Octet** chdata() const;
    Octet*  chdata(int i) const;
    int           n_channels() const;
    DataType      dtype() const;
    void    clipf(Boolean cf);
    Boolean clipf() const;

    void raster_info(Info& i);
    static unsigned char** raster_float_to_Octet( RasterIpChannels& ras );

    int           rows_;
    int           columns_;
    DataType      dtype_;
    Octet** chdata_;
    int           n_channels_;
    Boolean clipf_;           // flag that determines wheter to clip or not
                        //   when converting from eDATA_FLOAT to Octet
};

inline Octet** RasterIpChannels::chdata() const   { return chdata_; }
inline Octet* RasterIpChannels::chdata(int i) const {
    assert((0 <= i) && (i < n_channels_));
    return chdata_[i];
}
inline int RasterIpChannels::n_channels() const   { return n_channels_; }
inline DataType RasterIpChannels::dtype() const   { return dtype_; }
inline void RasterIpChannels::clipf(Boolean cf)   { clipf_ = cf; }
inline Boolean RasterIpChannels::clipf() const   { return clipf_; }


const int   p_max       =     3;    // max space dim;  MAX(p_max) == 6 !!

typedef struct region_str
    {
      int my_contor;
      int my_class;
      int my_label;
      int *my_region;
```

```
      struct region_str *next_region_str;
    }REGION;

class SegmenterMS {

 public:

  SegmenterMS( );
  ~SegmenterMS();

  struct sRectangle {
    int x, y;                       // upper left corner
    unsigned int width, height;             // rectangle dimensions
  };

  RasterIpChannels*     result_ras_;      // result of the visit to a raster

  Boolean   ms_segment( RasterIpChannels*, sRectangle*, int, long,
                        unsigned int, Boolean);

 protected:

  int       _p;         // parameter space dimension
  int       _p_ptr;           // counter for the number of components already
                              // stored in _data_all
  int**             _data_all;      // input data
  int*_data0,*_data1,*_data2;   //to speed up

  int       _colms, _rows;    // width, height dimensions of the input data
  int           _ro_col;        // how many points in input data
  int**         _data;        // region of the input data currently segmented
  int       _rcolms, _rrows;// width, height dimensions of the region
  int       _n_points;  // number of data points (per channel)
  float           _min_per;       // minimum/maximum cluster size
  int       _NJ;        // maximum number of subsamples
  Boolean       auto_segm;
  int*          gen_remain;     //the map of remaining points
  Octet*        gen_class;      //labels
  Octet*        taken;
  int*          conn_selected;  //used by connected component
  int           n_remain;       //# of remaining points
  int           _n_colors;      //# of colors
  int**         _col_all;       //colors in the image (LUV)
  int*_col0,*_col1,*_col2;      //to speed up
  int *         _col_RGB;       //colors in the image (RGB)
  int *         _col_index;     //color address
  int *         _col_misc;      //misc use in histogram and conversion
  int*          _m_colors;      //how many of each color

  int*          _col_remain;    //the map of remaining colors
  int*          _m_col_remain;  //table of remaining colors
  int           _n_col_remain;  //# of remaining colors

  void convert_RGB_LUV( RasterIpChannels* , long );
  void cut_rectangle( sRectangle* );
  void init_matr(void);
  float my_sampling( int, float T[Max_rects][p_max]);
  int  subsample( float* );
  void test_neigh(Octet* , int *, int*, int);
  void my_actual(Octet*);
  void init_neigh(void);
  void conn_comp(Octet *, int* , Octet* ,float [] [p_max],int,int);
  void find_other_neigh(int, int *, REGION *);
```

```
  void new_auto_loop(float *, Octet*);
  void nauto_loop(float *, int *, Octet *, int *);
  void optimize(float [][p_max], int);
  void get_codeblock(float [][p_max], int);
  void get_codeblock1(float [][p_max], int);
  void new_codebook(float [][p_max], int);
  void my_clean(float [][p_max], int);
  void my_histogram(RasterIpChannels*, long);
  void eliminate_class(Octet *,int *,int *,int, Octet *, float [][p_max],REGION
*);
  void eliminate_region(int *,int,float [][p_max],REGION *);
  void destroy_region_list(REGION *);
  REGION *create_region_list(int *, int);
  void initializations(RasterIpChannels*, sRectangle [], int *, long , int *);
};

// List of color components; the possible cases are:

#define RedColor        (1<<0)
#define GreenColor           (1<<1)
#define BlueColor       (1<<2)
#define Lightness       (1<<3)
#define Ustar           (1<<4)
#define Vstar           (1<<5)

#define MAXV 256
```

MAKEFILE

```
segm: segm.cc segm_main.cc
	g++ -O -o segm segm.cc segm_main.cc -lm
```

Robust Analysis of Feature Spaces: Color Image Segmentation

Dorin Comaniciu Peter Meer
Department of Electrical and Computer Engineering
Rutgers University, Piscataway, NJ 08855, USA

Keywords: *robust pattern analysis, low-level vision, content-based indexing*

Abstract

A general technique for the recovery of significant image features is presented. The technique is based on the mean shift algorithm, a simple nonparametric procedure for estimating density gradients. Drawbacks of the current methods (including robust clustering) are avoided. Feature space of any nature can be processed, and as an example, color image segmentation is discussed. The segmentation is completely autonomous, only its class is chosen by the user. Thus, the same program can produce a high quality edge image, or provide, by extracting all the significant colors, a preprocessor for content-based query systems. A 512×512 *color image is analyzed in less than 10 seconds on a standard workstation. Gray level images are handled as color images having only the lightness coordinate.*

1 Introduction

Feature space analysis is a widely used tool for solving low-level image understanding tasks. Given an image, feature vectors are extracted from local neighborhoods and mapped into the space spanned by their components. Significant features in the image then correspond to high density regions in this space. Feature space analysis is the procedure of recovering the centers of the high density regions, i.e., the representations of the significant image features. Histogram based techniques, Hough transform are examples of the approach.

When the number of distinct feature vectors is large, the size of the feature space is reduced by grouping nearby vectors into a single cell. A discretized feature space is called an accumulator. Whenever the size of the accumulator cell is not adequate for the data, serious artifacts can appear. The problem was extensively studied in the context of the Hough transform, e.g. [5]. Thus, for satisfactory results *a feature space should have continuous coordinate system.* The content of a continuous feature space can be modeled as a sample from a multivariate, multimodal probability distribution. Note that for real images the number of modes can be very large, of the order of tens.

The highest density regions correspond to clusters centered on the modes of the underlying probability distribution. Traditional clustering techniques [6], can be used for feature space analysis but they are reliable only if the number of clusters is small *and* known a priori. Estimating the number of clusters from the data is computationally expensive and not guaranteed to produce satisfactory result.

A much too often used assumption is that the individual clusters obey multivariate normal distributions, i.e., the feature space can be modeled as a mixture of Gaussians. The parameters of the mixture are then estimated by minimizing an error criterion. For example, a large class of thresholding algorithms are based on the Gaussian mixture model of the histogram, e.g. [11]. However, there is no theoretical evidence that an extracted normal cluster necessarily corresponds to a significant image feature. On the contrary, a strong artifact cluster may appear when several features are mapped into partially overlapping regions.

Nonparametric density estimation [4, Chap. 6] avoids the use of the normality assumption. The two families of methods, Parzen window, and k-nearest neighbors, both require additional input information (type of the kernel, number of neighbors). This information must be provided by the user, and for multimodal distributions it is difficult to guess the optimal setting.

Nevertheless, a reliable general technique for feature space analysis can be developed using a simple nonparametric density estimation algorithm. In this paper we propose such a technique whose robust behavior is superior to methods employing robust estimators from statistics.

2 Requirements for Robustness

Estimation of a cluster center is called in statistics the multivariate location problem. To be robust, an estimator must tolerate a percentage of outliers, i.e., data points not obeying the underlying distribution

of the cluster. Numerous robust techniques were proposed [10, Sec. 7.1], and in computer vision the most widely used is the *minimum volume ellipsoid* (MVE) estimator proposed by Rousseeuw [10, p. 258].

The MVE estimator is affine equivariant (an affine transformation of the input is passed on to the estimate) and has high breakdown point (tolerates up to half the data being outliers). The estimator finds the center of the highest density region by searching for the minimal volume ellipsoid containing at least h data points. The multivariate location estimate is the center of this ellipsoid. To avoid combinatorial explosion a probabilistic search is employed. Let the dimension of the data be p. A small number of $(p+1)$-tuple of points are randomly chosen. For each $(p+1)$-tuple the mean vector and covariance matrix are computed, defining an ellipsoid. The ellipsoid is inflated to include h points, and the one having the minimum volume provides the MVE estimate.

Based on MVE, a robust clustering technique with applications in computer vision was proposed in [7]. The data is analyzed under several "resolutions" by applying the MVE estimator repeatedly with h values representing fixed percentages of the data points. The best cluster then corresponds to the h value yielding the highest density inside the minimum volume ellipsoid. The cluster is removed from the feature space, and the whole procedure is repeated till the space is not empty. The robustness of MVE should ensure that each cluster is associated with only one mode of the underlying distribution. The number of significant clusters is not needed a priori.

The robust clustering method was successfully employed for the analysis of a large variety of feature spaces, but was found to become less reliable once the number of modes exceeded ten. This is mainly due to the normality assumption embedded into the method. The ellipsoid defining a cluster can be also viewed as the high confidence region of a multivariate normal distribution. Arbitrary feature spaces are not mixtures of Gaussians and constraining the shape of the removed clusters to be elliptical can introduce serious artifacts. The effect of these artifacts propagates as more and more clusters are removed. Furthermore, the estimated covariance matrices are not reliable since are based on only $p+1$ points. Subsequent postprocessing based on all the points declared inliers cannot fully compensate for an initial error.

To be able to correctly recover a large number of significant features, the problem of feature space analysis must be solved in context. In image understanding tasks the data to be analyzed originates in the *image domain*. That is, the feature vectors satisfy additional, spatial constraints. While these constraints are indeed used in the current techniques, their role is mostly limited to compensating for feature allocation errors made during the *independent* analysis of the feature space. To be robust *the feature space analysis must fully exploit the image domain information.*

As a consequence of the increased role of image domain information the burden on the feature space analysis can be reduced. First *all* the significant features are extracted, and *only after then* are the clusters containing the instances of these features recovered. The latter procedure uses image domain information and avoids the normality assumption.

Significant features correspond to high density regions and to locate these regions a search window must be employed. The number of parameters defining the shape and size of the window should be minimal, and therefore whenever it is possible *the feature space should be isotropic.* A space is isotropic if the distance between two points is independent on the location of the point pair. The most widely used isotropic space is the Euclidean space, where a sphere, having only one parameter (its radius) can be employed as search window. The isotropy requirement determines the mapping from the image domain to the feature space. If the isotropy condition cannot be satisfied, a Mahalanobis metric should be defined from the statement of the task.

We conclude that robust feature space analysis requires a reliable procedure for the detection of high density regions. Such a procedure is presented in the next section.

3 Mean Shift Algorithm

A simple, nonparametric technique for estimation of the density gradient was proposed in 1975 by Fukunaga and Hostetler [4, p. 534]. The idea was recently generalized by Cheng [2].

Assume, for the moment, that the probability density function $p(\mathbf{x})$ of the p-dimensional feature vectors $\mathbf{x}$ is unimodal. This condition is for sake of clarity only, later will be removed. A sphere $\mathcal{S}_{\mathbf{x}}$ of radius r, centered on $\mathbf{x}$ contains the feature vectors $\mathbf{y}$ such that $\| \mathbf{y}-\mathbf{x} \| \leq r$. The expected value of the vector $\mathbf{z} = \mathbf{y}-\mathbf{x}$, given $\mathbf{x}$ and $\mathcal{S}_{\mathbf{x}}$ is

$$\begin{aligned} \mu = E[\,\mathbf{z}|\mathcal{S}_{\mathbf{x}}\,] &= \int_{\mathcal{S}_{\mathbf{x}}} (\mathbf{y}-\mathbf{x}) p(\mathbf{y}|\mathcal{S}_{\mathbf{x}}) d\mathbf{y} \qquad (1) \\ &= \int_{\mathcal{S}_{\mathbf{x}}} (\mathbf{y}-\mathbf{x}) \frac{p(\mathbf{y})}{p(\mathbf{y} \in \mathcal{S}_{\mathbf{x}})} d\mathbf{y} \end{aligned}$$

If $\mathcal{S}_{\mathbf{X}}$ is sufficiently small we can approximate

$$p(\mathbf{y} \in \mathcal{S}_{\mathbf{X}}) = p(\mathbf{x}) V_{\mathcal{S}_{\mathbf{X}}} \qquad \text{where} \quad V_{\mathcal{S}_{\mathbf{X}}} = c \cdot r^p \tag{2}$$

is the volume of the sphere. The first order approximation of $p(\mathbf{y})$ is

$$p(\mathbf{y}) = p(\mathbf{x}) + (\mathbf{y} - \mathbf{x})^T \nabla p(\mathbf{x}) \tag{3}$$

where $\nabla p(\mathbf{x})$ is the gradient of the probability density function in $\mathbf{x}$. Then

$$\mu = \int_{\mathcal{S}_{\mathbf{X}}} \frac{(\mathbf{y} - \mathbf{x})(\mathbf{y} - \mathbf{x})^T}{V_{\mathcal{S}_{\mathbf{X}}}} \frac{\nabla p(\mathbf{x})}{p(\mathbf{x})} d\mathbf{y} \tag{4}$$

since the first term term vanishes. The value of the integral is [4, p. 535]

$$\mu = \frac{r^2}{p+2} \frac{\nabla p(\mathbf{x})}{p(\mathbf{x})} \tag{5}$$

or

$$E[\,\mathbf{x} \mid \mathbf{x} \in \mathcal{S}_{\mathbf{X}}\,] - \mathbf{x} = \frac{r^2}{p+2} \frac{\nabla p(\mathbf{x})}{p(\mathbf{x})} \tag{6}$$

Thus, the mean shift vector, the vector of difference between the local mean and the center of the window, is proportional to the gradient of the probability density at $\mathbf{x}$. The proportionality factor is reciprocal to $p(\mathbf{x})$. This is beneficial when the highest density region of the probability density function is sought. Such region corresponds to large $p(\mathbf{x})$ and small $\nabla p(\mathbf{x})$, i.e., to small mean shifts. On the other hand, low density regions correspond to large mean shifts (amplified also by small $p(\mathbf{x})$ values). The shifts are always in the direction of the probability density maximum, the mode. At the mode the mean shift is close to zero. This property can be exploited in a simple, adaptive steepest ascent algorithm.

Mean Shift Algorithm

1. Choose the radius r of the search window.
2. Choose the initial location of the window.
3. Compute the mean shift vector and translate the search window by that amount.
4. Repeat till convergence.

To illustrate the ability of the mean shift algorithm, 200 data points were generated from two normal distributions, both having unit variance. The first hundred points belonged to a zero-mean distribution, the second hundred to a distribution having mean 3.5. The data is shown as a histogram in Figure 1. It should be emphasized that the feature space is processed as an ordered one-dimensional sequence of points, i.e., it is continuous. The mean shift algorithm starts from the location of the mode detected by the one-dimensional MVE mode detector, i.e., the center of the shortest rectangular window containing half the data points [10, Sec. 4.2]. Since the data is bimodal with nearby modes, the mode estimator fails and returns a location in the trough. The starting point is marked by the cross at the top of Figure 1.

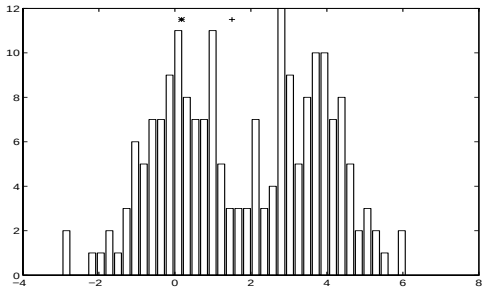


Figure 1: An example of the mean shift algorithm.

In this synthetic data example no a priori information is available about the analysis window. Its size was taken equal to that returned by the MVE estimator, 3.2828. Other, more adaptive strategies for setting the search window size can also be defined.

Table 1: *Evolution of Mean Shift Algorithm*

Initial Mode	Initial Mean	Final Mean
1.5024	1.4149	0.1741

In Table 1 the initial values and the final location, shown with a star at the top of Figure 1, are given.

The mean shift algorithm is the tool needed for feature space analysis. The unimodality condition can be relaxed by randomly choosing the initial location of the search window. The algorithm then converges to the closest high density region. The outline of a general procedure is given below.

Feature Space Analysis

1. Map the image domain into the feature space.
2. Define an adequate number of search windows at random locations in the space.
3. Find the high density region centers by applying the mean shift algorithm to each window.
4. Validate the extracted centers with image domain constraints to provide the *feature palette*.
5. Allocate, using image domain information, all the feature vectors to the feature palette.

The procedure is very general and applicable to any feature space. In the next section we describe a color image segmentation technique developed based on this outline.

4 Color Image Segmentation

Image segmentation, partioning the image into homogeneous regions, is a challenging task. The richness of visual information makes bottom-up, solely image driven approaches always prone to errors. To be reliable, the current systems must be large and incorporate numerous ad-hoc procedures, e.g. [1]. The paradigms of gray level image segmentation (pixel-based, area-based, edge-based) are also used for color images. In addition, the physics-based methods take into account information about the image formation processes as well. See, for example, the reviews [8, 12]. The proposed segmentation technique does not consider the physical processes, it uses only the given image, i.e., a set of RGB vectors. Nevertheless, can be easily extended to incorporate supplementary information about the input. As homogeneity criterion color similarity is used.

Since perfect segmentation cannot be achieved without a top-down, knowledge driven component, a bottom-up segmentation technique should

- only provide the input into the next stage where the task is accomplished using a priori knowledge about its goal; and
- eliminate, as much as possible, the dependence on user set parameter values.

Segmentation resolution is the most general parameter characterizing a segmentation technique. While this parameter has a continuous scale, three important classes can be distinguished.

Undersegmentation corresponds to the lowest resolution. Homogeneity is defined with a large tolerance margin and only the most significant colors are retained for the feature palette. The region boundaries in a correctly undersegmented image are the dominant edges in the image.

Oversegmentation corresponds to intermediate resolution. The feature palette is rich enough that the image is broken into many small regions from which any sought information can be assembled under knowledge control. Oversegmentation is the recommended class when the goal of the task is object recognition.

Quantization corresponds to the highest resolution. The feature palette contains all the important colors in the image. This segmentation class became important with the spread of image databases, e.g., [3, 9]. The full palette, possibly together with the underlying spatial structure, is essential for content-based queries.

The proposed color segmentation technique operates in any of the these three classes. The user only chooses the desired class, the specific operating conditions are derived automatically by the program.

Images are usually stored and displayed in the RGB space. However, to ensure the isotropy of the feature space, a uniform color space with the perceived color differences measured by Euclidean distances should be used. We have chosen the $L^*u^*v^*$ space [13, Sec. 3.3.9], whose coordinates are related to the RGB values by nonlinear transformations. The daylight standard D_{65} was used as reference illuminant. The chromatic information is carried by u^* and v^*, while the lightness coordinate L^* can be regarded as the relative brightness. Psychophysical experiments show that $L^*u^*v^*$ space may not be perfectly isotropic [13, p. 311], however, it was found satisfactory for image understanding applications. The image capture/display operations also introduce deviations which are most often neglected.

The steps of color image segmentation are presented below. The acronyms **ID** and **FS** stand for image domain and feature space respectively. All feature space computations are performed in the $L^*u^*v^*$ space.

1. [**FS**] *Definition of the segmentation parameters.* The user only indicates the desired class of segmentation. The class definition is translated into three parameters

- the radius of the search window, r;
- the smallest number of elements required for a significant color, N_{min};
- the smallest number of contiguous pixels required for a significant image region, N_{con}.

The size of the search window determines the resolution of the segmentation, smaller values corresponding to higher resolutions. The subjective (perceptual) definition of a homogeneous region seems to depend on the "visual activity" in the image. Within the same segmentation class an image containing large homogeneous regions should be analyzed at higher resolution than an image with many textured areas. The simplest measure of the "visual activity" can be derived from the global covariance matrix. The square root of its trace, σ, is related to the power of the signal (image). The radius r is taken proportional to σ. The rules defining the three segmentation class parameters are given in Table 2. These rules were used in the segmentation of a large variety images, ranging from simple blood cells to complex indoor and outdoor scenes.

When the goal of the task is well defined and/or all the images are of the same type, the parameters can be fine tuned.

Table 2: *Segmentation Class Parameters*

Segmentation Class	Parameter		
	r	N_{min}	N_{con}
Undersegmentation	0.4σ	400	10
Oversegmentation	0.3σ	100	10
Quantization	0.2σ	50	0

2. [**ID+FS**] *Definition of the search window.*
The initial location of the search window in the feature space is randomly chosen. To ensure that the search starts close to a high density region several location candidates are examined. The random sampling is performed in the image domain and a few, $M = 25$, pixels are chosen. For each pixel, the mean of its 3×3 neighborhood is computed and mapped into the feature space. If the neighborhood belongs to a larger homogeneous region, with high probability the location of the search window will be as wanted. To further increase this probability, the window containing the highest density of feature vectors is selected from the M candidates.

3. [**FS**] *Mean shift algorithm.*
To locate the closest mode the mean shift algorithm is applied to the selected search window. Convergence is declared when the magnitude of the shift becomes less than 0.1.

4. [**ID+FS**] *Removal of the detected feature.*
The pixels yielding feature vectors inside the search window at its final location are discarded from both domains. Additionally, their 8-connected neighbors in the image domain are also removed *independent* of the feature vector value. These neighbors can have "strange" colors due to the image formation process and their removal cleans the background of the feature space. Since all pixels are reallocated in Step 7, possible errors will be corrected.

5. [**ID+FS**] *Iterations.*
Repeat Steps 2 to 4, till the number of feature vectors in the selected search window no longer exceeds N_{min}.

6. [**ID**] *Determining the initial feature palette.*
In the feature space a significant color must be based on minimum N_{min} vectors. Similarly, to declare a color significant in the image domain more than N_{min} pixels of that color should belong to a connected component. From the extracted colors only those are retained for the initial feature palette which yield at least one connected component in the image of size larger than N_{min}. The neighbors removed at Step 4. are also considered when defining the connected components Note that the threshold is not N_{con} which is used only at the postprocessing stage.

7. [**ID+FS**] *Determining the final feature palette.*
The initial feature palette provides the colors allowed when segmenting the image. If the palette is not rich enough the segmentation resolution was not chosen correctly and should be increased to the next class. All the pixel are reallocated based on this palette. First, the pixels yielding feature vectors inside the search windows at their final location are considered. These pixels are allocated to the color of the window center without taking into account image domain information. The windows are then inflated to double volume (their radius is multiplied with $\sqrt[3]{2}$). The newly incorporated pixels are retained only if they have at least one neighbor which was already allocated to that color. The mean of the feature vectors mapped into the same color is the value retained for the final palette. At the end of the allocation procedure a small number of pixels can remain unclassified. These pixels are allocated to the closest color in the final feature palette.

8. [**ID+FS**] *Postprocessing.*
This step depends on the goal of the task. The simplest procedure is the removal from the image of all small connected components of size less than N_{con}. These pixels are allocated to the majority color in their 3×3 neighborhood, or in the case of a tie to the closest color in the feature space.

In Figure 2 the *house* image containing 9603 different colors is shown. The segmentation results for the three classes and the region boundaries are given in Figure 5a–f. Note that undersegmentation yields a good edge map, while in the quantization class the original image is closely reproduced with only 37 colors. A second example using the oversegmentation class is shown in Figure 3. Note the details on the fuselage.

5 Discussion

The simplicity of the basic computational module, the mean shift algorithm, enables the feature space analysis to be accomplished very fast. From a 512×512 pixels image a palette of 10–20 features can be extracted in less than 10 seconds on a Ultra SPARC 1 workstation. To achieve such a speed the implementation was optimized and whenever possible, the feature space (containing fewer distinct elements than the image domain) was used for array scanning; lookup tables were employed instead of fre-

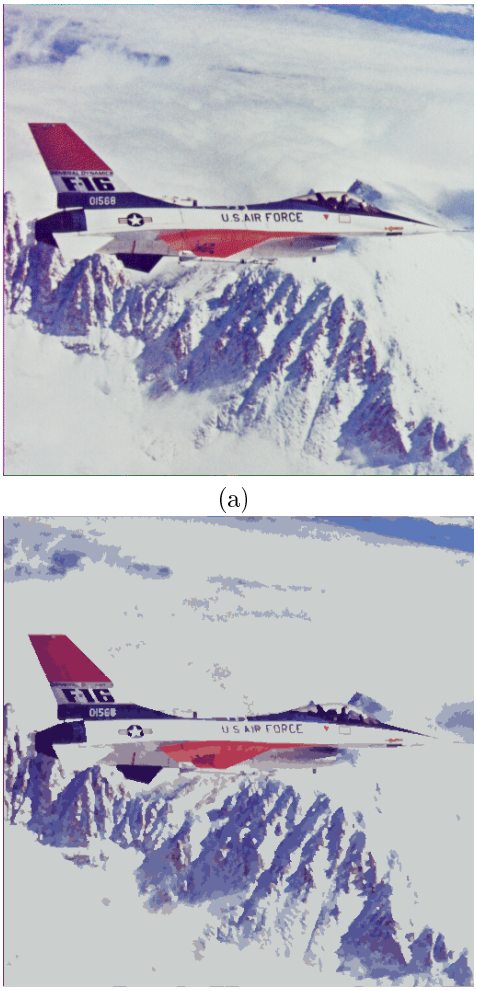
Figure 2: The *house* image, 255 × 192 pixels, 9603 colors.

quently repeated computations; direct addressing instead of nested pointers; fixed point arithmetic instead of floating point calculations; partial computation of the Euclidean distances, etc.

The analysis of the feature space is completely autonomous, due to the extensive use of image domain information. All the examples in this paper, and dozens more not shown here, were processed using the parameter values given in Table 2. Recently Zhu and Yuille [14] described a segmentation technique incorporating complex global optimization methods (snakes, minimum description length) with sensitive parameters and thresholds. To segment a color image over a hundred iterations were needed. When the images used in [14] were processed with the technique described in this paper, the same quality results were obtained unsupervised and in less than a second. Figure 4 shows one of the results, to be compared with Figure 14h in [14]. The new technique can be used *unmodified* for segmenting gray level images, which are handled as color images with only the L^* coordinates. In Figure 6 an example is shown.

The result of segmentation can be further refined by local processing in the image domain. For example, robust analysis of the pixels in a large connected component yields the inlier/outlier dichotomy which then can be used to recover discarded fine details.

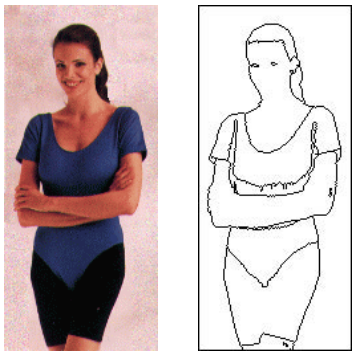
In conclusion, we have presented a general technique for feature space analysis with applications in many low-level vision tasks like thresholding, edge detection, segmentation. The nature of the feature space is not restricted, currently we are working on applying the technique to range image segmentation, Hough transform and optical flow decomposition.



(a)

(b)

Figure 3: Color image segmentation example. (a) Original image, 512 × 512 pixels, 77041 colors. (b) Oversegmentation: 21/21 colors.



(a) (b)

Figure 4: Performance comparison. (a) Original image, 116 × 261 pixels, 200 colors. (b) Undersegmentation: 5/4 colors. Region boundaries.

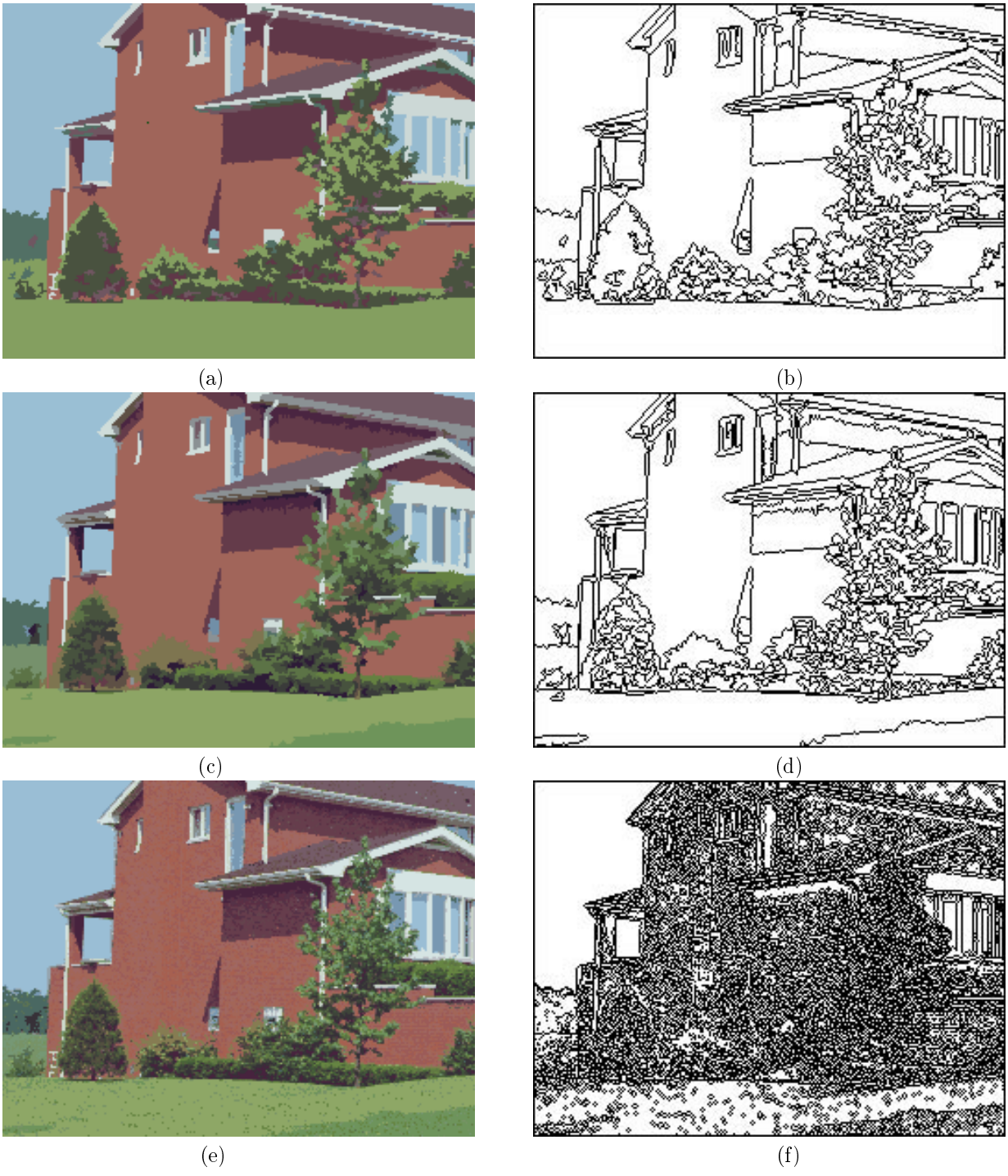


Figure 5: The three segmentation classes for the *house* image. The right column shows the region boundaries. (a)(b) Undersegmentation. Number of colors extracted initially and in the feature palette: 8/8. (c)(d) Oversegmentation: 24/19 colors. (e)(f) Quantization: 49/37 colors.

Acknowledgement

The research was supported by the National Science Foundation under the grant IRI-9530546.

References

[1] J.R. Beveridge, J. Griffith, R.R. Kohler, A.R. Hanson, E.M. Riseman, "Segmenting images using localized histograms and region merging", *Int'l. J. of Comp. Vis.*, vol. 2, 311–347, 1989.

[2] Y. Cheng, "Mean shift, mode seeking, and clustering", *IEEE Trans. Pattern Anal. Machine Intell.*, vol. 17, 790–799, 1995.

[3] M. Flickner et al., "Query by image and video content: The QBIC system", *Computer*, vol. 28, no. 9, 23–32, 1995.

[4] K. Fukunaga, *Introduction to Statistical Pattern Recognition*, Second Ed., Boston: Academic Press, 1990.

[5] J. Illingworth, J. Kittler, "A survey of the Hough transform", *Comp. Vis., Graph. and Imag. Proc.*, vol. 44, 87–116, 1988.

[6] A.K. Jain, R.C. Dubes, *Algorithms for Clustering Data*, Englewood Cliff, NJ: Prentice Hall, 1988.

[7] J.-M. Jolion, P. Meer, S. Bataouche, "Robust clustering with applications in computer vision," *IEEE Trans. Pattern Anal. Machine Intell.*, vol. 13, 791–802, 1991.

[8] Q.T. Luong, "Color in computer vision", In *Handbook of Pattern Recognition and Computer Vision*, C.H. Chen, L.F. Pau, and P.S P. Wang (Eds.), Singapore: World Scientific, 311–368, 1993.

[9] A. Pentland, R.W. Picard, S. Sclaroff, "Photobook: Content-based manipulation of image databases", *Int'l. J. of Comp. Vis.* vol. 18, 233–254, 1996.

[10] P.J. Rousseeuw, A.M. Leroy, *Robust Regression and Outlier Detection*. New York: Wiley, 1987.

[11] P.K. Sahoo, S. Soltani, A.K.C. Wong, "A survey of thresholding techniques", *Comp. Vis., Graph. and Imag. Proc.*, vol. 41, 233–260, 1988.

[12] W. Skarbek, A. Koschan, *Colour Image Segmentation – A Survey*, Technical Report 94-32, Technical University Berlin, October 1994.

[13] G. Wyszecki, W.S. Stiles, *Color Science: Concepts and Methods, Quantitative Data and Formulae*, Second Ed. New York: Wiley, 1982.

[14] S.C. Zhu, A. Yuille, "Region competition: Unifying snakes, region growing, and Bayes/MDL for multiband image segmentation", *IEEE Trans. Pattern Anal. Machine Intell.*, Vol. 18, 884–900, 1996.

(a) (b) (c)

Figure 6: Gray level image segmentation example. (a) Original image, 256×256 pixels. (b) Undersegmentation: 5 gray levels. (c) Region boundaries.