Internship Report

on

Parallelized reversible digital watermarking using C++ in OpenCV

Made by:

Asmita Poddar

At

Indian Institute of Technology, Kharagpur

Guide: Dr. Rajat Subhra Chakraborty

Department: Computer Science and Engineering

Reversible digital watermarking

Digital watermarking is a method used for copyright protection and authentication. Reversible digital watermarking is a technique to losslessly embed and retrieve secret information in a cover image, where after watermark extraction, both the watermark and the cover medium remain unmodified, bit-by-bit.

Here, the reversible colour image watermarking algorithm, in YCoCg colour space, is based on histogram bin shifting of prediction errors, using weighted mean based prediction technique to predict the pixel values. The transformation from the BGR to YCoCg-R colour space is reversible, with higher transform coding gain and near to optimal compression performance, as compared to BGR and other reversible colour spaces, resulting in considerably higher embedding capacity.

Reversible watermarking is most widely used in industries dealing with highly sensitive data, such as the military, medical and legal industries, where data integrity is the major concern for users.

Steps for colour image watermarking,

1. Read the colour image.

2. Convert the image into the YCoCg equivalent.

The image is split into the RGB channels. Suitable operations are performed to convert it into the Y, Co, Cg channels and then merged to give the YCoCg image.

3. The image is split into 3 channels.

4. The embedWatermarkInterolation function is called to embed watermark into each channel of the image.

(i) Call the Interpolated function

The interpolated matrix (x) is created.

- The margins, and the base pixels have the same value as the original matrix.
- The matrix is interpolated by finding mean of the means along the 45 degree and 135 degree line, and then finding its standard deviation.
- The same is done along the 0 degree and 90 degree lines.

The error matrix (e) is created by subtracting the interpolated matrix from the original matrix.

(ii) Call the createHistogram function.

- The unique elements in the error matrix are found.
- The max and min error are found.
- The error histogram is created. (For each unique element of the error matrix, the count is noted.)
- LM = max_count; RM = second_max_count;

(iii) Bitstream to be embedded is created.

- Pixel values of 0, 1, 254, 255 are noted.
  len_locmap = no. of such values
  len_locmap may be larger than the actual size of the locmap since it also contains those pixels which have become 1 or 255 after embedding.
- Maximum embedding capacity is found.
  cnt = LM + RM;
  Maximum embedding capacity: = cnt - 18 - len_locmap;

(iv) Embedding the watermark

- If value of pixel of error matrix = 999, do not embed (watermarked image = original image)
- If (error == LM or RM)
  e_new.at<float>(i,j) = err.e.at<float>(i,j) + sign_e * b;
  sign_e = 1 or -1 depending on whether error is < LM or > RM
  b = bit to be watermarked if there is place to be watermarked, else 0;
- If (error < LM or error > RM)
  Embed 1
- Watermarked image = Interpolated matrix (x) - Newly created error matrix after embedding watermark (e_new)
- Location map creation:
  If Value of pixel of original image matrix = 0, locmap = 0;
  If Value of pixel of watermarked image matrix = 0 or 255, locmap = 1;

(v) Create overhead

- Create 8 bit binary representation of LM and RM.
  RM_bin (9) = concat (sign of LM, binary representation of LM)
  RM_bin (9) = concat (sign of RM, binary representation of RM)
- overhead = Concatenate binary reps of LM and RM and locmap
  Length = 9 + 9 + locap.size()

5. The watermarked image is obtained in YCoCg and is converted to RGB.

6. The watermarked image is converted to YCoCg and the extractWatermarkInterpolation function is called to extract the watermark and get the original image, channel wise.

(i) Call the Interpolate function.

The interpolated matrix (x) and error matrix (e) are created.

(ii) Calculate LM and RM.

The first nine bits of the overhead gives LM (1st bit gives sign and the next 8 bits give the binary representation of LM, which is converted to decimal). The next 9 bits give RM (10th bit sign, the next 8, RM)

(iii) The watermark is extracted.

7. The extracted image is obtained in YCoCg and converted to RGB.

8. The extracted image and the original image are compared to check if they match bit by bit. The extracted watermark and the original watermark are compared to check if they match bit by bit. If they do, reversible digital watermarking has been successfully achieved.

This algorithm was implemented using OpenCV C++.

Algorithm 1. EMBED WATERMARK /* Embed watermark bits into the prediction errors */

Input: Colour cover image of size M × N pixels in YCoCg-R colour space (I), Watermark bits (W), Embedding Threshold (T)
Output: Watermarked image Iwm in the YCoCg-R colour space

1: for Colour channels P ∈ {Co, Cg, Y } in order do
2: if W is not empty then
3: for i = 1 to M do
4: for j = 1 to N do
5: if P (i, j) is not a base pixel then
6: P (i, j) ← P redictweightedmeanP (i, j)
7: Compute prediction error eP (i, j) = P (i, j) − P (i, j)
8: if eP (i, j) ≥ 0 then
9: sign(eP (i, j)) ← 1
10: else
11: sign(eP (i, j)) ← −1
12: end if
13: if |eP (i, j)| ≤ T then
14: e P (i, j) ← sign(eP (i, j)) × [2 × |eP (i, j)| + next bit of W]
15: else
16: e P (i, j) ← sign(eP (i, j)) × [|eP (i, j)| + T + 1]
17: end if
18: Pwm(i, j) ← P (i, j) + e P (i, j)

19: else
20: Pwm(i, j) ← P (i, j)
21: end if
22: end for 2
3: end for
24: end if
25: end for
26: Obtain watermarked image Iwm by combining the watermarked colour channels Ywm, Cowm and Cgwm.

Algorithm 2. EXTRACT WATERMARK /* Extract watermark bits from the prediction errors */

Input: Color watermarked image of size M × N pixels in YCoCg-R color space (Iwm), Embedding Threshold (T)
Output: Retrieved cover image (Iret), Watermark (W)

1: for Color channels P ∈ {Co, Cg, Y } in order do
2: for i = 1 to M do
3: for j = 1 to N do
4: if Pwm(i, j) is not a base pixel then
5: P wm(i, j) ← P redictweightedmeanPwm(i, j)
6: Compute prediction error ePwm(i, j) ← Pwm(i, j) − P wm(i, j)
7: if ePwm(i, j) ≥ 0 then
8: sign(ePwm(i, j)) ← 1
9: else
10: sign(ePwm(i, j)) ← −1
11: end if
12: if |ePwm(i, j)| ≤ (2T + 1) then
13: (Next bit of W) ← |ePwm(i, j)| − 2 × |ePwm(i,j)| 2
14: e Pwm(i, j) ← sign(ePwm(i, j)) × |ePwm(i,j)| 2
15: else
16: e Pwm(i, j) = sign(ePwm(i, j)) × [|ePwm(i, j)| − T − 1]
17: end if
18: Pret(i, j) = P wm(i, j) + e Pwm(i, j)
19: else
20: Pret(i, j) = Pwm(i, j)
21: end if
22: end for
23: end for
24: end for
25: Obtain original cover image Iret in YCoCg-R color space by combining the Yret, Coret and Cgret color components

For parallelized code

Parallel computing is a type of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved at the same time. In computer architecture, multithreading is the ability of a central processing unit (CPU) or a single core in a multi-core processor to execute multiple processes or threads concurrently, appropriately supported by the operating system.

Amdahl's law, also known as **Amdahl's** argument, is used to find the maximum expected improvement to an overall system when only part of the system is improved. It is often used in parallel computing to predict the theoretical maximum speed-up using multiple processors.

A task executed by a system whose resources are improved compared to an initial similar system can be split up into two parts:

- a part that does not benefit from the improvement of the resources of the system;
- a part that benefits from the improvement of the resources of the system.

According to the original statement of Amdahl's Law, the potential speedup gained by parallel execution of a program is limited by the portion that can be "parallelized".

Let P = ideal speedup (i.e., number of concurrent processors)
Let F = Fraction of sequential program that can be parallelized.

Then *maximum* speedup =

1 / ((1-F) + (F/P)).

This program for reversible digital watermarking can be divided into three parts:

- Interpolation
- Embedding
- Extraction

The interpolation part can be parallelized. This is because interpolation of each pixel of the original image to construct the interpolated matrix is independent of the others. Hence, the matrix is divided into a number of blocks (depending on the

number of threads available) and each of the blocks is executed independently, to create the interpolated matrix. This results in speed- up of the program.

However, for the **embedding and extraction parts**, a location map needs to be created, which needs to be updated every time a watermark bit is embedded/extracted. Since there is a dependency throughout the loop during the embedding/extraction process, these **loops cannot be parallelized**.

However, according to Amdahl's Law, the speed up will increase with increase in number of threads but will reach a saturation point once the critical number of threads have been used.

For interpolation part:

Image: Lena

Parallelized (4 threads)	0.37842	0.294319	0.248803
Un-parallelized	0.437271	0.37356	0.438043
Speed up	**13.4587%**	**21.234%**	**43.3452%**

For the overall program:

Image	**Lena**	**Car**	**Mandrill**	**Zelda**
Parallelized (4 threads)	25.0015	30.4603	33.2294	21.8223
Parallelized (3 threads)	26.3878	31.9196	33.3927	22.1092
Parallelized (2 threads)	26.8965	31.7167	34.668	22.7616
Un-parallelized	28.3163	32.9	34.8873	23.2353
Speed up	**11.654%**	**4.1247%**	**4.9892%**	**8.2456%**

However, the average speed-up for the interpolation part of the parallelized program over the un-parallelized program is 26.012% using 4 threads.

The average overall speed up of the parallelized program over the un-parallelized program is 8% using 4 threads.

A decrease in overall speed-up is noticed for the execution of the entire program as the speed-up is limited by the serial part of the program.

Code snippet for **parallel for loop** using OpenCV C++**:**

```
This snippet initialises the passed array with value 'flag'.

cvSetNumThreads(threads);

class Ones_init : public cv:: ParallelLoopBody
{
	private:
		int i , flag;
		double **matrix;
	public:
		Ones_init( double **m, int ii, int flagg) : matrix(m), i(ii),
flag(flagg)  {}

	 void operator()( const cv::Range &r )  const
	{
		int j;

		for( j = r.start; j < r.end; ++j)
			{
				matrix[i][j]=flag;
			}
	 }
};

The parallel for loop is called from the calling function in the following way:

	  for (i = 0; i<nr; i++)
			parallel_for_(cv::Range(0,nc), Ones_init( img_x, i, -1) );
```

Here, the **cvSetNumThreads** functions sets the number of threads to be used.

A class 'Ones_init' is created where the parameters to be operated on are passed. 'Range' signifies the range of values till which the loop is to be executed, 'img_x' is the matrix in which values are to be filled, 'i' signifies the row number, and the last parameter is the value with which the matrix is to be initialised.

Code snippet for **a regular for loop** in OpenCV C++:

```
This snippet initialises the array with value 'flag'.

for (i = 0; i<nr; i++)
	{
		for( j = 0; j < nc; ++j)
		{
			matrix[i][j]=flag;
		}
	}
```

Here, the program is executed serially.

PARALLELIZED REVERSIBLE DIGITAL WATERMARKING OPENCV

```
#include "opencv2/core/core.hpp"
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include "vector"
#include "math.h"
#include "stdlib.h"
#include "iostream"
#include "stdio.h"
#include "omp.h"

using namespace cv;
using namespace std;

const int threads= 4;
int LN, RN;
const double huge_val = 999;

struct histograms
{
	double error;
	int count;
};

struct Mat_overhead
{
	Mat watermarked_image;
	vector<int> overhead;
	int len_wm;
};

struct LM_RM
{
	int LM;
	int RM;
	Mat error_histogram;
};

struct errors
{
	Mat x;
	Mat e;
};

int rounding( double value )
{
    double intpart, fractpart;
    fractpart = modf(value, &intpart);
       //for +ve numbers, when fraction is 0.5, odd numbers are rounded up
    //and even numbers are rounded down
    //and vice versa for -ve numbers
       if (value > 0)
       {
              if(fractpart<0.5)
                     return (int)intpart;
              else
                     return (int)(intpart+1);
       }
       else
       {
              if(fractpart>-0.5)
```

```
                        return (int)intpart;
                else
                        return (int)(intpart-1);
        }
}

string type2str(int type) {
  string r;

  uchar depth = type & CV_MAT_DEPTH_MASK;
  uchar chans = 1 + (type >> CV_CN_SHIFT);

  switch ( depth ) {
    case CV_8U:  r = "8U"; break;
    case CV_8S:  r = "8S"; break;
    case CV_16U: r = "16U"; break;
    case CV_16S: r = "16S"; break;
    case CV_32S: r = "32S"; break;
    case CV_32F: r = "32F"; break;
    case CV_64F: r = "64F"; break;
    default:     r = "User"; break;
  }

  r += "C";
  r += (chans+'0');

  return r;
}

class Means1 : public cv:: ParallelLoopBody
{
        private:
                Mat p;
                int i;
                double **i_x;        double **i_e;

        public:

        Means1 (double **i_xx, double **i_ee, int ii, Mat pp) : i_x(i_xx), i_e(i_ee),
i(ii), p(pp){}
        //      void operator()( const cv::Range &r )  const {}
         void operator()( const cv::Range &r )  const
        {
                int j; double v;
                double x_45, x_135, u, sigma_e45, sigma_e135, w45, w135;

                double S45_1, S45_2, S45_3,  S135_1, S135_2, S135_3;

                for( j = r.start; j < r.end; ++j)
                {
                         x_45 = ( i_x[2*i][2*j+2] + i_x[2*i+2][2*j] ) / 2;
                          x_135 = ( i_x[2*i][2*j] + i_x[2*i+2][2*j+2] ) / 2;
                          u =( x_45+x_135)/2;
                         // cout<<"index: "<<2*i+1<<" , "<<2*j+1<<endl;
                          S45_1 = i_x[2*i][2*j+2];
                          S45_2=x_45;
                          S45_3= i_x[2*i+2][2*j];
                          S135_1= i_x[2*i][2*j];
                          S135_2= x_135;
                          S135_3= i_x[ 2*i+2][ 2*j+2];

sigma_e45 = ( (S45_1-u)*(S45_1-u) + (S45_2-u)*(S45_2-u) + (S45_3-u)*(S45_3-u) ) / 3;
```

```
sigma_e135 =((S135_1-u)*(S135_1-u) + (S135_2-u)*(S135_2-u) + (S135_3-u)*(S135_3-u))/3;
                        w45 = sigma_e135 / ( sigma_e45 + sigma_e135 );
                        w135 = 1 - w45;
                        if( (sigma_e45 == 0) && (sigma_e135 == 0) )
                           i_x[2*i+1][2*j+1] = u;
                     else
                           i_x[2*i+1][2*j+1] =rounding( w45*x_45 + w135*x_135 );

               i_e[2*i+1][2*j+1] = i_x[2*i+1][2*j+1] - p.at<double>((2*i+1),(2*j+1));
                       }
        }
};

class Means2 : public cv:: ParallelLoopBody
{
       private:
              Mat s;
              int i;
              double **i_x;
              double **i_e;

       public:
     Means2(Mat ss, int ii, double **i_xx, double **i_ee) : s(ss), i(ii), i_x(i_xx),
i_e(i_ee) {}

        void operator()( const cv::Range &r )  const
       {
              int j;
              double S0_1, S0_2, S0_3,  S90_1, S90_2, S90_3; double x_0, x_90, u,
sigma_e0, sigma_e90, w0, w90;

              for( j = r.start; j < r.end; ++j)
              {
                      if( (i+j)%2 == 1 )
                      {
                                  x_0 = (i_x[i][j-1] + i_x[i][j+1] ) / 2;
                                  x_90 = ( i_x[i-1][j] + i_x[i+1][j] ) / 2;
                                   u = ( i_x[i][j-1] + i_x[i][j+1] + i_x[i-1][j] +
i_x[i+1][j] ) / 4;

                                   S0_1 = i_x[i][j-1];
                                   S0_2= x_0;
                                   S0_3= i_x[i][j+1];
                                   S90_1 = i_x[i-1][j];
                                   S90_2= x_90;
                                   S90_3 = i_x[i+1][j];

     sigma_e0 = ( (S0_1 -u)*(S0_1 -u) + (S0_2 -u)*(S0_2 -u) + (S0_3 -u)*(S0_3 -u) )/3;
     sigma_e90 =( (S90_1-u)*(S90_1-u) + (S90_2-u)*(S90_2-u) + (S90_3-u)*(S90_3-u))/3;
                                  w0 = sigma_e90 / ( sigma_e0 + sigma_e90 );
                                  w90 = 1 - w0;

                                  if( (sigma_e0 == 0) && (sigma_e90 == 0) )
                                           i_x[i][j] = u;
                                  else
                                         i_x[i][j] = rounding(( w0*x_0 + w90*x_90 ));
                                  i_e[i][j] = i_x[i][j] - s.at<double>(i,j) ;
                           }
                     }
       }
};
```

```
class Equality : public cv:: ParallelLoopBody
{
	private:
		int i , nr, nc;
		double **matrix;
		Mat p;
	public:
		Equality( double **m, int ii, int nrr, int ncc, Mat pp) : matrix(m),
i(ii), nr(nrr), nc(ncc), p(pp)  {}

	 void operator()( const cv::Range &r )  const
	{
		int j;
		for( j = r.start; j < r.end; ++j)
			{
				if( ( i == 0 ) || ( j == 0) || ( i == nr-1 ) || ( j == nc-
1 ) )
				matrix[i][j]=p.at<double>(i,j);

			}
	 }
};

class DownSample : public cv:: ParallelLoopBody
{
	private:
		int i;
		double **matrix;
		Mat p;
	public:
		DownSample( double **m, int ii, Mat pp) : matrix(m), i(ii), p(pp)  {}

	 void operator()( const cv::Range &r )  const
	{
		int j;
		for( j = r.start; j < r.end; ++j)
			{
				if ( ( i%2 == 0 ) && (j%2 == 0 ) )
				matrix[i][j]=p.at<double>(i,j);
			}
	 }
};

class Ones_init : public cv:: ParallelLoopBody
{
	private:
		int i , flag;
		double **matrix;
	public:
		Ones_init( double **m, int ii, int flagg) : matrix(m), i(ii),
flag(flagg)  {}

	 void operator()( const cv::Range &r )  const
	{
		int j;

		for( j = r.start; j < r.end; ++j)
			{
				matrix[i][j]=flag;
			}
	 }
};
```

```
errors  Interpolated( Mat p)
{
       //Interpolate

       int i, j;
         int nr= p.size().height;
         int nc= p.size().width;
              const int a=nr;
              const int b= nc;
        double **img_x= new double*[a];
        double **img_e= new double* [a];

              for( i = 0; i < 512; ++i)
              {
                     img_x[i] = new double[b];
                     img_e[i] = new double[b];
              }

         errors error_em;

         cvSetNumThreads(threads);

          error_em.x = Mat_<double>(nr,nc);
          //error_em.x= -1 * Mat::ones(nr, nc, CV_64FC1);

          for (i = 0; i<nr; i++)
                     parallel_for_(cv::Range(0,nc), Ones_init( img_x, i, -1) );

          //Create Margin

           for (i = 0; i<nr; i++)
                     parallel_for_(cv::Range(0,nc), Equality( img_x, i, nr, nc, p) );

          // Down Sample => Get the Low Resolution Pixels
              for (i = 0; i<nr;i++)
              parallel_for_(cv::Range(0,nc), DownSample( img_x, i, p) );

              //Error Matrix

           error_em.e = Mat_<double>(nr,nc);
              //error_em.e = huge_val* Mat::ones( nr, nc, CV_64FC1);
              for (i = 0; i<nr; i++)
                     parallel_for_(cv::Range(0,nc), Ones_init( img_e, i, 999) );

              // STEP I (center high resolution pixels)

              for (i = 0; i<(nr/2 - 1); i++)
               parallel_for_( cv::Range(0,(nr/2-1)), Means1(img_x, img_e, i, p) );

               // STEP II (residual high resolution pixels)

              //cout<<" NO > OF THREDS: "<<getNumThreads();
              for (i = 1; i<(nr-1); i++)
                      parallel_for_( cv::Range(1,(nr-1)), Means2(p, i , img_x, img_e)
);

              for( i=0; i < nr; i++)
              {
                     for( j=0; j < nc; j++)
                     {
                            error_em.x.at<double>(i,j) = img_x[i][j];
```

```
                        error_em.e.at<double>(i,j) = img_e[i][j];
                    }
                }

        return error_em;
}

std::vector<double> unique(const cv::Mat& input, bool sort = false)
{
    std::vector<double> out;
    for (int y = 0; y < input.rows; ++y)
    {
        const double* row_ptr = input.ptr<double>(y);
        for (int x = 0; x < input.cols; ++x)
        {
            double value = row_ptr[x];

            if ( std::find(out.begin(), out.end(), value) == out.end() )
                out.push_back(value);
        }
    }

    if (sort)
        std::sort(out.begin(), out.end());
    return out;
}


LM_RM createHistogram(Mat e_hist)
{
       int i, j, k;
       int nr= e_hist.size().height;
       int nc= e_hist.size().width;

        double max_err = 0;
        double min_err = 0;

       LM_RM lm_rm;

       lm_rm.error_histogram= Mat_<double>(nr, nc);

       //Calculating unique element
       std::vector<double> unik = unique (e_hist, true);

       int histSize = unik.size();

       Mat x_hist = Mat_<double>(nr,nc);
       for( i=0; i < nr; i++)
       {
              for( j = 0; j < nc; j++)
              x_hist.at<double>(i,j)=e_hist.at<double>(i,j);
       }
       //x_hist=e_hist;
       Mat new_hist = Mat_<double>(nr,nc);
       new_hist=e_hist;

       // Finding max and min error

       for (i = 0; i< nr; i++)
       {
              for (j = 0; j<nc; j++)
              {
```

```
                if( e_hist.at<double>(i,j) == huge_val )
                x_hist.at<double>(i,j) = 0;
                else if( e_hist.at<double>(i,j) > max_err )
                max_err = e_hist.at<double>(i,j);
                 else if( e_hist.at<double>(i,j) < min_err )
                  min_err = e_hist.at<double>(i,j);
        }

}

 histograms *histogram= new histograms[histSize];

 for (i = 0; i<unik.size(); i++)
 {
        histogram[i].error = unik[i];
        histogram[i].count = 0;
 }

 //Create histogram
 for (i = 0; i<nr; i++)
 {
        for (j = 0; j<nc; j++)
        {
                if( e_hist.at<double>(i,j) == huge_val )
     ;
                else
                {
                        for (k = 0; k<unik.size(); k++)
                        {
                                if( e_hist.at<double>(i,j) == histogram[k].error )
              histogram[k].count = histogram[k].count + 1;
                        }
                }
        }
 }
 for(i=0;i<nr; i++)
 {
        for(j=0;j<nc;j++)
                lm_rm.error_histogram.at<double>(i,j)=e_hist.at<double>(i,j);
 }

 //Compute LM RM

int max_count = -1;
for (i = 0; i<unik.size(); i++)
{
       //X(i) = histogram(1,i).error; % For Plot
       // Y(i) = histogram(1,i).count; % For Plot
        if( histogram[i].count > max_count )
        {
                max_count = histogram[i].count;
                lm_rm.LM = histogram[i].error;
        }
}

int second_max_count = -1;
for (i = 0; i<unik.size(); i++)
{
       if( histogram[i].count == max_count )
 ;
       else
              {
```

```
                        if( histogram[i].count > second_max_count )
                        {
                                second_max_count = histogram[i].count;
                                lm_rm.RM = histogram[i].error;
                        }
                }
        }

        /*
        //Calculate LE and RE
        histograms *LE= new histograms[unik.size()] ;
        histograms *RE= new histograms[unik.size()];
        int posLE = 0;
        int posRE = 0;
    for (i = 0; i<unik.size(); i++)
        {
                if( histogram[i].error <= lm_rm.LM )
                {
                        posLE = posLE + 1;
                        LE[posLE].error = histogram[i].error;
                        LE[posLE].count = histogram[i].count;
                }
                if( histogram[i].error >= lm_rm.RM )
                {
                        posRE = posRE + 1;
                        RE[posRE].error = histogram[i].error;
                        RE[posRE].count = histogram[i].count;
                }
        }
        //Compute LN RN
        int mincount_LE = LE[1].count;
        LN = LE[1].error;
        for (j = 0; j<posLE; j++)
        {
                if( LE[j].count < mincount_LE )
                {
                        mincount_LE = LE[j].count;
                  LN = LE[j].error;
                }
        }
        int mincount_RE = RE[1].count;
        RN = RE[1].error;
    for (j = 0; j<posRE; j++)
        {
                if( RE[j].count < mincount_RE )
                {
                        mincount_RE = RE[j].count;
                        RN = RE[j].error;
                }
        }
        cout<<"LN: "<<LN<<"RN: "<<RN<<endl;
        for"(i=0;i<unik.size();i++)
        {       cout<<"LE error: "<<LE[i].error<<" LE count: "<<LE[i].count<<endl;
                cout<<"RE error: "<<RE[i].error<<" RE count: "<<RE[i].count<<endl;
        }  */

                return lm_rm;
}

vector<int> EightBits(int I)
{
        vector<int> bitstring(8); int i=0;
```

```
	//[bitstring] = EightBits(I) converts integer I into its 8-bit binary
	//encoding, bitstring.
	//Example, EightBits(2) returns bitstring = 00000010
	//Input:   I: an integer
	//Output:  bitstring: a 1X8 array of 0s and 1s


	//binary encoding of I consisting of 8 or lesser number of bits
	//bin = dec2bin(I8);
	if(I<0)
	I=-I;
	int bin[8];

	while(I>0)
	{
		bin[i]=I%2;
		I=I/2;
		i++;  //will signify no. of bits occupied
	}

	int j;
	//If number of bits < 8, shift each bit right so that the rightmost bit
occupies the 8th position.

	//Shift each bit of bin right by (8 - nc) positions
	 for (j = 0; j<i; j++)
		bitstring[ 7-j] = bin[j];
	 //Fill in the remaining (8 - nc) left bits by zeros.
    for (j = 1; j<(8 - i); j++)
	 bitstring[j] = 0;

	 return bitstring;
}

Mat_overhead EmbedWatermarkInterpolation(Mat p, vector<int> &watermark_original)
{
	int i, j;
	int nr= p.size().height;
	int nc= p.size().width;
	LM_RM lm_rm;
	errors err;

	vector<int> watermark;   //to be embedded
	//int64 t= getTickCount();
	err = Interpolated(p);

	lm_rm=createHistogram(err.e);

	 Mat e_new= Mat_<double>(nr,nc);
	 for( i =0 ; i < nr; i++)
	 {
		for( j=0; j < nc; j++)
			e_new.at<double>(i,j)=err.e.at<double>(i,j);
	 }

	//Create bitstream to be embedded-----------------------------------------
    //Location Map for overflow or underflow.

		int len_locmap = 0;  //Total no. of pixels with values 0, 255, 1, 254.

	for (i = 0; i<nr; i++)
	{
```

```
		for (j = 0; j<nc; j++)
		{
			//Only 254 or 1 can change to 255 and 0 respectively
			 // So we take max length (for 0,255,1,254)
			 // actual loc map consists of those pixel positions which are
either
			 // 0, 255 or have changed to 0, 255.
			 //  So actual length may be smaller than len_locmap.

			if( (p.at<double>(i,j) == 0 ) || ( p.at<double>(i,j) == 255 ) ||
( p.at<double>(i,j) == 1 ) || ( p.at<double>(i,j) == 254 ) )
			len_locmap = len_locmap + 1;
		}
	}

	Mat_overhead ob1;
	ob1.watermarked_image= Mat_<double>(nr,nc);
	//watermarked_image = zeros(nr,nc);
	 for( i =0 ; i < nr; i++)
	 {
		 for( j=0; j < nc; j++)
			 ob1.watermarked_image.at<double>(i,j)=p.at<double>(i,j);
	 }

	int poswm = 0; // To keep track of bits to be embedded
	int cnt = 0; // To find Maximum Embedding capacity
	vector<int> locmap; // To prevent overflow/underflow

	// Get Maximum Capacity-------------

	for( i=0; i<nr; i++)
	{
		for (j = 0; j<nc; j++)
			{
				if( ( p.at<double>(i,j) == 0 ) || ( p.at<double>(i,j) ==
255 ) ) ;

				else
				{
					if( err.e.at<double>(i,j) == huge_val ) ;
				    else if( err.e.at<double>(i,j) <= (double)lm_rm.LM )
						{
							if( err.e.at<double>(i,j) ==
(double)lm_rm.LM )

							cnt = cnt + 1;
						}
					 else if( err.e.at<double>(i,j) >= (double)lm_rm.RM
)
						 {
							if( err.e.at<double>(i,j) ==
(double)lm_rm.RM )

							cnt = cnt + 1;
						 }
				}
			}
	}

	if( watermark_original.size() >( cnt - 18 - len_locmap ) ) //original watermark
	{
		for( i=0; i < ( cnt - 18 - len_locmap ) ; i++)
			watermark.push_back(watermark_original[i]);
	}
```

```
    ob1.len_wm = watermark.size();

       //Store just 18 bits for 2 signed integers along with 'len_locmap' bits for
overflow/underflow.
       int k = 1; double v;
       for (i = 0; i<nr; i++)
       {
              for (j = 0; j<nc; j++)
              {
                     if( (i == 0) || (j == 0) || (i == 7) || (j == 7) )
                     {
                            if( ( i%2==0) && (j%2==1) )
                            {
                                   if( k <= ( 18 + len_locmap ) )
                                   {
                                          if (p.at<double>(i,j)<0)
                                                 v=-p.at<double>(i,j);
                                          else
                                                 v=p.at<double>(i,j);

                                          watermark.push_back( ( (int) (v ) ) %2 );
//LSB= (int)(p.at<double>(i,j))%2
                                      k = k + 1;
                                   }
                            }
                            else if( (i%2==1) && (j%2==0) )
                                    {
                                          if( k <= ( 18 + len_locmap ) )
                                          {
                                           if (p.at<double>(i,j)<0)
                                             v=-p.at<double>(i,j);
                                           else
                                                 v=p.at<double>(i,j);

                                                 watermark.push_back( ( (int) (v ) )%2
);
                                                 k = k + 1;
                                          }
                                   }
                     }
              }
       }

// Embed---------------------------------------------------------------------

       int ncwm = watermark.size(); int b, sign_e;

       for (i = 0;i<nr; i++)
       {
              for (j = 0; j<nc; j++)
              {
                     if( ( p.at<double>(i,j) == 0 ) || ( p.at<double>(i,j) == 255 ) )
             locmap.push_back(0);
             else
                     {
                            if( lm_rm.error_histogram.at<double>(i,j) == huge_val )
// Do not Embed (Low resolution or marginal pixels)
                ob1.watermarked_image.at<double>(i,j) = p.at<double>(i,j);
                         else if( lm_rm.error_histogram.at<double>(i,j) <= lm_rm.LM )
                            {
                                   sign_e = -1;   // sign(e) = -1 if e in LE i.e. <=
LM
```

```
                                if( lm_rm.error_histogram.at<double>(i,j) ==
lm_rm.LM ) // Embed next bit
                                        {
                                                //cnt = cnt + 1;
                                                if( poswm < ncwm )  // More bits to
embed
                                                        {
                                                                poswm = poswm + 1;
                                                                b = watermark[poswm-1];
                                                        }
                                                 else
                                                    b = 0;  // No more bits => e will
be same
                                                e_new.at<double>(i,j) =
lm_rm.error_histogram.at<double>(i,j) + sign_e * b;
                                        }
                                else if( lm_rm.error_histogram.at<double>(i,j) <
lm_rm.LM ) // Embed 1
                                        e_new.at<double>(i,j) =
lm_rm.error_histogram.at<double>(i,j) + sign_e * 1;

                                 ob1.watermarked_image.at<double>(i,j) =
err.x.at<double>(i,j) - e_new.at<double>(i,j);
                                        // update locmap
                                if( ( ob1.watermarked_image.at<double>(i,j) == 0 )
|| ( ob1.watermarked_image.at<double>(i,j) == 255 ) )
                        locmap.push_back( 1 );
                        }
                        else if( lm_rm.error_histogram.at<double>(i,j) >= lm_rm.RM
)
                        {
                                sign_e = 1;  //  sign(e) = 1 if e in RE i.e. <= RM
                                 if( lm_rm.error_histogram.at<double>(i,j) ==
lm_rm.RM )  // Embed next bit
                                        {
                                                //cnt = cnt + 1;
                                                if( poswm < ncwm )  // More bits to
embed
                                                {
                                                        poswm = poswm + 1;
                                                        b = watermark[poswm-1];
                                                }
                                                else
                                                        b = 0;  // No more bits => e
will be same

                                                e_new.at<double>(i,j)
=lm_rm.error_histogram.at<double>(i,j) + sign_e * b;
                                        }
                                 else if( lm_rm.error_histogram.at<double>(i,j) >
lm_rm.RM )  // Embed 1
                                   e_new.at<double>(i,j) =
lm_rm.error_histogram.at<double>(i,j) + sign_e * 1;

                ob1.watermarked_image.at<double>(i,j) = err.x.at<double>(i,j) -
e_new.at<double>(i,j);
                //update locmap
                if( ( ob1.watermarked_image.at<double>(i,j) == 0 ) || (
ob1.watermarked_image.at<double>(i,j) == 255 ) )
                    locmap.push_back(1);
                        }
                 }
```

```
                }
        }

        // GET binary representations of LM and RM----------------
        vector<int> temp = EightBits(lm_rm.LM);
        vector<int> LM_bin(9);
        for(i=0 ; i<9; i++)
                LM_bin[i]=0;
        if( lm_rm.LM < 0 )
                LM_bin[0] = 1;
        else
                LM_bin[0] = 0;
        for (i = 0; i<8; i++)
        {
                if( temp[i] == 0 )
                        LM_bin[i+1] = 0;
                else if( temp[i] == 1 )
                        LM_bin[i+1] = 1;
        }

        temp = EightBits(lm_rm.RM);
        vector<int>RM_bin(9);
        for(i=0 ; i<9; i++)
                RM_bin[i]=0;
        if( lm_rm.RM < 0 )
                RM_bin[0] = 1;
        else
                RM_bin[0] = 0;
        for (i = 0; i<8; i++)
        {
                if( temp[i] == 0 )
                        RM_bin[i+1] = 0;
                else if( temp[i] == 1 )
                        RM_bin[i+1] = 1;
        }

        // Concatenate binary reps of LM and RM and locmap
        vector<int> b_vect;

        for(i=0; i<9;i++)
        b_vect.push_back(LM_bin[i]);

        for(i=0; i<9;i++)
        b_vect.push_back(RM_bin[i]);

        for(i=0; i<locmap.size();i++)
        b_vect.push_back(locmap[i]);

        ob1.overhead = b_vect;

        return ob1;
}

int bi2de(vector<int> bin)
{
        int i=7, dec = 0, rem, num, base = 1;

    while (i>=0)
    {
       dec = dec + bin[i] * base;
       base = base * 2;
        i--;
```

```
    }
    return dec;
}

Mat YCoCg2RGB( Mat watermarked_image_ee)
{
       int nr= watermarked_image_ee.size().height;
       int nc= watermarked_image_ee.size().width;
       watermarked_image_ee.convertTo(watermarked_image_ee, CV_64FC3);

       // Split into Y, Co, Cg
       Mat Y = Mat_<double>(nr, nc);
       Mat Co = Mat_<double>(nr, nc);
       Mat Cg = Mat_<double>(nr, nc);
       Mat rgb_image;
       Mat YCoCg[3];

       split(watermarked_image_ee, YCoCg);
       Y= YCoCg[0];
       Co = YCoCg[1];
       Cg= YCoCg[2];

         Mat R = Mat_<double>(nr, nc);
         Mat G = Mat_<double>(nr, nc);
         Mat B = Mat_<double>(nr, nc);
         Mat t = Mat_<double>(nr, nc);

         int i, j;

         for(i=0;i<nr;i++)
         {
                for (j=0;j<nc; j++)
                {      t.at<double>(i,j) =  (double)(Y.at<double>(i,j)) -
(double)(floor)(0.5*Cg.at<double>(i,j));
                       G.at<double>(i,j) = (double)(Cg.at<double>(i,j)) +
(double)(t.at<double>(i,j));
                       B.at<double>(i,j) = (double)(t.at<double>(i,j)) -
(double)(floor)(0.5*Co.at<double>(i,j));
                       R.at<double>(i,j) = (double)(B.at<double>(i,j)) +
(double)(Co.at<double>(i,j));
                }
         }

        Mat YCoCg_new[3];
        YCoCg_new[0] = B;
        YCoCg_new[1] = G;
        YCoCg_new[2] = R;

         merge( YCoCg_new,3, rgb_image);

          return rgb_image;
}

Mat RGB2YCoCg( Mat images)
{
       int nr= images.size().height;
       int nc= images.size().width;
       images.convertTo(images, CV_64FC3);
       //Split into R, G, B
       Mat b=Mat_<double>(nr, nc);
       Mat g=Mat_<double>(nr, nc);
       Mat r=Mat_<double>(nr, nc);
```

```
	Mat arr[3];
	split(images, arr);                    //GIVES Lesser time and better result
	b= arr[0];
	g= arr[1];
	r= arr[2];

	  Mat Co= Mat_<double>(nr, nc);
	  Mat t= Mat_<double>(nr, nc);
	  Mat Cg= Mat_<double>(nr, nc);
	  Mat Y= Mat_<double>(nr, nc);

	  int i,j;
	  for(i=0;i<nr;i++)
	  {
		for (j=0;j<nc; j++)
		{
			Co.at<double>(i,j) = (double)(r.at<double>(i,j)) -
(double)(b.at<double>(i,j));
		t.at<double>(i,j) = (double)(b.at<double>(i,j)) +
(double)(floor)(0.5*Co.at<double>(i,j));
		Cg.at<double>(i,j) = (double)(g.at<double>(i,j)) -
(double)(t.at<double>(i,j));
		Y.at<double>(i,j) = (double)(t.at<double>(i,j)) +
(double)(floor)(0.5*Cg.at<double>(i,j));
		}
	  }

	   Mat arr_new[3];
	  arr_new[0]=Y;
	  arr_new[1]=Co;
	  arr_new[2]=Cg;

	   Mat colors;

	    merge( arr_new,3, colors);

	    return colors;
}

Mat extractWatermarkInterpolation( Mat watermarked_image_e, int l, vector<int>
overheads, vector<int> wmm)
{
	int nr= watermarked_image_e.size().height;
	int nc= watermarked_image_e.size().width;
	int i, j;
	errors errs;

	errs= Interpolated(watermarked_image_e);
	long long sums1=0;
	for( i=0; i<nr; i++)
	{
		for(j=0;j<nc;j++)
			sums1+=errs.e.at<double>(i,j);
	}

	Mat e_new  = Mat_<double>(nr,nc);
	Mat xxx  = Mat_<double>(nr,nc);
	Mat eee = Mat_<double>(nr,nc);

	for(i=0; i<nr; i++)
	{
		for(j=0; j<nc; j++)
```

```
        {
                e_new.at<double>(i,j) = errs.e.at<double>(i,j);
                eee.at<double>(i,j) = errs.e.at<double>(i,j);
                xxx.at<double>(i,j) = errs.x.at<double>(i,j);
        }
}

vector<int> watermark;
Mat  p=Mat_<double>(nr, nc);

for(i=0;i<nr;i++)
{
        for(j=0;j<nc;j++)
                p.at<double>(i,j) = watermarked_image_e.at<double>(i,j) ;
}
int LM_extract, RM_extract;

// Compute LM, RM

vector<int> LM_RM_e;
for(i=0;i<overheads.size();i++)
        LM_RM_e.push_back(overheads[i]);

std::vector<int>  temp(8);

if( LM_RM_e[0] == 1 )
{
        for(i=0;i<8;i++)
        temp[i]=overheads[i+1];
        LM_extract = (-1)*bi2de(temp);
}
else if( LM_RM_e[0] == 0 )
{
        for(i=0;i<8;i++)
        temp[i]=overheads[i+1];
        LM_extract = bi2de( temp );
}

if( LM_RM_e[9] == 1 )
{
        for(i=10;i<=17;i++)
                temp[i-10]=overheads[i];
        RM_extract = (-1)*bi2de( temp );
}
else if( LM_RM_e[9] == 0 )
{
        for(i=10;i<=17;i++)
                temp[i-10]=overheads[i];
        RM_extract = bi2de( temp);
}

// Compute Locmap
// locmap = LM_RM( 1, 19:( 19 + len_locmap - 1 ) );
vector<int> locmap(overheads.size()-18);
if( overheads.size() > 18 )
        {for( i=18; i<overheads.size(); i++)
        locmap[i-18]=overheads[i];
    }

// EXTRACT ---------------------------------------------------------------

int sign_e, b;
```

```
        Mat p1 = Mat_<double>(nr,nc);
           p1= -1 * Mat::ones(nr, nc, CV_64FC1);

        int pos_locmap = 0;


        for (i = 0; i<nr;i++)
        {
                for (j = 0; j<nc; j++)
               {
                         if( e_new.at<double>(i,j) == huge_val )
                              {
                                     //e(i,j) = 999;
                                     eee.at<double>(i,j) = 0;
                                     p1.at<double>(i,j) = p.at<double>(i,j);
                                  if( (p.at<double>(i,j)==0) || (p.at<double>(i,j)==255)
)
                                            pos_locmap = pos_locmap + 1;
                              }
             else if( e_new.at<double>(i,j) == LM_extract )
                    {
                           if( (p.at<double>(i,j)==0) || (p.at<double>(i,j)==255) )
                           {
                                  pos_locmap = pos_locmap + 1;
                                  if( locmap[pos_locmap-1] == 0 )
                                         p1.at<double>(i,j) = p.at<double>(i,j);
                                 else if( locmap[pos_locmap-1] == 1 )
                                       {
                                              sign_e = -1;
                                              b = 0;
                                             watermark.push_back(b);
                                             eee.at<double>(i,j) = e_new.at<double>(i,j) -
sign_e*b;
                                             p1.at<double>(i,j) = xxx.at<double>(i,j) -
eee.at<double>(i,j);
                                        }
                           }
                               else
                               {
                                      sign_e = -1;
                                      b = 0;
                                      watermark.push_back(b);
                                      eee.at<double>(i,j) = e_new.at<double>(i,j) -
sign_e*b;
                                      p1.at<double>(i,j) = xxx.at<double>(i,j) -
eee.at<double>(i,j);
                               }
                       }
            else if( e_new.at<double>(i,j) == RM_extract )
                  {
                         if( (p.at<double>(i,j)==0) || (p.at<double>(i,j)==255) )
                  {
                                      pos_locmap = pos_locmap + 1;
                                      if( locmap[pos_locmap-1] == 0 )
                                             p1.at<double>(i,j) = p.at<double>(i,j);
                                      else if( locmap[pos_locmap-1] == 1 )
                     {
                                             sign_e = 1;
                                             b = 0;
                                             watermark.push_back(b);
                                             eee.at<double>(i,j) = e_new.at<double>(i,j) -
sign_e*b;
```

```
                                        p1.at<double>(i,j) = xxx.at<double>(i,j) -
eee.at<double>(i,j);
                                }
                        }
                        else
                        {
                                sign_e = 1;
                                b = 0;
                                watermark.push_back(b);
                                eee.at<double>(i,j) = e_new.at<double>(i,j) -
sign_e*b;
                                p1.at<double>(i,j) = xxx.at<double>(i,j) -
eee.at<double>(i,j);
                        }
                }
                    else if( e_new.at<double>(i,j) == (LM_extract-1) )
                    {
                        if( (p.at<double>(i,j)==0) || (p.at<double>(i,j)==255) )
                        {
                                pos_locmap = pos_locmap + 1;
                                if( locmap[pos_locmap-1] == 0 )
                                        p1.at<double>(i,j) = p.at<double>(i,j);
                            else if( locmap[pos_locmap-1] == 1 )
                                {
                                        sign_e = -1;
                                        b = 1;
                                        watermark.push_back(b);
                                        eee.at<double>(i,j) = e_new.at<double>(i,j) -
sign_e*b;
                                        p1.at<double>(i,j) = xxx.at<double>(i,j) -
eee.at<double>(i,j);
                                }
                        }
                        else
                        {
                                sign_e = -1;
                                b = 1;
                                watermark.push_back(b);
                                eee.at<double>(i,j) = e_new.at<double>(i,j) -
sign_e*b;
                            p1.at<double>(i,j) = xxx.at<double>(i,j) -
eee.at<double>(i,j);
                         }
                    }
                    else if( e_new.at<double>(i,j) == (RM_extract+1) )
            {
                        if( (p.at<double>(i,j)==0) || (p.at<double>(i,j)==255) )
                {
                                pos_locmap = pos_locmap + 1;
                                if( locmap[pos_locmap-1] == 0 )
                                   p1.at<double>(i,j) = p.at<double>(i,j);
                                else if( locmap[pos_locmap-1] == 1 )
                                { sign_e = 1;
                                        b = 1;
                                        watermark.push_back(b);
                                        eee.at<double>(i,j) = e_new.at<double>(i,j) -
sign_e*b;
                                        p1.at<double>(i,j) = xxx.at<double>(i,j) -
eee.at<double>(i,j);
                                 }
                        }
                        else
```

```
                        {
                                sign_e = 1;
                                b = 1;
                                 watermark.push_back(b);
                                eee.at<double>(i,j) = e_new.at<double>(i,j) -
sign_e*b;
                                p1.at<double>(i,j) = xxx.at<double>(i,j) -
eee.at<double>(i,j);
                        }
                }

                else if( e_new.at<double>(i,j) < (LM_extract-1) )
        {
                        if( (p.at<double>(i,j)==0) || (p.at<double>(i,j)==255) )
            {
                                pos_locmap = pos_locmap + 1;
                                if( locmap[pos_locmap-1] == 0 )
                                        p1.at<double>(i,j) = p.at<double>(i,j);
                                else if( locmap[pos_locmap-1] == 1 )
                    {
                                        sign_e = -1;
                                        eee.at<double>(i,j) = e_new.at<double>(i,j) -
sign_e*1;
                                        p1.at<double>(i,j) = xxx.at<double>(i,j) -
eee.at<double>(i,j);
                                }
                        }
            else
                        {
                                sign_e = -1;
                    eee.at<double>(i,j) = e_new.at<double>(i,j) - sign_e*1;
                    p1.at<double>(i,j) = xxx.at<double>(i,j) - eee.at<double>(i,j);
                                }
                        }

               else if( e_new.at<double>(i,j) > (RM_extract+1) )
                        {
                                if( (p.at<double>(i,j)==0) || (p.at<double>(i,j)==255) )
                                 {
                                pos_locmap = pos_locmap + 1;
                                if( locmap[pos_locmap-1] == 0 )
                                        p1.at<double>(i,j) = p.at<double>(i,j);
                                else if( locmap[pos_locmap-1] == 1 )
                                {
                                        sign_e = 1;
                                         eee.at<double>(i,j) = e_new.at<double>(i,j)
- sign_e*1;
                                        p1.at<double>(i,j) = xxx.at<double>(i,j) -
eee.at<double>(i,j);
                                }
                        }
                        else
                        {
                                sign_e = 1;
                                 eee.at<double>(i,j) = e_new.at<double>(i,j) -
sign_e*1;
                                p1.at<double>(i,j) = xxx.at<double>(i,j) -
eee.at<double>(i,j);
                        }
                }
             }
       }
```

```
	vector<int> tempp(l);
	for(i=0; i<l;i++)
	tempp[i]=watermark[i];


	//int flag=0;
	//for(i=0;i<tempp.size();i++)
	//{
		//<<watermark[i];
	//	if(tempp[i]!=wmm[i])
	//		flag++;
	//}
	//cout<<"Extracted Watermark size: "<<tempp.size();
	//cout<<" Flag_watermark: "<<flag<<endl;

	return p1;
}

double MSE ( Mat matrix1, Mat matrix2)
{
	//   'nmse' gives the Mean Square Error of matrices 1 and 2
	int nr, nc, ns;  double nmse = 0, temp; int i, j, k;

	//Check dimensions of 2 matrices
	if ( matrix1.size() == matrix2.size() )
	{
		//Get dimensions of the matrices
		nr = matrix1.size().height;
		nc = matrix1.size().width;
		ns = matrix1.channels();
		for (i = 0; i < nr; i++)
		{
			for (j = 0; j < nc; j++)
				{
					for (k = 0; k < ns; k++)
						{
							if ( matrix1.at<Vec3d>(i,j)[k] ==
matrix2.at<Vec3d>(i,j)[k] )
							 ;
							else
							{
								temp =
matrix1.at<Vec3d>(i,j)[k] - matrix2.at<Vec3d>(i,j)[k];
								nmse = nmse + temp * temp;
							}
						}
				}
		}
	}

   else
		cout<< endl<< " Dimension mismatch in MSE calculation."<<endl; //Error
if dimensions mismatch
	nmse = nmse / (nr*nc*ns);
	return nmse;
}

void PSNR(Mat image_original, Mat image_modified )
{
	//[ psnr ] = PSNR( image_original, image_modified )
	//   Computes PSNR value of an image.
```

```
        //   Input: image_original -> Original image filename.
        //          image_modified -> Modified image filename.
        //   Output: psnr -> PSNR of the modified image (image_modified) w.r.t the
        //                   original image (image_original).

        double mse = MSE( image_original, image_modified );
        double psnr = 10 * log10( 255*255 / mse );

        cout<< endl << "Calculated PSNR = " << psnr;
}



int main(int argc, char **argv )
{
    Mat image;
        int i, j, k;
       image = imread("G:\\mandrill.tiff", CV_LOAD_IMAGE_COLOR);

       if(! image.data )
       {
              cout <<  "Could not open or find the image" << std::endl ;
              return -1;
       }

        namedWindow( "Display window", CV_WINDOW_AUTOSIZE );
       imshow( "Display window", image );
          int nor= image.size().height;
         int noc= image.size().width;
         image.convertTo(image, CV_64FC3);

        double t= getTickCount();
          Mat color = RGB2YCoCg (image);

        long long int summ1=0, summ2=0, summ3=0;

        //Create Channel

         Mat p1 = Mat_<double>(nor,noc);
         Mat p2 = Mat_<double>(nor,noc);
         Mat p3 = Mat_<double>(nor,noc);

         Mat_overhead o1, o2, o3;
         Mat watermarked_image_channels[3], watermarked_image_final;
         Mat rgb_image_watermarked, colored[3];


         split( color, colored);
         p1=colored[0];
         p2=colored[1];
         p3=colored[2];

          vector<int> watermarks(999999);     //Watermark in int

          for(i=0; i< 999999; i++)
                 watermarks[i]=rand() % 2 ;

         o1 = EmbedWatermarkInterpolation(p1, watermarks);  //Y_wm
         o2 = EmbedWatermarkInterpolation(p2, watermarks);  //Co_wm
         o3 = EmbedWatermarkInterpolation(p3, watermarks);  //Cg_wm
         watermarked_image_channels[0]= o1.watermarked_image;
         watermarked_image_channels[1] = o2.watermarked_image;
```

```
          watermarked_image_channels[2] = o3.watermarked_image;
          merge( watermarked_image_channels, 3, watermarked_image_final);

       rgb_image_watermarked= YCoCg2RGB(watermarked_image_final);

       //rgb_image_watermarked.convertTo(rgb_image_watermarked, CV_8UC3);
       //imshow("Watermarked image in RGB", rgb_image_watermarked) ;


       //RECEIVER END

       Mat image_2b_extracted = RGB2YCoCg( rgb_image_watermarked);

       Mat z1 = Mat_<double>(nor,noc);
       Mat z2 = Mat_<double>(nor,noc);
       Mat z3 = Mat_<double>(nor,noc);
       Mat re[3];

        split(image_2b_extracted, re);
        z1 = re[0];
        z2 = re[1];
        z3 = re[2];
       Mat extracted_image_channels[3], extracted_image;

       extracted_image_channels[0] = extractWatermarkInterpolation( z1, o1.len_wm,
o1.overhead, watermarks); //WATERMARKED IN YCoCg
       extracted_image_channels[1] = extractWatermarkInterpolation( z2, o2.len_wm,
o2.overhead, watermarks);
       extracted_image_channels[2] = extractWatermarkInterpolation( z3, o3.len_wm,
o3.overhead, watermarks);
       //cout<<"LENGTH OF WATERMARK: "<<o1.len_wm + o2.len_wm + o3.len_wm<<endl;

       merge( extracted_image_channels, 3, extracted_image);

       Mat final_image= YCoCg2RGB(extracted_image);

       PSNR(image, rgb_image_watermarked);

       cout<<"Time taken: "<< ( ((double)(getTickCount())-t)/ getTickFrequency() );
       final_image.convertTo(final_image, CV_8UC3 );
       imshow( "Extracted image final", final_image);

    cvWaitKey();
 return 0;
}
```