[CTT451] – [Nhập môn Thị giác Máy tính]

Tháng 5/2013

Kalman và Particle Filter

Bộ môn TGMT và KH Rô-bốt
Khoa Công nghệ thông tin
ĐH Khoa học tự nhiên TP HCM

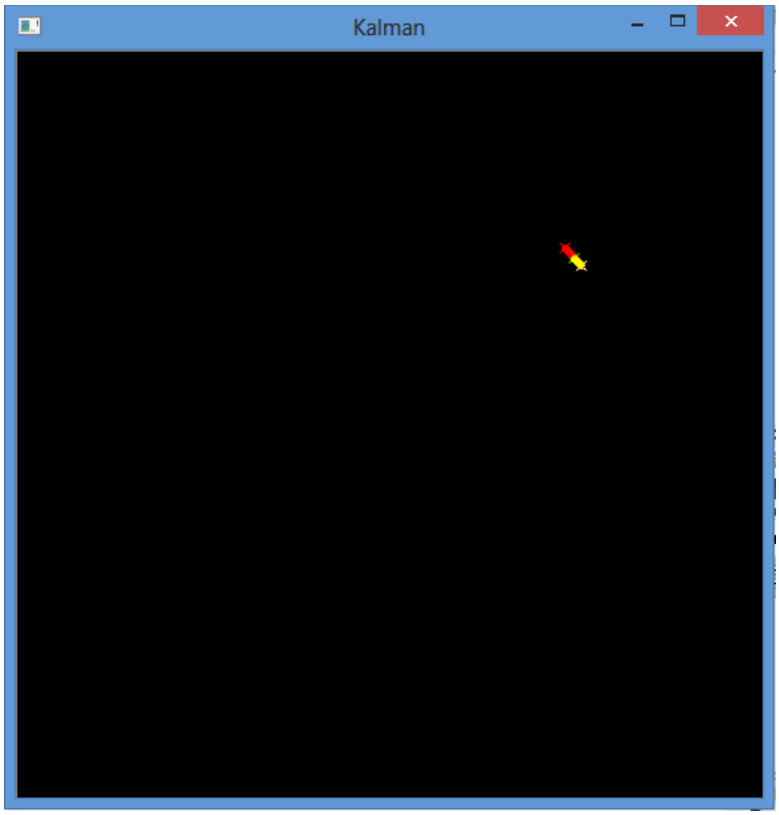
MỤC LỤC

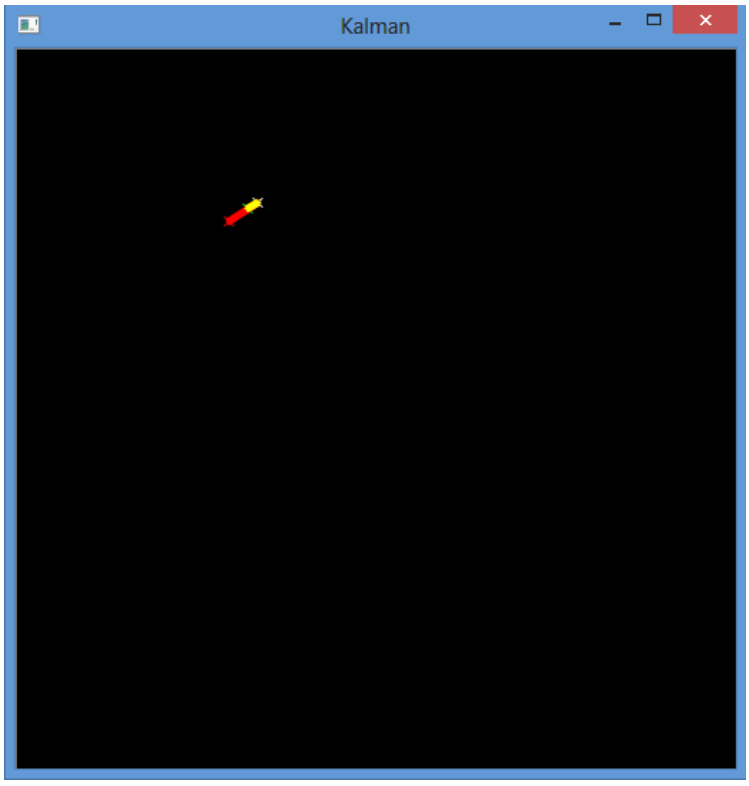
1 Ví dụ sử dụng Kalman Filter:

Giả sử chúng ta có một điểm chuyển động xung quanh một vòng tròn, như một chiếc xe chạy trên đường đua. Chiếc xe chuyển động với vận tốc là một hằng số. Xác định vị trí chiếc xe sử dụng phương pháp theo vết.

```
#include "opencv2/video/tracking.hpp"
#include "opencv2/highgui/highgui.hpp"

#include <stdio.h>

using namespace cv;

static inline Point calcPoint(Point2f center, double R, double angle)
{
    return center + Point2f((float)cos(angle), (float)-sin(angle))*(float)R;
}
int main(int, char**)
{
    Mat img(500, 500, CV_8UC3);
    KalmanFilter KF(2, 1, 0);
    Mat state(2, 1, CV_32F); /* (phi, delta_phi) */
    Mat processNoise(2, 1, CV_32F);
    Mat measurement = Mat::zeros(1, 1, CV_32F);
    char code = (char)-1;

    for(;;)
    {
           randn( state, Scalar::all(0), Scalar::all(0.1) );
           KF.transitionMatrix = *(Mat_<float>(2, 2) << 1, 1, 0, 1);

           setIdentity(KF.measurementMatrix);
           setIdentity(KF.processNoiseCov, Scalar::all(1e-5));
           setIdentity(KF.measurementNoiseCov, Scalar::all(1e-1));
           setIdentity(KF.errorCovPost, Scalar::all(1));

           randn(KF.statePost, Scalar::all(0), Scalar::all(0.1));

           for(;;)
           {
                  Point2f center(img.cols*0.5f, img.rows*0.5f);
                  float R = img.cols/3.f;
                  double stateAngle = state.at<float>(0);
                  Point statePt = calcPoint(center, R, stateAngle);

                  Mat prediction = KF.predict();
                  double predictAngle = prediction.at<float>(0);
                  Point predictPt = calcPoint(center, R, predictAngle);

                  randn( measurement, Scalar::all(0),
Scalar::all(KF.measurementNoiseCov.at<float>(0)));

                  // generate measurement
                  measurement += KF.measurementMatrix*state;

                  double measAngle = measurement.at<float>(0);
```

```
                Point measPt = calcPoint(center, R, measAngle);

                // plot points
    #define drawCross( center, color, d )     \
    line( img, Point( center.x - d, center.y - d ),         \
    Point( center.x + d, center.y + d ), color, 1, CV_AA, 0);     \
    line( img, Point( center.x + d, center.y - d ),         \
    Point( center.x - d, center.y + d ), color, 1, CV_AA, 0 )

                img = Scalar::all(0);
                drawCross( statePt, Scalar(255,255,255), 3 );
                drawCross( measPt, Scalar(0,0,255), 3 );
                drawCross( predictPt, Scalar(0,255,0), 3 );
                line( img, statePt, measPt, Scalar(0,0,255), 3, CV_AA, 0 );
                line( img, statePt, predictPt, Scalar(0,255,255), 3, CV_AA, 0 );

                if(theRNG().uniform(0,4) != 0)
                        KF.correct(measurement);

                randn( processNoise, Scalar(0),
Scalar::all(sqrt(KF.processNoiseCov.at<float>(0, 0))));
                state = KF.transitionMatrix*state + processNoise;

                imshow( "Kalman", img );
                code = (char)waitKey(100);

                if( code > 0 )
                        break;
            }
            if( code == 27 || code == 'q' || code == 'Q' )
                    break;
    }

    return 0;
}
```


Kalman


Kalman

2 Mean-Shift

int cvMeanShift(

const CvArr* prob_image,

CvRect window,

CvTermCriteria criteria,

CvConnectedComp* comp

);

prob_image biểu diễn vị trí mật độ có thể của dữ liệu.

window cửa sổ kernel ban đầu

criteria: quyết định số lần lặp của window có thể dừng

comp chứa tọa độ tìm kiếm của window.

3 Phát hiện và theo vết khuôn mặt trong video

```
#include "opencv/cv.h"
#include "opencv/cxcore.h"
#include "opencv/highgui.h"
#include <iostream>
#include <stdlib.h>
#include <time.h>

using namespace std;

// Create memory for calculations
static CvMemStorage* storage = 0;

// Create a new Haar classifier
static CvHaarClassifierCascade* cascade = 0;

// Create a string that contains the cascade name
const char* cascade_name = "haarcascade_frontalface.xml";
/*    "haarcascade_profileface.xml";*/

void detect_and_draw(IplImage* img);

bool writeActive;
IplImage* im;
IplImage* imFace;
IplImage* imR;
IplImage* imG;
IplImage* imB;
```

```
struct face
{
	CvRect rectangle;
	int dx, dy, confidence;
	float weight;
	CvHistogram* rHistogram;
	CvHistogram* gHistogram;
	CvHistogram* bHistogram;
};

vector<face> allFaces;

CvHistogram* getHistogram(IplImage* im, CvRect r)
{
	cvSetImageROI(im, r);
	int numBins = 64;
	float range[] = {0.0,255.0};
	float* ranges[] = {range};
	CvHistogram* h = cvCreateHist(1, &numBins, CV_HIST_ARRAY, ranges);
	cvCalcHist(&im, h);
	/*for(int binIter=0; binIter<numBins; binIter++)
	{
		cout<<cvQueryHistValue_1D(h, binIter)<<" ";
	}
	cout<<endl;*/
	cvResetImageROI(im);
	return h;
}

vector<face> getSamples(face f, int predX, int predY, int predW, int predH, int
numSamples, float searchSTD, int wLimit, int hLimit)
{
	vector<face> samples;

	float u1,u2,u3,u4,n1,n2,n3,n4,probability,scale,rCorr,gCorr,bCorr,likelihood;
	int newWidth,newHeight;

	//generate random samples
	for(int randGenIter=0; randGenIter<numSamples; randGenIter++)
	{
		//generate two random uniformly distributed numbers
		u1 = ((float)rand())/RAND_MAX;
		u2 = ((float)rand())/RAND_MAX;
		u3 = ((float)rand())/RAND_MAX;
		u4 = ((float)rand())/RAND_MAX;
		//get normally distributed random numbers using box-muller transform (has
mean 0 and std 1)
		n1 = sqrt(-2*log(u1)) * cos(2*3.14159265359*u2);
		n2 = sqrt(-2*log(u1)) * sin(2*3.14159265359*u2);
		n3 = sqrt(-2*log(u3)) * sin(2*3.14159265359*u4);


		//probability = pow(2.71828,-0.5*n1*n1)/sqrt(2*3.14159265359) *
pow(2.71828,-0.5*n2*n2)/sqrt(2*3.14159265359);
		//probability *= pow(2.71828,-0.5*n3*n3)/sqrt(2*3.14159265359);

		//make std dev one third of face dimensions and mean at the predicted
position
```

```
			n1*=f.rectangle.width * searchSTD;
			n1+=predX;
			n2*=f.rectangle.height * searchSTD;
			n2+=predY;

			n3=1;
			/*n3*=0.05;
			n3+=1;
			n3 = MIN(1.3, MAX(0.7,n3) );/**/



			scale = n3;
			newWidth = predW * scale;
			//scale = n4;
			newHeight = predH * scale;

			if (n1>0 && n2>0 && n1<wLimit-newWidth && n2<hLimit-newHeight)//if
randomized position is on the image
			{
				//declare a face at the location
				face newFace;
				newFace.rectangle = cvRect(n1,n2,newWidth,newHeight);
				newFace.rHistogram = getHistogram(imR, newFace.rectangle);
				newFace.gHistogram = getHistogram(imG, newFace.rectangle);
				newFace.bHistogram = getHistogram(imB, newFace.rectangle);
				newFace.dx=0;
				newFace.dy=0;

				//calculate likelihood / weight
				//cout<<" "<<newFace.rectangle.x<<" "<<newFace.rectangle.y<<"
"<<newFace.rectangle.width<<" "<<newFace.rectangle.height<<endl;
				//cout<<" "<<allFaces.at(faceIter).rectangle.x<<"
"<<allFaces.at(faceIter).rectangle.y<<" "<<allFaces.at(faceIter).rectangle.width<<"
"<<allFaces.at(faceIter).rectangle.height<<endl;
				rCorr = cvCompareHist(newFace.rHistogram, f.rHistogram,
CV_COMP_CORREL);
				gCorr = cvCompareHist(newFace.gHistogram, f.gHistogram,
CV_COMP_CORREL);
				bCorr = cvCompareHist(newFace.bHistogram, f.bHistogram,
CV_COMP_CORREL);

				likelihood = (rCorr*0.4 + gCorr*0.3 + bCorr*0.3);

				newFace.weight = pow(2.718281828, -16.0 * (1-likelihood));
				//cout<<newFace.weight<<endl;
				samples.push_back(newFace);
			}
		}

		return samples;
}


vector<face> resample(vector<face> samples, face f, int wLimit, int hLimit)
{
		float totalWeight=0;
		for(int sampleIter=0; sampleIter<samples.size(); sampleIter++)
```

```
	{
		//cout<<samples.at(sampleIter).weight<<endl;
		totalWeight+=samples.at(sampleIter).weight;
	}
	vector<face> resamples;
	vector<face> allResamples;
	int numSamplesToDraw;
	for(int sampleIter=0; sampleIter<samples.size(); sampleIter++)
	{
		resamples.clear();
		numSamplesToDraw = (int)(((samples.at(sampleIter).weight/totalWeight) *
samples.size())+0.5);

		//predicted position
		int predX = samples.at(sampleIter).rectangle.x;
		int predY = samples.at(sampleIter).rectangle.y;

		resamples = getSamples(f, predX, predY,
samples.at(sampleIter).rectangle.width, samples.at(sampleIter).rectangle.height,
numSamplesToDraw, 0.1, wLimit, hLimit);
		//add resamples to the vector of all resamples
		for(int resampleIter=0; resampleIter<resamples.size(); resampleIter++)
		{
			allResamples.push_back(resamples.at(resampleIter));
		}
	}
	return allResamples;
}

void drawFaces()
{
	//copy the image and draw the faces
	cvCopy(im, imFace);

	CvPoint pt1, pt2;
	CvScalar rectColor;
	//draw the faces
	for(int faceIter = 0; faceIter < allFaces.size(); faceIter++ )
	{
		pt1.x = allFaces.at(faceIter).rectangle.x;
		pt2.x = pt1.x + allFaces.at(faceIter).rectangle.width;
		pt1.y = allFaces.at(faceIter).rectangle.y;
		pt2.y = pt1.y + allFaces.at(faceIter).rectangle.height;

		rectColor = cvScalar(0,0,0,0);
		cvRectangle( imFace, pt1, pt2, rectColor, 3, 8, 0 );
		rectColor = cvScalar(0,255,0,0);
		cvRectangle( imFace, pt1, pt2, rectColor, 1, 8, 0 );
	}

	cvShowImage("Faces",imFace);
}

int main( int argc, char** argv )
{
	//initialize random seed
	srand ( time(NULL) );
```

```
        cout<<"wait..";
        cvWaitKey(3000);
        cout<<"go"<<endl;

        // Load the HaarClassifierCascade
    cascade = (CvHaarClassifierCascade*)cvLoad( cascade_name, 0, 0, 0 );
        // Allocate the memory storage
    storage = cvCreateMemStorage(0);

        const char* filename = "data//ForrestGump.avi";
        //CvCapture* capture = cvCreateFileCapture(filename);
        CvCapture* capture = cvCaptureFromCAM(0);

        if(!capture) cout << "No camera detected" << endl;
        cvNamedWindow( "result", 1 );
        IplImage* iplImg = cvQueryFrame( capture );
        cvShowImage("result", iplImg);
        cvWaitKey(0);

        int i = cvGrabFrame(capture);

        im = cvRetrieveFrame(capture);

        CvVideoWriter* writer = cvCreateVideoWriter("out.mp4", CV_FOURCC('F','M','P','4'),
10, cvSize(im->width,im->height), 1);

        imFace = cvCloneImage(im);
        IplImage* imCopy = cvCloneImage(im);

        //allocate some images used to extract a skin likelihood map
        IplImage* imHSV = cvCreateImage(cvSize(im->width,im->height),IPL_DEPTH_8U, 3);
        IplImage* skin = cvCreateImage(cvSize(im->width,im->height),IPL_DEPTH_8U, 1);

        imR = cvCreateImage(cvSize(im->width,im->height),IPL_DEPTH_8U, 1);
        imG = cvCreateImage(cvSize(im->width,im->height),IPL_DEPTH_8U, 1);
        imB = cvCreateImage(cvSize(im->width,im->height),IPL_DEPTH_8U, 1);

        IplImage* sampling = cvCreateImage(cvSize(im->width,im->height),IPL_DEPTH_32F, 1);

        int frame = 0;

        int keyPressed = -1;

        writeActive=true;

        while(i!=0 && (keyPressed==-1 || keyPressed=='f' || keyPressed=='w'))
        {
                cvShowImage("Given",im);

                cvCvtColor(im, imHSV, CV_BGR2HSV);
                CvScalar pixRGB, pixHSV;
                float cb, cr;

                //separate into channels
                cvSetImageCOI(im,1);
                cvCopy(im,imB);
                cvSetImageCOI(im,2);
                cvCopy(im,imG);
```

```
		cvSetImageCOI(im,3);
		cvCopy(im,imR);
		cvSetImageCOI(im,0);

		if(keyPressed=='w')
		{
			writeActive=!writeActive;
		}

		if (frame==0 || allFaces.size()==0 || keyPressed=='f') //detect faces
		{
			// Clear the memory storage which was used before
			 cvClearMemStorage( storage );

			// There can be more than one face in an image. So create a growable
sequence of faces.
			// Detect the objects and store them in the sequence
			CvSeq* faces = cvHaarDetectObjects( im, cascade, storage, 1.1, 2,
CV_HAAR_DO_CANNY_PRUNING, cvSize(40, 40) );

			for(int currentFace=0; currentFace<faces->total; currentFace++)
			{
				//get rectangle bounding first face
				CvRect* r = (CvRect *)cvGetSeqElem( faces, currentFace );

				face newFace;
				newFace.rectangle = *r;
				newFace.confidence = 2;
				newFace.dx = 0;
				newFace.dy = 0;

				//find the total amount of skin-colored pixels in the face
				float skinSum=0;
				for (int y=r->y; y<r->y+r->height; y++)
				{
					for (int x=r->x; x<r->x+r->width; x++)
					{
						pixRGB = cvGet2D(im,y,x);
						pixHSV = cvGet2D(imHSV,y,x);
						cb =  0.148*pixRGB.val[2] - 0.291*pixRGB.val[1]
+ 0.439*pixRGB.val[0] + 128;
						cr =  0.439*pixRGB.val[2] - 0.368*pixRGB.val[1]
- 0.071*pixRGB.val[0] + 128;
						if ( ( pixHSV.val[0]>245 || pixHSV.val[0]<25.5)
&& 140<=cr && cr<=165 && 140<=cb && cb<=195)
						{
							skinSum++;
						}
					}
				}
				//if less than 30% skin, this face doesnt count
				if (skinSum / (r->width*r->height) < 0.3)
				{
					//break;
				}

				//check to see if this face is roughly matching an existing
face
```

```
                        bool matchesExisting=false;

                        for(int faceIter = 0; faceIter < allFaces.size(); faceIter++ )
                        {
                                //find the width and height of the region of overlap
                                int overlapWidth, overlapHeight;
                                if ( newFace.rectangle.x <
allFaces.at(faceIter).rectangle.x)
                                {
                                        overlapWidth = min( newFace.rectangle.x +
newFace.rectangle.width - allFaces.at(faceIter).rectangle.x,
allFaces.at(faceIter).rectangle.width);
                                }else{
                                        overlapWidth = min(
allFaces.at(faceIter).rectangle.x + allFaces.at(faceIter).rectangle.width -
newFace.rectangle.x, newFace.rectangle.width);
                                }
                                if ( newFace.rectangle.y <
allFaces.at(faceIter).rectangle.y)
                                {
                                        overlapHeight = min( newFace.rectangle.y +
newFace.rectangle.height - allFaces.at(faceIter).rectangle.y,
allFaces.at(faceIter).rectangle.height);
                                }else{
                                        overlapHeight = min(
allFaces.at(faceIter).rectangle.y + allFaces.at(faceIter).rectangle.height -
newFace.rectangle.y, newFace.rectangle.height);
                                }

                                //if region of overlap is greater than 60% of larger
rectangle, then faces are the same
                                if ( ((float)overlapWidth*overlapHeight)/(max(
newFace.rectangle.width*newFace.rectangle.height,
allFaces.at(faceIter).rectangle.width*allFaces.at(faceIter).rectangle.height) )>0.6)
                                {
                                        matchesExisting = true;
                                        allFaces.at(faceIter).confidence = min( 4,
allFaces.at(faceIter).confidence+2);
                                        allFaces.at(faceIter).rectangle =
newFace.rectangle;
                                        allFaces.at(faceIter).dx = newFace.dx;
                                        allFaces.at(faceIter).dy = newFace.dy;
                                        allFaces.at(faceIter).rHistogram =
getHistogram(imR, newFace.rectangle);
                                        allFaces.at(faceIter).gHistogram =
getHistogram(imG, newFace.rectangle);
                                        allFaces.at(faceIter).bHistogram =
getHistogram(imB, newFace.rectangle);
                                        break;
                                }
                        }

                        if (!matchesExisting) //if its new, add it to our vector
                        {
                                newFace.rHistogram = getHistogram(imR,
newFace.rectangle);
                                newFace.gHistogram = getHistogram(imG,
newFace.rectangle);
```

```
                                        newFace.bHistogram = getHistogram(imB,
newFace.rectangle);
                                        allFaces.push_back(newFace);
                                }
                        }

                        //subtract 1 from confidence of all faces
                        for(int faceIter = 0; faceIter < allFaces.size(); faceIter++ )
                        {
                                allFaces.at(faceIter).confidence--;
                                if (allFaces.at(faceIter).confidence < 1) //if confidence
gone, remove it
                                {
                                        allFaces.erase(allFaces.begin()+faceIter,
allFaces.begin()+faceIter+1);
                                        faceIter--;
                                }
                        }
                }

                if(allFaces.size()>0) //track faces
                {
                        cvSet(sampling,cvScalar(0));
                        for(int faceIter = 0; faceIter < allFaces.size(); faceIter++ )
//first face only for now
                        {
                                //predicted position
                                int predX = allFaces.at(faceIter).rectangle.x +
allFaces.at(faceIter).dx;
                                int predY = allFaces.at(faceIter).rectangle.y +
allFaces.at(faceIter).dy;

                                vector<face> samples = getSamples(allFaces.at(faceIter),
predX, predY, allFaces.at(faceIter).rectangle.width,
allFaces.at(faceIter).rectangle.height, 100, 0.2, im->width, im->height);

                                //do importance resampling a number of times
                                for(int resampling=0; resampling<3; resampling++)
                                {
                                        samples = resample(samples, allFaces.at(faceIter), im-
>width, im->height);
                                }

                                int bestIdx=0;
                                float bestWeight=0;

                                //generate random samples
                                for(int sampleIter=0; sampleIter<samples.size(); sampleIter++)
                                {
                                        if (samples.at(sampleIter).weight > bestWeight)
                                        {
                                                bestWeight = samples.at(sampleIter).weight;
                                                bestIdx = sampleIter;
                                        }
                                        //cvSet2D(sampling,
samples.at(sampleIter).rectangle.y,samples.at(sampleIter).rectangle.x,
cvScalar(samples.at(sampleIter).weight));
                                }
```

```
                        //move to best sample
                        allFaces.at(faceIter).dx = samples.at(bestIdx).rectangle.x -
allFaces.at(faceIter).rectangle.x;
                        allFaces.at(faceIter).dy = samples.at(bestIdx).rectangle.y -
allFaces.at(faceIter).rectangle.y;
                        allFaces.at(faceIter).rectangle =
samples.at(bestIdx).rectangle;
                        /*cvCopyHist(samples.at(bestIdx).rHistogram,
&allFaces.at(faceIter).rHistogram);
                        cvCopyHist(samples.at(bestIdx).gHistogram,
&allFaces.at(faceIter).gHistogram);
                        cvCopyHist(samples.at(bestIdx).bHistogram,
&allFaces.at(faceIter).bHistogram);*/
                    }
                    drawFaces();
                    //cvCvtColor(imFace, imCopy, CV_BGR2RGB);
                    //if(writeActive)
                    //cvWriteFrame(writer, imFace);
                    //scaleAndShow(sampling,"Sampling");
                    //cvWaitKey(1);
            }

            i = cvGrabFrame(capture);
            im = cvRetrieveFrame(capture);
            frame++;

            keyPressed = cvWaitKey(100);
        }

        cvReleaseImage(&im);
        cvReleaseImage(&imCopy);
        cvReleaseImage(&imHSV);
        cvReleaseImage(&skin);
        cvReleaseCapture(&capture);
        cvReleaseVideoWriter(&writer);
        return 0;
}
```