ПРИЛОЖЕНИЕ Б
(*обязательное*)

Исходный текст типа `FeaturesCalculation`

```
FeaturesCalculation::FeaturesCalculation(Mat  img,  QVector<Seed>
seedVector, Mat first)
{
    srcImg = img.clone();
    firstImg = first.clone();
    seedVect = QVector<Seed>(seedVector);
    calculateTextureGLCM();
}

FeaturesCalculation::~FeaturesCalculation()
{

}

enum  PARAMETR  {  AREA,  LUMA,  MATEXP,  MASSCENTRE,  ELLONGATION,
CONTRAST,    HOMOGENEITY,    DISSIMILARITY,    ENERGY,    ENTROPY,
CORRELATION };

void     FeaturesCalculation::calculateSomeGeometryParam(PARAMETR
param)
{
    int i = 0, oldArea, oldLuma,  oldPixel;
    int len = seedVect.length();
    Mat channel[3], YCRImage;
    cvtColor(firstImg, YCRImage, COLOR_BGR2YCrCb);
    split(YCRImage, channel);

    double *m11 = new double[len];
    double *m02 = new double[len];
    double *m20 = new double[len];

    for(int i =0; i < len; i++)
    {
        m11[i] = 0;
        m02[i] = 0;
        m20[i] = 0;
    }

    for( int y = 0; y < srcImg.rows; y++ )
       {
            for( int x = 0; x < srcImg.cols; x++ )
            {
                if(!(srcImg.at<Vec3b>(y,x)[0] == 0 &&
                        srcImg.at<Vec3b>(y,x)[1] == 0 &&
                        srcImg.at<Vec3b>(y,x)[2] == 0))
                {
```

```
                    i = 0;
                    for(Seed s: seedVect)
                    {

                        if(s.GetColor().val[0]                  ==
srcImg.at<Vec3b>(y,x)[0] &&
                          s.GetColor().val[1]                   ==
srcImg.at<Vec3b>(y,x)[1] &&
                          s.GetColor().val[2]                   ==
srcImg.at<Vec3b>(y,x)[2])
                        {
                            if(param == AREA)
                            {
                                oldArea = s.GetArea();
                                seedVect[i].SetArea(oldArea+1);
                                break;
                            }
                            if(param == LUMA)
                            {
                                oldLuma = s.GetLuma();
                                int  l  =  channel[0].at<uchar>(y,
x);
                                seedVect[i].SetLuma(oldLuma     +
l);
                                oldPixel = s.GetCountOfPixels();

seedVect[i].SetCountOfPixels(oldPixel+1);
                                break;
                            }
                            if(param == MATEXP)
                            {

seedVect[i].countOfPixelsOnLevel[channel[0].at<uchar>(y,x)]++;
                            }
                            if(param == MASSCENTRE)
                            {
                                seedVect[i].centerMass.x += x ;
                                seedVect[i].centerMass.y += y ;
                            }
                            if(param == ELONGATION)
                            {
                                m11[i]        +=        (x        -
seedVect[i].centerMass.x)*(y - seedVect[i].centerMass.y);
                                m02[i]      +=      qPow((y      -
seedVect[i].centerMass.y), 2);
                                m20[i]      +=      qPow((x      -
seedVect[i].centerMass.x), 2);
                            }
                        }
                        i++;
                    }
                }
```

```
            }
        }
    if(param == LUMA)
    {
        float luma = 0;
        for(int k = 0; k < seedVect.length(); k++)
        {
            luma        =        seedVect[k].GetLuma()        /
seedVect[k].GetCountOfPixels();
            seedVect[k].SetLuma(luma);
            luma = 0;
        }
    }
    if(param == MATEXP)
    {
        for(int index = 0; index < seedVect.length(); index++)
        {
            for(int k = 0; k < 256; k++)
            {
                seedVect[index].matExpect      +=      k      *
seedVect[index].countOfPixelsOnLevel[k]                        /
seedVect[index].GetArea();
            }
        }
    }
    if(param == MASSCENTRE)
    {
        for(int index = 0; index < seedVect.length(); index++)
        {
            seedVect[index].centerMass.x                      /=
seedVect[index].GetArea();
            seedVect[index].centerMass.y                      /=
seedVect[index].GetArea();
        }
    }
    if(param == ELONGATION)
    {
        double m1=0, m2=0;
        for(int index = 0; index < seedVect.length(); index++)
        {
            m1 = (m20[index] + m02[index] + qSqrt((m20[index] -
m02[index])*(m20[index]         -         m02[index])         +
4*m11[index]*m11[index]));
            m2 = (m20[index] + m02[index] - qSqrt((m20[index] -
m02[index])*(m20[index]         -         m02[index])         +
4*m11[index]*m11[index]));

        }
    }
}

void FeaturesCalculation::calculateArea()
```

```
{
    calculateSomeGeometryParam(AREA);
}



void FeaturesCalculation::calculatePerimetr()
{
    int oldPerimetr, i;

    for( int y = 0; y < srcImg.rows; y++ )
       {
            for( int x = 0; x < srcImg.cols; x++ )
            {
                if(!(srcImg.at<Vec3b>(y,x)[0] == 0 &&
                        srcImg.at<Vec3b>(y,x)[1] == 0 &&
                        srcImg.at<Vec3b>(y,x)[2] == 0))
                {
                    if(HaveBlackNeighbors( x, y))
                    {
                        i = 0;
                        for(Seed s: seedVect)
                        {

                            if(s.GetColor().val[0]              ==
srcImg.at<Vec3b>(y,x)[0] &&
                               s.GetColor().val[1]              ==
srcImg.at<Vec3b>(y,x)[1] &&
                               s.GetColor().val[2]              ==
srcImg.at<Vec3b>(y,x)[2])
                            {
                                oldPerimetr = s.GetPerimetr();

seedVect[i].SetPerimetr(oldPerimetr+1);
                                break;
                            }
                            i++;
                        }
                    }
                }
            }
       }
}

void FeaturesCalculation::calculateCompactness()
{
    double compact = 0;
    for(int i = 0; i < seedVect.length(); i++)
    {
        compact                                                 =
seedVect[i].GetPerimetr()*seedVect[i].GetPerimetr()             /
seedVect[i].GetArea();
```

```
        seedVect[i].SetCompactness(compact);
    }
}

bool FeaturesCalculation::HaveBlackNeighbors( int x, int y)
{
    if((srcImg.at<Vec3b>(y-1,x)[0] == 0 &&
            srcImg.at<Vec3b>(y-1,x)[1] == 0 &&
            srcImg.at<Vec3b>(y-1,x)[2] == 0) ||
            (srcImg.at<Vec3b>(y+1,x)[0] == 0 &&
             srcImg.at<Vec3b>(y+1,x)[1] == 0 &&
             srcImg.at<Vec3b>(y+1,x)[2] == 0) ||
            (srcImg.at<Vec3b>(y,x-1)[0] == 0 &&
            srcImg.at<Vec3b>(y,x-1)[1] == 0 &&
            srcImg.at<Vec3b>(y,x-1)[2] == 0) ||
            (srcImg.at<Vec3b>(y,x+1)[0] == 0 &&
             srcImg.at<Vec3b>(y,x+1)[1] == 0 &&
             srcImg.at<Vec3b>(y,x+1)[2] == 0))
    {
        return true;
    }
    else
    {
        return false;
    }

}

QVector<Seed> FeaturesCalculation::GetSeedVector()
{
    return seedVect;
}

void FeaturesCalculation::calculateLumaParameter(Mat srcImage)
{

    calculateSomeGeometryParam(LUMA);
}

void FeaturesCalculation::calculateMatExpectation()
{
    calculateSomeGeometryParam(MATEXP);
}

void FeaturesCalculation::calculateDispertion()
{
    for(int index = 0; index < seedVect.length(); index++)
    {
        for(int k = 0; k < 256; k++)
        {
            seedVect[index].dispersion     +=     qPow((k     -
seedVect[index].matExpect),2)                                   *
```

```
seedVect[index].countOfPixelsOnLevel[k]                                 /
seedVect[index].GetArea();
        }
    }
}

void FeaturesCalculation::calculateMassCenter()
{
    calculateSomeGeometryParam(MASSCENTRE);
}


void FeaturesCalculation::calculateElongation()
{

    calculateSomeGeometryParam(ELONGATION);
}

void FeaturesCalculation::calculateTextureGLCM()
{
    int leftP = 0, rightP = 0;
    int count;
    Mat gray = Mat::zeros(srcImg.size(), CV_8UC1);
    cvtColor(firstImg, gray, CV_BGR2GRAY);

    for( int y = 0; y < srcImg.rows; y++ )
       {
            for( int x = 0; x < srcImg.cols; x++ )
            {
                //printf("p(%d,%d)[%d][%d][%d]\n",y,x,
srcImg.at<Vec3b>(y,x)[0],srcImg.at<Vec3b>(y,x)[1],srcImg.at<Vec3
b>(y,x)[2]);
                if(!(srcImg.at<Vec3b>(y,x)[0] == 0 &&
                        srcImg.at<Vec3b>(y,x)[1] == 0 &&
                        srcImg.at<Vec3b>(y,x)[2] == 0))
                {
                    int i = 0;
                    for(Seed s: seedVect)
                    {

                        if((s.GetColor().val[0]                 ==
srcImg.at<Vec3b>(y,x)[0] &&
                            s.GetColor().val[1]                 ==
srcImg.at<Vec3b>(y,x)[1] &&
                            s.GetColor().val[2]                 ==
srcImg.at<Vec3b>(y,x)[2]) &&
                                !(srcImg.at<Vec3b>(y,x+1)[0]   ==
0 &&
                                 srcImg.at<Vec3b>(y,x+1)[1]  ==  0
&&
                                 srcImg.at<Vec3b>(y,x+1)[2]  ==  0
))
```

```
                        {

                            leftP = gray.at<uchar>(y,x)/8;
                            rightP = gray.at<uchar>(y,x+1)/8;


                            count                                   =
seedVect[i].GLCM.at<uchar>(leftP, rightP);
                            count++;
                            seedVect[i].GLCM.at<uchar>(leftP,
rightP) = count;

                        }
                        i++;
                    }

                }
            }
    }

    for(int k = 0; k < seedVect.length(); k++)
    {
        for(int i=0; i< seedVect[k].GLCM.rows; i++)
            for(int j=0; j< seedVect[k].GLCM.cols; j++)
            {
                if(seedVect[k].GLCM.at<uchar>(i,j) != 0)
                    seedVect[k].countOfPairs++;
            }

    }

    for(int k = 0; k < seedVect.length(); k++)
    {
        createGLCM(k);
    }
}

void FeaturesCalculation::createGLCM(int indexOfSeed)
{
    if(!srcImg.empty())
    {
        Mat GLCMtrasposed = Mat::zeros(32, 32, CV_8UC1);
        Mat GLCMsymmetric = Mat::zeros(32, 32, CV_8UC1);

        imshow("glcm1", seedVect[indexOfSeed].GLCM);

        for(int i=0; i< seedVect[indexOfSeed].GLCM.rows; i++)
            for(int   j=0;   j<   seedVect[indexOfSeed].GLCM.cols;
j++)
            {
                GLCMtrasposed.at<uchar>(j,i)                    =
seedVect[indexOfSeed].GLCM.at<uchar>(i,j);
```

```
            }

        for(int i=0; i<seedVect[indexOfSeed].GLCM.rows; i++)
            for(int j=0; j<seedVect[indexOfSeed].GLCM.cols; j++)
            {
                GLCMsymmetric.at<uchar>(i,j)                    =
(seedVect[indexOfSeed].GLCM.at<uchar>(i,j)                      +
GLCMtrasposed.at<uchar>(i,j));//  countOfPairs;
            }

        seedVect[indexOfSeed].GLCM = GLCMsymmetric;

    }
}

void FeaturesCalculation::calculateContrast()
{
    calculateTextureParameter(CONTRAST);
}

void FeaturesCalculation::calculateHomogeneity()
{
   calculateTextureParameter(HOMOGENEITY);
}

void FeaturesCalculation::calculateDissimilarity()
{
    calculateTextureParameter(DISSIMILARITY);
}

void FeaturesCalculation::calculateEntropy()
{
    calculateTextureParameter(ENTROPY);
}

void FeaturesCalculation::calculateEnergy()
{
    calculateTextureParameter(ENERGY);
}

void FeaturesCalculation::calculateCorrelation()
{
    calculateTextureParameter(CORRELATION);
}

void      FeaturesCalculation::calculateTextureParameter(PARAMETR
param)
{
    float qrt = 0, fabs = 0, thigmaSqr = 0, U = 0;
    for(int k = 0; k < seedVect.length(); k++)
    {
        for(int i=0; i< seedVect[k].GLCM.rows; i++)
```

```
            for(int j=0; j< seedVect[k].GLCM.cols; j++)
            {

                if(param == DISSIMILARITY)
                {
                    qrt = qPow(i-j,2);
                    fabs = qSqrt(qrt);
                    seedVect[k].dissimilarity                    +=
fabs*seedVect[k].GLCM.at<uchar>(i,j);
                }

                if(param == ENERGY)
                {
                    seedVect[k].energy                           +=
qPow(seedVect[k].GLCM.at<uchar>(i,j),2);
                }

                if(param == ENTROPY)
                {
                    if (seedVect[k].GLCM.at<uchar>(i,j) != 0)
                    {
                        float           ln           =           -
qLn(seedVect[k].GLCM.at<uchar>(i,j));
                        seedVect[k].entropy          +=          -
qLn(seedVect[k].GLCM.at<uchar>(i,j))                             *
seedVect[k].GLCM.at<uchar>(i,j);
                    }
                   printf("ln = %f, char =%d entropy = %f\n",
ln, seedVect[k].GLCM.at<uchar>(i,j), seedVect[k].entropy);
                }

                if(param == HOMOGENEITY)
                {
                    qrt = qPow(i-j,2);
                    if(qrt != 1)
                        seedVect[k].homogeneity  +=  (1/(1-qPow(i-
j,2)))*seedVect[k].GLCM.at<uchar>(i,j);
                }

                if(param == CONTRAST)
                {
                    int pixel = seedVect[k].GLCM.at<uchar>(i,j);
                    float dividing = pixel;
                    float qrt = qPow(i-j, 2);
                    float contr = qrt * dividing;

                    seedVect[k].contrast += contr;
                }

                if(param == CORRELATION)
                {
                    calculateUandThigma(k, U, thigmaSqr);
```

```
                    seedVect[k].correlation                    +=
seedVect[k].GLCM.at<uchar>(i,j)*(i-U)*(j-U) / thigmaSqr;
                }

            }


        if(param == DISSIMILARITY)
            seedVect[k].dissimilarity                          =
seedVect[k].dissimilarity / (seedVect[k].countOfPairs);

        if(param == HOMOGENEITY)
            seedVect[k].homogeneity  =  seedVect[k].homogeneity  /
(seedVect[k].countOfPairs);

        if(param == CONTRAST)
            seedVect[k].contrast                               =
seedVect[k].contrast/(seedVect[k].countOfPairs);

        if(param == ENERGY)
            seedVect[k].energy     =     seedVect[k].energy     /
(seedVect[k].countOfPairs);

        if(param == ENTROPY)
            seedVect[k].entropy    =    seedVect[k].entropy    /
(seedVect[k].countOfPairs);

        if(param == CORRELATION)
            seedVect[k].correlation  =  seedVect[k].correlation  /
(seedVect[k].countOfPairs);
    }
}

void  FeaturesCalculation::calculateUandThigma(int  index,  float
&U, float &thigmaSqr)
{
    for(int i=0; i< seedVect[index].GLCM.rows; i++)
        for(int j=0; j< seedVect[index].GLCM.cols; j++)
        {
            U += seedVect[index].GLCM.at<uchar>(i,j)*i;
        }
    U = U / seedVect[index].countOfPairs;

    for(int i=0; i< seedVect[index].GLCM.rows; i++)
        for(int j=0; j< seedVect[index].GLCM.cols; j++)
        {
            thigmaSqr  +=  seedVect[index].GLCM.at<uchar>(i,j)*(i-
U)*(i-U);
        }
    thigmaSqr = thigmaSqr / seedVect[index].countOfPairs;
}
```