```
 * Embedd_aux_LBP_func.hpp

#ifndef EMBEDD_AUX_LBP_FUNC_HPP_
#define EMBEDD_AUX_LBP_FUNC_HPP_


#include <stdio.h>
#include <math.h>
#include <limits.h>
#include <malloc.h>
#include <assert.h>
#include <stdint.h>
#include <string.h>

#define IMSE_SCALE_FACTOR_BIT       20 // gives the same results
#define IMSE_EPS_SCALE_BIT          10 // gives the same results
//#define IMSE_FEATURE_SCALE_BIT        10 // scalling the feature
#define imse_INTER_LINEAR           1

#define imse_INTER_LINEAR             1
#define imse_INTER_CUBIC              2
#define imse_INTER_AREA               3

#define imse_TYPE_MASK          0x00000FFF
#define imse_MAGIC_VAL          0x42FF0000

#define imse_MAGIC_MASK       0xFFFF0000
#define imse_MAT_MAGIC_VAL    0x42420000
#define imse_AUTOSTEP         0x7fffffff

#define imse_MATND_MAGIC_VAL    0x42430000
#define imse_MAX_DIM            32

#define imse_CN_MAX                64
#define imse_MAT_CN_MASK          ((imse_CN_MAX - 1) << imse_CN_SHIFT)
#define imse_MAT_CN(flags)        ((((flags) & imse_MAT_CN_MASK) >> imse_CN_SHIFT) + 1)
#define imse_MAT_TYPE_MASK        (imse_DEPTH_MAX*imse_CN_MAX - 1)
#define imse_MAT_TYPE(flags)      ((flags) & imse_MAT_TYPE_MASK)
#define imse_MAT_CONT_FLAG_SHIFT  14
#define imse_MAT_CONT_FLAG        (1 << imse_MAT_CONT_FLAG_SHIFT)
#define imse_IS_MAT_CONT(flags)   ((flags) & imse_MAT_CONT_FLAG)

#define imse_8U   0
#define imse_8S   1
#define imse_16U  2
#define imse_16S  3
#define imse_32S  4
#define imse_32F  5
#define imse_64F  6
#define imse_64S  7

#define IMSE_CV_32S         4

#define imse_CN_SHIFT   3
#define imse_DEPTH_MAX  (1 << imse_CN_SHIFT)

#define imse_MAT_DEPTH_MASK       (imse_DEPTH_MAX - 1)
#define imse_MAT_DEPTH(flags)     ((flags) & imse_MAT_DEPTH_MASK)

#define imse_MAKETYPE(depth,cn) (imse_MAT_DEPTH(depth) + (((cn)-1) << imse_CN_SHIFT))
#define imse_MAKE_TYPE imse_MAKETYPE
```

```
#define imse_8UC1 imse_MAKETYPE(imse_8U,1)
#define imse_8UC2 imse_MAKETYPE(imse_8U,2)
#define imse_8UC3 imse_MAKETYPE(imse_8U,3)
#define imse_8UC4 imse_MAKETYPE(imse_8U,4)
#define imse_8UC(n) imse_MAKETYPE(imse_8U,(n))

#define imse_16UC1 imse_MAKETYPE(imse_16U,1)
#define imse_16UC2 imse_MAKETYPE(imse_16U,2)
#define imse_16UC3 imse_MAKETYPE(imse_16U,3)
#define imse_16UC4 imse_MAKETYPE(imse_16U,4)
#define imse_16UC(n) imse_MAKETYPE(imse_16U,(n))

#define  imse_CAST_8U(t)  (uint8_t)(!((t) & ~255) ? (t) : (t) > 0 ? 255 : 0)

/* the alignment of all the allocated buffers */
#define  imse_MALLOC_ALIGN    16

/* BGR/RGB -> Gray */
#define csc_shift  14
#define cscGr  4899 //cscGr_32f  0.299f*(1<<14) fix(cscGr_32f,csc_shift)
#define cscGg  9617 //cscGg_32f  0.587f*(1<<14) fix(cscGg_32f,csc_shift)
#define cscGb  /*fix(cscGb_32f,csc_shift) cscGb_32f  0.114f*/ ((1 << csc_shift) - cscGr -
cscGg)

    typedef struct imse_Point
    {
        int x;
        int y;
    }imse_Point;

    /************************************ imse_Rect
*************************************/

    typedef struct imse_Rect
    {
        int x;
        int y;
        int width;
        int height;
    }imse_Rect;

      //-----------------------------------------------  LBPEvaluator
-------------------------------------

        #define CALC_SUM_(p0, p1, p2, p3, offset) \
            ((p0)[offset] - (p1)[offset] - (p2)[offset] + (p3)[offset])

        #define CALC_SUM(rect,offset) CALC_SUM_((rect)[0], (rect)[1], (rect)[2], (rect)[3],
offset)

        #define CV_SUM_PTRS( p0, p1, p2, p3, sum, rect, step )                    \
            /* (x, y) */                                                          \
            (p0) = sum + (rect).x + (step) * (rect).y,                            \
            /* (x + w, y) */                                                      \
            (p1) = sum + (rect).x + (rect).width + (step) * (rect).y,             \
            /* (x + w, y) */                                                      \
            (p2) = sum + (rect).x + (step) * ((rect).y + (rect).height),          \
            /* (x + w, y + h) */                                                  \
            (p3) = sum + (rect).x + (rect).width + (step) * ((rect).y + (rect).height)
```

```
//------------- functions as macros

/* min & max without jumps */
#define  imse_IMIN(a, b)  ((a) ^ (((a)^(b)) & (((a) < (b)) - 1)))

#define  imse_IMAX(a, b)  ((a) ^ (((a)^(b)) & (((a) > (b)) - 1)))

#define imse_IS_MATND_HDR(mat) \
    ((mat) != NULL && (((const imse_MatND*)(mat))->type & imse_MAGIC_MASK) ==
imse_MATND_MAGIC_VAL)

#define imse_IS_MATND(mat) \
    (imse_IS_MATND_HDR(mat) && ((const imse_MatND*)(mat))->data_ptr != NULL)

#define imse_IS_MAT_HDR(mat) \
    ((mat) != NULL && \
    (((const imse_Mat*)(mat))->type & imse_MAGIC_MASK) == imse_MAT_MAGIC_VAL && \
    ((const imse_Mat*)(mat))->cols > 0 && ((const imse_Mat*)(mat))->rows > 0)

#define imse_IS_MAT(mat) \
    (imse_IS_MAT_HDR(mat) && ((const imse_Mat*)(mat))->data_ptr != NULL)
/* 0x3a50 = 11 10 10 01 01 00 00 ~ array of log2(sizeof(arr_type_elem)) */
/*0xfa50 = 11 11 10 10 01 01 00 00  array of log2(sizeof(arr_type_elem)) *// Added on
25.10.2010
#define imse_ELEM_SIZE(type) \
    (imse_MAT_CN(type) << ((((sizeof(size_t)/4+1)*16384|0xfa50) >> imse_MAT_DEPTH(type)*2) &
3))

#define imse_ARE_TYPES_EQ(mat1, mat2) \
    ((((mat1)->type ^ (mat2)->type) & imse_MAT_TYPE_MASK) == 0)

    /*! \fn size_t imse_alignSize(size_t sz, int n)
    \brief No description yet.
    \param size_t sz No description yet.
    \param int n No description yet.
    */
    static size_t imse_alignSize(size_t sz, int n)
    {
        return (sz + n-1) & -n;
    }

    /*! \fn int imse_clip(int x, int a, int b)
    \brief No description yet.
    \param int x No description yet.
    \param int a No description yet.
    \param int bNo description yet.
    */
    static int imse_clip(int x, int a, int b)
    {
        return x >= a ? (x < b ? x : b-1) : a;
    }

    static void* imse_AlignPtr( const void* ptr, int align)
    {
        assert( (align & (align-1)) == 0 );
        return (void*)( ((size_t)ptr + align - 1) & ~(size_t)(align-1) );
    }

    static void* imse_Malloc( size_t size )
    {
```

```
        uint8_t* udata = (uint8_t*)malloc(size + sizeof(void*) + imse_MALLOC_ALIGN);
        if(!udata)
            return NULL;//return OutOfMemoryError(size);
        uint8_t** adata = (uint8_t**)imse_AlignPtr((uint8_t**)udata + 1, imse_MALLOC_ALIGN);
        adata[-1] = udata;
        return adata;

    }


    static void imse_Free_(void* ptr)
    {
        if(ptr)
        {
            uint8_t* udata = ((uint8_t**)ptr)[-1];
            free(udata);
        }

        ptr=NULL;
    }
#define imse_Free(ptr) (imse_Free_(*(ptr)), *(ptr)=0)

    /*! \fn short imse_saturate_uchar(const int val ,const int SHIFT)
    \brief Saturates the int number to the short type
    \param const int val Integer value that needs saturation
    \param const int SHIFT The number of shift bits used for saturation
    \warning None
    */
    static unsigned char imse_saturate_uchar(const int val ,const int SHIFT)
    {   //SHIFT=INTER_RESIZE_COEF_BITS*2;
        //INTER_RESIZE_COEF_BITS=11;
        //UCHAR_MAX= 255
        int v;
        int DELTA;

        DELTA = 1 << (SHIFT-1);
        v=(val+DELTA)>>SHIFT;
        return (unsigned char)((unsigned)v <= UCHAR_MAX ? v : v > 0 ? UCHAR_MAX : 0);
    }


    /*! \fn void short imse_saturate_short_from_int(int v)
    \brief Saturates the int number to the short type
    \param int v Integer value that needs saturation
    \warning None
    */
    static short imse_saturate_short_from_int(int v)
    {
        short val;
        if (v < SHRT_MIN)
            return SHRT_MIN;

        if (v > SHRT_MAX)
            return SHRT_MAX;

        val= (short) v;

        return val;

    }
```

```
#ifndef __cplusplus
    typedef int bool;
    #define false 0
    #define true 1
#endif



 int imse_Round(double val)
 {
     return (int)floor(val);
     //return (int)lrint(val); // round to nearest integer
 }


    typedef struct imse_intMat
    {
          // includes several bit-fields:
            //  * the magic signature
            //  * continuity flag
            //  * depth
            //  * number of channels
            int flags;
            // the number of rows and columns
            int rows, cols;
            // a distance between successive rows in bytes; includes the gap if any
            size_t step;
            // pointer to the data
            uint8_t* data;

            // helper fields used in locateROI and adjustROI
            uint8_t* datastart;
            uint8_t* dataend;

    }
    imse_intMat;

    typedef struct imse_MatND
    {
        int type;
        int dims;

        uint8_t* data_ptr;

        struct
        {
            int size;
            int step;
        }
        dim[imse_MAX_DIM];
    }
    imse_MatND;

    typedef struct imse_Mat
    {
        int type;
        int step;

        uint8_t* data_ptr;
```

```
    int rows;
    int cols;

}imse_Mat;

int imse_LinkCvMat_to_imse_Mat( CvMat frameCopy, imse_Mat* img)
{
    if (frameCopy.data.ptr == NULL )
    {
        printf("imse_LinkCvMat_to_imse_MAT():: Error-> invalid Cvmat data_ptr");
            return 1;
    }
    if (frameCopy.rows <=0 || frameCopy.cols <=0)
    {
        printf("imse_LinkCvMat_to_imse_MAT():: Error-> Invalid dimension(rows, cols)");
            return 1;
    }

    // start linking
    img->data_ptr = frameCopy.data.ptr;

    img->type = frameCopy.type;
    img->step = frameCopy.step;
    img->rows = frameCopy.rows;
    img->cols = frameCopy.cols;

return 0;
}


static imse_intMat voidTo_intMat(const imse_Mat* m, bool copyData )
{
    imse_intMat aux;
    aux.flags = imse_MAGIC_VAL + (m->type & (imse_MAT_TYPE_MASK|imse_MAT_CONT_FLAG));
    aux.rows = m->rows;
    aux.cols = m->cols;
    aux.step = m->step;
    aux.data = m->data_ptr;

    aux.datastart = m->data_ptr;
    aux.dataend = m->data_ptr;

    if (aux.step == 0 )
        aux.step = m->cols*imse_ELEM_SIZE(aux.flags);
    size_t minstep = m->cols*imse_ELEM_SIZE(aux.flags);
    aux.dataend += aux.step*(aux.rows-1)+ minstep;

    if ( copyData )
    {
        // Not implemented
        //aux.data = 0;
        //aux.datastart = 0;
        //aux.dataend = 0;
        //imse_copyTo_intMat(m,aux);

    }

return aux;
}

static imse_intMat imse_voidToIntMat(const void* arr, bool copyData, bool allowND, int
```

```
coiMode)
    {
        imse_intMat m;
        if( imse_IS_MAT(arr) )
            m = voidTo_intMat((const imse_Mat*)arr, copyData );
        else
        {
            m.flags = 0;
            m.rows = 0;
            m.cols = 0;
            m.step = 0;
            m.data = NULL;
            m.datastart = NULL;
            m.dataend = NULL;
        }
        return m;
    }


    ////////////// -------- New function for embedded support ---------/////////////

    /* Decrements imse_Mat data reference counter and deallocates the data if
       it reaches 0 */
    static  void  imse_DecRefData( void* arr )
    {
        if( imse_IS_MAT( arr ))
        {
            imse_Mat* mat = (imse_Mat*)arr;
            if(mat->data_ptr != NULL)
                    imse_Free( &mat->data_ptr ); //04.11.2010

        }
        else if( imse_IS_MATND( arr ))
        {
            imse_MatND* mat = (imse_MatND*)arr;
            if(mat->data_ptr != NULL)
                    imse_Free( &mat->data_ptr ); //04.11.2010

        }
    }


    // Deallocates the imse_Mat structure and underlying data
    static void
    imse_ReleaseMat( imse_Mat** array )
    {
        if( !array )
            return;//CV_Error( CV_HeaderIsNull, "" );

        if( *array )
        {
            imse_Mat* arr = *array;

            if( (!imse_IS_MAT_HDR(arr)) && !imse_IS_MATND_HDR(arr) )
                return; //CV_Error( CV_StsBadFlag, "" );

            *array = 0;

            imse_DecRefData( arr );
            imse_Free( &arr );
        }
```

```
}


static void imse_intMat_release(imse_intMat dst)
{

    if (dst.datastart!=NULL)
        imse_Free(& dst.datastart);
    dst.data=dst.dataend=dst.datastart=0;
    dst.step=dst.rows=dst.rows=0;

}


static void imse_create_intMat(imse_intMat dst, int _rows, int _cols, int _type)
{
    if( dst.rows == _rows && dst.cols == _cols && imse_MAT_TYPE(dst.flags) == _type
            && dst.data )
            return;
    if(dst.data)
        imse_intMat_release(dst);
    if( _rows > 0 && _cols > 0 )
    {
        dst.flags = imse_MAGIC_VAL + imse_MAT_CONT_FLAG + _type;
        dst.rows = _rows;
        dst.cols = _cols;
        dst.step = imse_ELEM_SIZE(dst.flags)*dst.cols;
        int64_t _nettosize = (int64_t)dst.step*dst.rows;
        size_t nettosize = (size_t)_nettosize;
        if( _nettosize != (int64_t)nettosize )
                  return ;// CV_Error(CV_StsNoMem, "Too big buffer is allocated");

        dst.datastart = dst.data = (uint8_t*)imse_Malloc(nettosize);
        dst.dataend = dst.data + nettosize;
    }

}


static void imse_CheckHuge( imse_Mat* arr )
{
    if( (int64_t)arr->step*arr->rows > INT_MAX )
        arr->type &= ~imse_MAT_CONT_FLAG;
}


// Creates CvMat header only
static imse_Mat* imse_CreateMatHeader( int rows, int cols, int type )
{
    type = imse_MAT_TYPE(type);

    if( rows <= 0 || cols <= 0 )
        ;//CV_Error( CV_StsBadSize, "Non-positive width or height" );

    int min_step = imse_ELEM_SIZE(type)*cols;
    if( min_step <= 0 )
        ; //CV_Error( CV_StsUnsupportedFormat, "Invalid matrix type" );

    imse_Mat* arr = (imse_Mat*)imse_Malloc( sizeof(*arr));

    arr->step = min_step;
```

```
    arr->type = imse_MAT_MAGIC_VAL | type | imse_MAT_CONT_FLAG;
    arr->rows = rows;
    arr->cols = cols;
    arr->data_ptr = 0;


    imse_CheckHuge( arr );
    return arr;
}

// Allocates underlying array data
static void
imse_CreateData( void* arr )
{
    if( imse_IS_MAT_HDR( arr ))
    {
        size_t step, total_size;
        imse_Mat* mat = (imse_Mat*)arr;
        step = mat->step;

        if( mat->data_ptr != 0 )
            return;//CV_Error( CV_StsError, "Data is already allocated" );

        if( step == 0 )
            step = imse_ELEM_SIZE(mat->type)*mat->cols;


        int64_t _total_size = (int64_t)step*mat->rows ; //4.11.2010
        total_size = (size_t)_total_size;
        if(_total_size != (int64_t)total_size)
            return; //CV_Error(CV_StsNoMem, "Too big buffer is allocated" );
        mat->data_ptr = (uint8_t*)imse_Malloc( total_size ); //4.11.2010

    }
    else if( imse_IS_MATND_HDR( arr ))
    {
        imse_MatND* mat = (imse_MatND*)arr;
        int i;
        size_t total_size = imse_ELEM_SIZE(mat->type);

        if( mat->data_ptr != 0 )
           return;//CV_Error( CV_StsError, "Data is already allocated" );

        if( imse_IS_MAT_CONT( mat->type ))
        {
            total_size = (size_t)mat->dim[0].size*(mat->dim[0].step != 0 ?
                         mat->dim[0].step : total_size);
        }
        else
        {
            for( i = mat->dims - 1; i >= 0; i-- )
            {
                size_t size = (size_t)mat->dim[i].step*mat->dim[i].size;

                if( total_size < size )
                    total_size = size;
            }
        }
        mat->data_ptr = (uint8_t*)imse_Malloc( total_size ); //4.11.2010
    }
    else
```

```
            ;//CV_Error( CV_StsBadArg, "unrecognized or unsupported array type" );
    }

    // Creates CvMat and underlying data
    static imse_Mat*
    imse_CreateMat( int height, int width, int type )
    {
        imse_Mat* arr = imse_CreateMatHeader( height, width, type );
        imse_CreateData( arr );

        return arr;
    }

////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
/////  Begin Resize section

    /*! \fn void imse_VResizeLinear(const int** src, uint8_t* dst, const short* beta,
            int width, const int SHIFT  )
    \brief Perform vertical Linear resize
    \param const int** src Source used for resizing.
    \param uint8_t* dst Destination of the resized data.
    \param const short* beta No description yet.
    \param int width Width of the data.
    \param const int SHIFT shift bytes to be used.
    */
    static void imse_VResizeLinear (const int** src, uint8_t* dst, const short* beta,
            int width, const int SHIFT)
    {   //SHIFT=INTER_RESIZE_COEF_BITS * 2;
        int b0 = beta[0], b1 = beta[1];
        const int *S0 = src[0], *S1 = src[1];
        int x=0;
        for( ; x <= width - 4; x += 4 )
        {
            int t0, t1;
            t0 = S0[x]*b0 + S1[x]*b1;
            t1 = S0[x+1]*b0 + S1[x+1]*b1;

            dst[x] = imse_saturate_uchar(t0, SHIFT);
            dst[x+1] = imse_saturate_uchar(t1, SHIFT);

            t0 = S0[x+2]*b0 + S1[x+2]*b1;
            t1 = S0[x+3]*b0 + S1[x+3]*b1;

            dst[x+2] = imse_saturate_uchar(t0, SHIFT);
            dst[x+3] = imse_saturate_uchar(t1, SHIFT);
        }

        for( ; x < width; x++ )
        {
            dst[x] = imse_saturate_uchar(S0[x]*b0 + S1[x]*b1, SHIFT);
        }
    }


    /*! \fn void imse_HResizeLinear(const uint8_t** src, int** dst, int count,
            const int* xofs, const short* alpha,
            int swidth, int dwidth, int cn, int xmin, int xmax,const int ONE )
    \brief Perform horizontal Linear resize
    \param const uint8_t** src Source used for resizing.
    \param int** dst Destination of the resized data.
```

```
\param int count No description yet.
\param onst int* No description yet.
\param const short* alphat No description yet.
\param int swidth No description yet.
\param int xmin No description yet.
\param int xmax No description yet.
\param const int ONE No description yet.
*/
static void imse_HResizeLinear(const uint8_t** src, int** dst, int count,
        const int* xofs, const short* alpha,
        int swidth, int dwidth, int xmin, int xmax,const int ONE )
{   //ONE=INTER_RESIZE_COEF_SCALE
    int dx, k;
    int dx0 = 0;

    for( k = 0; k <= count - 2; k++ )
        {
            const uint8_t *S0 = src[k], *S1 = src[k+1];
            int *D0 = dst[k], *D1 = dst[k+1];
            for( dx = dx0; dx < xmax; dx++ )
            {
                int sx = xofs[dx];
                int a0 = alpha[dx*2], a1 = alpha[dx*2+1];

                int t0 = S0[sx]*a0 + S0[sx + 1]*a1;
                int t1 = S1[sx]*a0 + S1[sx + 1]*a1;

                D0[dx] = t0; D1[dx] = t1;
             }

             for( ; dx < dwidth; dx++ )
             {
                 int sx = xofs[dx];
                 D0[dx] = S0[sx]*ONE; D1[dx] = S1[sx]*ONE;
             }
        }

    for( ; k < count; k++ )
        {
            const uint8_t *S = src[k];
            int *D = dst[k];
            for( dx = 0; dx < xmax; dx++ )
            {
                int sx = xofs[dx];
                D[dx] = S[sx]*alpha[dx*2] + S[sx+1]*alpha[dx*2+1];
            }

            for( ; dx < dwidth; dx++ )
                D[dx] = S[xofs[dx]]*ONE;
        }
}

/*! \fn void imse_resizeGeneric( const imse_intMat src, imse_intMat dst,
                            const int* xofs, const void* _alpha,
                            const int* yofs, const void* _beta,
                            int xmin, int xmax, int ksize, const int ONE,
                            const int SHIFT)
\brief Generic functions used for scalling.
\param imse_intMat src  Pointer to the matrix form of source
\param imse_intMat  dst  Pointer to the matrix form of destination
\param const int* xofs    No description yet
```

```
    \param const void* _alpha  No description yet
    \param const void* _beta    No description yet
    \param int xmin              No description yet
    \param int xmax              No description yet
    \param int ksize              No description yet
    \param const int ONE         No description yet
    \param const int SHIFT       No description yet
    \warning    Support only linear resizing
    */
    static void imse_resizeGeneric( const imse_intMat src, imse_intMat dst,
                                const int* xofs, const void* _alpha,
                                const int* yofs, const void* _beta,
                                int xmin, int xmax, int ksize, const int ONE,
                                const int SHIFT)
    {
        const short* alpha = (const short*)_alpha;
        const short* beta = (const short*)_beta;
        const int MAX_ESIZE=16;

        int ssize_width= src.cols;
        int ssize_height= src.rows;
        int dsize_width= dst.cols;
        int dsize_height= dst.rows;

        int bufstep = (int)imse_alignSize(dsize_width, 16);
        int* _buffer;

        const uint8_t* srows[MAX_ESIZE]; //={0};
        int* rows[MAX_ESIZE]; //={0};
        int prev_sy[MAX_ESIZE];
        int k, dy;

        _buffer = (int *) imse_Malloc(bufstep*ksize*sizeof(int));

        for( k = 0; k < ksize; k++ )
        {
            prev_sy[k] = -1;
            rows[k] = (int*)_buffer + bufstep*k;
        }

        // image resize is a separable operation. In case of not too strong
        for( dy = 0; dy < dsize_height; dy++, beta += ksize )
        {
            int sy0 = yofs[dy], k, k0=ksize, k1=0, ksize2 = ksize/2;

            for( k = 0; k < ksize; k++ )
            {
                int sy = imse_clip(sy0 - ksize2 + 1 + k, 0, ssize_height);
                for( k1 =imse_IMAX(k1,k) ; k1 < ksize; k1++ )
                {
                    if( sy == prev_sy[k1] ) // if the sy-th row has been computed already,
reuse it.
                    {
                        if( k1 > k )
                            memcpy( rows[k], rows[k1], bufstep*sizeof(rows[0][0]) );
                        break;
                    }
                }
                if( k1 == ksize )
                            k0 = imse_IMIN(k0,k) ; // remember the first row that needs to be
computed
```

```
                srows[k] = (const uint8_t*)(src.data + src.step*sy);
                prev_sy[k] = sy;
            }

            if( k0 < ksize )
                imse_HResizeLinear( srows + k0, rows + k0, ksize - k0, xofs, alpha,
                        ssize_width, dsize_width, xmin, xmax, ONE);

          imse_VResizeLinear( (const int**)rows, (uint8_t*)(dst.data + dst.step*dy), beta,
dsize_width, SHIFT);
        }

        imse_Free(&_buffer);
    }



    /*! \fn void imse_resize_int_embb( const imse_intMat src, imse_intMat dst, int
dsize_width,
                int dsize_height, int interpolation )
    \brief Internal function called by imse_resize that contains the functionality of
           the scalling. This uses only integer numbers.
    \param imse_intMat src  Pointer to the matrix form of source
    \param imse_intMat dst  Pointer to the matrix form of destination
    \param int dsize_width   Destination matrix width
    \param int dsize_height  Destination matrix height
    \param int interpolation Type of interpolation
    \warning    Support only linear resizing and area resizing
    */
    static void imse_resize_int_embb (const imse_intMat src, imse_intMat dst, int dsize_width,
                                                        int dsize_height, int interpolation)
    {
        const int MAX_ESIZE = 16; //To be declared outside as a constant
        const int INTER_RESIZE_COEF_BITS = 11;  // Fixed value
        const int INTER_RESIZE_COEF_SCALE = 1 << INTER_RESIZE_COEF_BITS;

        int ssize_width;

        // To verify if dsize_width and dsize_height is greater than zero
        if (dsize_width==0 || dsize_height==0 || src.cols==0 || src.rows==0)
             return;

        ssize_width = src.cols;

        imse_create_intMat (dst, dsize_height, dsize_width, imse_MAT_TYPE (src.flags));

        int depth = imse_MAT_DEPTH (src.flags);
        int k, dx, dy;
        int ksize = 2, ksize2 = 1;

        bool fixpt = depth == imse_8U;
        short* embedd_alpha = (short*) imse_Malloc (dsize_width * ksize * sizeof (short));
        short* embedd_beta = (short*) imse_Malloc (dsize_height * ksize * sizeof (short));
        int* embedd_xofs = (int*) imse_Malloc (dsize_width * sizeof (int));
        int* embedd_yofs = (int*) imse_Malloc(dsize_height * sizeof (int));

        int embedd_x, embedd_y; // by Laur
        int embedd_cbuf [MAX_ESIZE]; //// by Laur   (double)dsize_height/src.rows
        const int IMSE_EXTRA_PRECISION_BITS = 4;
        const int IMSE_RESIZE_COEF_SCALE =  1 << (INTER_RESIZE_COEF_BITS +
IMSE_EXTRA_PRECISION_BITS);
```

```
        const int EMBEDD_MASK = IMSE_RESIZE_COEF_SCALE - 1;         // The mask used to remove
the integer part
        int shifted_scale_x = (IMSE_RESIZE_COEF_SCALE * src.cols) / dsize_width;
        int shifted_scale_y = (IMSE_RESIZE_COEF_SCALE * src.rows) / dsize_height;
        int aux, auy;
        int embedd_xmin = 0, embedd_xmax = dsize_width;

        for(dx = 0; dx < dsize_width; dx++)
        {
            aux = (shifted_scale_x * (2 * dx + 1) - IMSE_RESIZE_COEF_SCALE) >> 1;
            embedd_x = (aux & EMBEDD_MASK);

            // low importance.
            if ((aux >> (INTER_RESIZE_COEF_BITS + IMSE_EXTRA_PRECISION_BITS)) < ksize2 - 1)
            {
                embedd_xmin = dx + 1;
                if (aux < 0)
                    embedd_x = 0, aux = 0;
            }
            // low importance.
            if ((aux >> (INTER_RESIZE_COEF_BITS + IMSE_EXTRA_PRECISION_BITS)) + ksize2 >=
ssize_width)
            {
                embedd_xmax = imse_IMIN (embedd_xmax, dx);
                if ((aux >> (INTER_RESIZE_COEF_BITS + IMSE_EXTRA_PRECISION_BITS)) >=
ssize_width - 1)
                    embedd_x = 0, aux = (ssize_width - 1) << (INTER_RESIZE_COEF_BITS +
IMSE_EXTRA_PRECISION_BITS);
            }

            embedd_xofs [dx] = (aux >> (INTER_RESIZE_COEF_BITS + IMSE_EXTRA_PRECISION_BITS));
// by Laur

            embedd_cbuf [0]= IMSE_RESIZE_COEF_SCALE - embedd_x;
            embedd_cbuf [1]= embedd_x;

            if (fixpt)
            {
                for (k = 0; k < ksize; k++)
                {
                    embedd_alpha [dx * ksize + k] = imse_saturate_short_from_int (embedd_cbuf
[k] >> IMSE_EXTRA_PRECISION_BITS);
                }
            }
            else
            {
                printf("imse_resize_int(): Error in width. Only fixed point is supported \n");
                //for( k = 0; k < ksize; k++ )
                //    alpha[dx*cn*ksize + k] = cbuf[k];
                //for( ; k < cn*ksize; k++ )
                //    alpha[dx*cn*ksize + k] = alpha[dx*cn*ksize + k - ksize];
            }
        }

        for (dy = 0; dy < dsize_height; dy++)
        {
                auy = (shifted_scale_y * (2 * dy + 1) - IMSE_RESIZE_COEF_SCALE) >> 1;
                embedd_y = (auy & EMBEDD_MASK);

                embedd_yofs[dy] = (auy >> (INTER_RESIZE_COEF_BITS +
```

```
IMSE_EXTRA_PRECISION_BITS));
                embedd_cbuf [0]= IMSE_RESIZE_COEF_SCALE - embedd_y;
                embedd_cbuf [1]= embedd_y;

            if (fixpt)
            {
                for(k = 0; k < ksize; k++)
                    embedd_beta [dy * ksize + k] = imse_saturate_short_from_int (embedd_cbuf
[k] >> IMSE_EXTRA_PRECISION_BITS);
            }
            else
            {
                printf("imse_resize_int(): Error in height. Only fixed point is supported
\n");
                //for( k = 0; k < ksize; k++ )
                //    beta[dy*ksize + k] = cbuf[k];
            }
        }

        imse_resizeGeneric (src, dst, embedd_xofs, embedd_alpha, embedd_yofs, embedd_beta,
embedd_xmin,
                            embedd_xmax, ksize, INTER_RESIZE_COEF_SCALE, 2 *
INTER_RESIZE_COEF_BITS);

        imse_Free (&embedd_alpha);
        imse_Free (&embedd_xofs);
        imse_Free (&embedd_beta);
        imse_Free (&embedd_yofs);
    }


    /*! \fn void imse_Resize( const void* srcarr, void* dstarr, int method )
    \brief Resizes the image to defined dimension by dstarr
    \param void* srcarr Void Pointer to the source image
    \param void* dstarr Void Pointer to the destination image
    \warning    Support only linear resizing
    */
    static void
    imse_Resize (const void* srcarr, void* dstarr, int method)
    {
        imse_intMat src_laur, dst_laur;

        src_laur = imse_voidToIntMat (srcarr,0,0,0);
        dst_laur = imse_voidToIntMat (dstarr,0,0,0);

        assert(imse_MAT_TYPE(src_laur.flags) == imse_MAT_TYPE(dst_laur.flags) );
        imse_resize_int_embb( src_laur, dst_laur, dst_laur.cols,  dst_laur.rows, method );

    }

/////  End Resize section
////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////



/*****************************************************************************\
*                                Color to/from Grayscale                      *
\*****************************************************************************/
```

```
/*! \fn void imse_BGRx2Gray_8u_CnC1R( const uint8_t* src,
                      uint8_t* dst, int size_width,
                      int size_height, int src_cn )
    \brief Function that converts from format (r,g,b) to the grey scale.
           The conversion weights are  cscGr_32f, cscGg_32f, cscGb_32f
    \param uint8_t* src Pointer to the source data
    \param uint8_t* dst Pointer to the destination data
    \param int size_width  Width(cols) of the source
    \param int size_height  Height(rows) of the source
    \param int src_cn  Number of channels of the source.
    \warning None

*/
static void
imse_BGRx2Gray_8u_CnC1R( const uint8_t* src,
                      uint8_t* dst, int size_width,
                      int size_height, int src_cn )
{
    int i,t0;
    int tab[256*3];
    int r = 0, g = 0, b = (1 << (csc_shift-1));
    //pointer verification an srcn_cn
    if (src_cn==0)
            return; // error
    if (src==0 || dst==0 )
            return;


    //Preparing the LUT for color weights intensities
    for( i = 0; i < 256; i++ )
    {
        tab[i] = b;
        tab[i+256] = g;
        tab[i+512] = r;
        g += cscGg;
        b += cscGb;
        r += cscGr;

    }
    for( i = 0; i < size_width*size_height; i++, src += src_cn )
    {
        t0 = tab[src[0]] + tab[src[1] + 256] + tab[src[2] + 512];
        dst[i] = (uint8_t)(t0 >> csc_shift);
    }
}


/*! \fn void imse_CvtColor( const imse_Mat* src, imse_Mat* dst)
    \brief Function Wrapper that receives imse_Mat types and calls
           the function imse_CvtColor()
    \param imse_Mat* src Pointer to the source in imse_Mat form
    \param imse_Mat* dst Pointer to the destination in imse_Mat form

    \warning dst must be created an initialized before calling this function

*/
static void
imse_CvtColor( const imse_Mat* src, imse_Mat* dst)
{
//Must Extract the number of channels from the source flags
int src_cn=0;
```

```
    src_cn= imse_MAT_CN(src->type);
    imse_BGRx2Gray_8u_CnC1R( src->data_ptr,
                          dst->data_ptr, src->cols,
                          src->rows,src_cn );

    }

//+++++++++ End Color Conversion
//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


#endif /* EMBEDD_AUX_LBP_FUNC_HPP_ */
```