A recursive implementation of the wavelet transform with efficient use of memory

José Oliver and M.P. Malumbres

Abstract—**In this paper, a description of a new 2D wavelet transform algorithm is presented and implemented. The proposed algorithm requires less memory than the regular transform, since it processes the image line by line, such as other proposals do. However, it clearly specifies how to perform the synchronization among different buffer levels, so that an implementation can be easily written. This paper presents a general specification of the algorithm, which can be adapted to any programming language, and it also includes an Appendix with a compact and full implementation in C language, which can be used straightforwardly. Experimental results show that, using the 5-Megapixel standard image Woman, our recursive algorithm requires 125 times less memory and it is more than 3 times faster than the usual algorithm.**

Key words—**Wavelet Transform, efficient use of memory, efficient use of cache.**

I. INTRODUCTION

THE discrete wavelet transform (DWT) is a new mathematical tool that has aroused great interest in the last years due to its nice features (multiresolution, space and frequency domains, high compactness, etc). However, one of its major drawbacks is the high memory requirements of the regular algorithms that compute it. In [1] it is introduced a first solution to overcome this drawback for the 1D DWT. In [2], this transform is extended to image wavelet transform (2D) and other issues related to the order of the data are solved. However, it is not easy to implement this algorithm due to some unclear aspects. We will address them in Section 2, while Section 3 describes the proposed solution and Section 4 shows some results. Then, some conclusions are drawn, and an Appendix is included so that an implementation in C language is proposed. This C implementation serves a double purpose. On the one hand, it shows how compact and simple an implementation of our algorithm is, and on the other hand, it provides a direct implementation in one of the most used programming languages, proposing efficient solutions for some implementation details.

II. THE LINE-BASED APPROACH

The basic idea of our proposal is the use of a line-based strategy such as that used in [2], where the key idea for saving memory is to get rid of the wavelet coefficients as soon as they have been calculated. It is clearly different to the regular DWT, in which at every decomposition level, the image is transformed first line by line, and then row by row, and so it must be kept entirely in memory.

In order to keep in memory only the part of image strictly necessary, and therefore reduce the amount of memory required, the order of the regular wavelet algorithm must be changed.

For the first decomposition level, the algorithm directly receives image lines, one by one. On every input line, a one-level 1D wavelet transform algorithm is applied so that it is divided into two parts, representing the horizontal details and a low-frequency smaller version. Then, these transformed lines are stored in a buffer associated to the first decomposition level. This buffer must be able to keep 2N+1 lines, where 2N+1 is the number of taps for the largest analysis filter bank. We only consider odd filter lengths because they have higher compression efficiency, however this analysis could be extended to even filters as well.

When there are enough lines in the buffer to perform one step of a column wavelet transform, the convolution process is calculated vertically twice, first using the low-pass filter and then the high-pass filter. The result of this operation is the first line of the HL_1, LH_1 and HH_1 wavelet subbands, and the first line of the LL_1 subband.

At this moment, for a dyadic wavelet decomposition, we can process and release the first line of the wavelet subbands. However, the first line of the LL_1 subband does not belong to the final result, but it is needed as incoming data for the following decomposition level. On the other hand, once the lines in the buffer have been used, the buffer is shifted twice using rotation so that two lines are discarded and another two image lines are input at the other end. Once the buffer is updated, the process can be repeated and more lines are obtained.

At the second level, its buffer is filled with the LL_1 lines that have been computed in the first level. Once the buffer is completely filled, it is processed in the very same way as we have described for the first level. In this

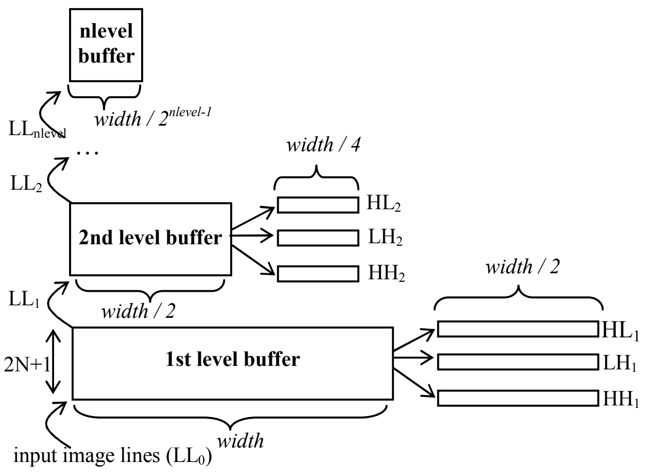

Figure 1: Overview of a line-based forward wavelet transform

```
function GetLLlineBwd( $level$ )
1) First base case: No more lines to read at this level
    if $LinesRead_{level} = MaxLines_{level}$
        return EOL
2) Second base case: The current level belongs to the space domain and not to the wavelet domain
    else if $level = 0$
        return ReadImageLineIO( )
    else
3) Recursive case
3.1) Recursively fill or update the buffer for this level
    if $buffer_{level}$ is empty
        for $i = N \ldots 2N$
            $buffer_{level}(i)$ = 1DFWT(GetLLlineBwd( $level$-1))
        FullSymmetricExtension( $buffer_{level}$ )
    else
        repeat twice
            Shift( $buffer_{level}$ )
            $line$ = GetLLlineBwd( $level$-1 )
            if $line$ = EOL
                $buffer_{level}(2N)$ = SymmetricExt( $buffer_{level}$ )
            else
                $buffer_{level}(2N)$ = 1DFWT( $line$ )
3.2) Calculate the WT from the lines in the buffer, then process the resulting subband lines (LL, HL, LH and HH)
    $\{LLline, HLline\}$ = ColumnFWT_LowPass( $buffer_{level}$ )
    $\{LHline, HHline\}$ = ColumnFWT_HighPass( $buffer_{level}$ )
    ProcessHighFreqSubLines( $\{HLline, LHline, HHline\}$ )
    set $LinesRead_{level} = LinesRead_{level} + 1$
    return $LLline$
end of fuction
```

Algorithm 1.1: Backward recursive function

manner, the lines of the second wavelet subbands are achieved, and the low-frequency lines from LL_2 are passed to the third level. As it is depicted in Figure 1, this process can be repeated until the desired decomposition level (*nlevel*) is reached.

Although this algorithm might seem quite simple, a major problem arises when it is implemented. This drawback is the synchronization among the buffers. Before a buffer can produce lines, it must be completely filled with lines from previous buffers, therefore they start working at different moments, i.e., they have different delays. Moreover, all the buffers exchange their result at different intervals, according to their level.

Handling several buffers with different delay and rhythm becomes a hard task. The next section proposes a recursive algorithm that clearly specifies how to perform this communication between buffers.

III. A RECURSIVE ALGORITHM FOR BUFFER SYNCHRONIZATION

In this section, we present both Forward and Inverse Wavelet Transform algorithms (FWT and IWT), which automatically solves the synchronization problem among levels that has been addressed in Section 2. In order to solve this problem, both algorithms define a recursive function that obtains the next low-frequency subband (LL) line from a contiguous level.

```
function WaveletTransform( $nlevel$ )
    set $LinesRead_{level} = 0 \quad \forall level \in nlevel$
    set $MaxLines_{level} = \frac{height}{2^{level}} \quad \forall level \in nlevel$
    set $buffer_{level} = empty \quad \forall level \in nlevel$
    repeat $\frac{height}{2^{nlevel}}$ times
        $LLline$ = GetLLlineBwd( $nlevel$ )
        ProcessLowFreqSubLine( $LLline$ )
end of function
```

Algorithm 1.2: Perform the FWT by calling a backward recursive function (see Algorithm 1.1)

A. FWT with Backward Recursion

The FWT starts requesting LL lines to the last level (*nlevel*). As seen in Figure 1, the *nlevel* buffer must be filled with lines from the *nlevel*-1 level before it can generate lines. In order to get them, the function recursively calls itself until the level 0 is reached. At this point, it no longer needs to call itself since it can return an image line that can be read directly from the input/output system. Notice that although we are calculating a forward wavelet transform, we do it by means of a backward recursion, since it goes from *nlevel* to 0.

The complete recursive algorithm is formally described in the frame entitled *Algorithm 1.1*, while *Algorithm 1.2* sets the variables up and performs the FWT by calling the recursive algorithm. Let us see the first algorithm more carefully.

The first time that the recursive function is called at every level, it has its buffer ($buffer_{level}$) empty. Then, its upper half (from N to 2N) is recursively filled with lines from the previous level. Recall that once a line is received, it must be transformed using a 1D FWT (either convolution [3] or lifting [4]) before it is stored. Once the upper half is full, the lower half is filled using symmetric extension (the N+1 line is copied into the N-1 position, …, the 2N is copied into the 0 position).

On the other hand, if the buffer is not empty, it simply has to be updated. In order to update it, it is shifted one position so that the line contained in the first position is discarded and a new line can be introduced in the last position (2N) using a recursive call. This operation is repeated twice.

However, if there are no more lines in the previous level, the recursive call will return *End Of Line* (EOL). That points out that we are about to finish the computation at this level, but we still need to continue filling the buffer. We fill it using symmetric extension again.

Once the buffer is filled or updated, both high-pass and low-pass filter banks are applied to every column in the buffer. As result of the convolution, we get a line of every wavelet subband at this level, and a LL line. The wavelet coefficients are processed according to the application (compressed, saved to secondary storage, etc.) and the function returns the LL line.

Every recursive function needs at least one base case to stop backtracking. This function has two base cases. The first case is when all the lines at this level have been

```
function GetLLlineFwd( level )
1) First base case: Buffer ready for another wavelet step
    if $lineinbuffer_{level}$
            set $lineinbuffer_{level} = false$
            set $LinesRead_{level} = LinesRead_{level} + 1$
            return ColumnIWT_HighPass( $buffer_{level}$ )
2) Second base case: All the lines have been read
    else if $LinesRead_{level} = MaxLines_{level}$
            return EOL
3) Third base case: The last level is accessed directly
    else if $level = nlevel$
            return RetrieveLowFreqSubLine ( )
    else
4) Recursive case
4.1) Recursively fill or update the buffer for this level
    if $buffer_{level}$ is empty
        for $i = N' \ldots 2N'$
            $buffer_{level}(i)$ = 1DIWT( BuildLine(level ) )
        FullSymmetricExtension( $buffer_{level}$ )
    else
        repeat twice
            Shift( $buffer_{level}$ )
            $line$ = BuildLine( $level$)
            if $line$ = EOL
                $buffer_{level}(2N')$ = SymmetricExt( $buffer_{level}$ )
            else
                $buffer_{level}(2N')$ = 1DIWT( $line$ )
4.2) Calculate one IWT step from the lines in the buffer
    set $lineinbuffer_{level} = true$
    set $LinesRead_{level} = LinesRead_{level} + 1$
    return ColumnIWT_LowPass( $buffer_{level}$ )
end of fuction
```

Algorithm 2.1: Forward recursive function

read. It is detected by keeping an account of the number of lines read and the maximum number of lines that can be read at every level. In this case, the function returns EOL. The second base case is achieved when the level reaches 0 and then no further recursive call is need since an image line can be read directly.

B. IWT with Forward Recursion

The IWT algorithm is described in Algorithms *2.1, 2.2* and *2.3*. This algorithm receives the subband lines that have been calculated in the FWT, and it recovers the original image lines. Both algorithms 2.1 and 2.3 are similar to the corresponding FWT 1.1 and 1.2, however some differences need to be explained.

The main difference is that the recursion is carried out forward, starting from 0 up to *nlevel*, and at that level the LL_{nlevel} subband lines are retrieved directly.

Moreover, in Algorithm 2.1, the lines for the buffers are obtained using a subfunction described in Algorithm 2.2. This function iteratively returns the concatenation of a line from the LL and from the HL subbands, or the concatenation of a line from LH and HH. Notice that the lines from HL, LH and HH are retrieved directly from the input data associated to that level, but the LL line has to be achieved recursively from the following level.

The last difference is that once a buffer is full, two LL lines can be returned, one line for every column

```
subfunction BuildLine(level )
    if $OddAccess_{level}$
        $OddAccess_{level} = false$
        $LLline =$ GetLLlineFwd($level$+1)
        $HLline =$ RetrieveHL_SubLine(level)
        return $LLline + HLline$
    else
        $OddAccess_{level} = true$
        $\{LHline, HHline\}$=RetrieveLH_HH_SubLines(level)
        return $LHline + HHline$
end of subfunction
```

Algorithm 2.2: Subfuction used for Algorithm 2.1

```
function InvWaveletTransfort( nlevel )
    set $LinesRead_{level} = 0 \quad \forall level \in nlevel$
    set $MaxLines_{level} = \frac{height}{2^{level}} \quad \forall level \in nlevel$
    set $buffer_{level} = empty \quad \forall level \in nlevel$
    set $lineinbuffer_{level} = false \quad \forall level \in nlevel$
    set $OddAccess_{level} = true \quad \forall level \in nlevel$
    repeat $height$ times
            $imageLine$ = GetLLlineFwd( 0 )
            WriteImageLineIO( $imageLine$ )
end of function
```

Algorithm 2.3: Perform the IWT by calling a forward recursive function (see Algorithm 2.1)

convolution. For this reason, $lineinbuffer_{level}$ shows if there is any line in the buffer that can be directly returned. It becomes the first base case in *Algorithm* 2.1.

C. Some theoretical considerations

The main advantage of line-based algorithms is its lower memory requirement compared to the regular wavelet transform. In these algorithms, every buffer needs to store $(2N+1)\times BufferWidth$ coefficients. If the image width is *w*, then the memory requirements for all the buffer is $(2N+1)\times w + (2N+1)\times w/2 + \ldots + (2N+1)\times w/2^{nlevel}$, which is asymptotically $2\times(2N+1)\times w$, considerably lower than the $width \times height$ coefficients required by the regular WT. This reduction in the amount of memory has another beneficial side effect when the algorithm is implemented in a computer. The subband buffers are more likely to fit in the cache memory than the whole image, and thus the execution time is substantially reduced.

A drawback that has not been considered yet is the need to reverse the order of the subbands, from the FWT to the IWT. The former starts generating lines from the first levels to the last ones, while the latter requires lines from the last levels before those from the first ones. This problem can be resolved using some buffers at both ends, so that data are supplied in the right order [2]. Other simpler solutions are: to save every level in secondary storage separately so that it can be read in a different order and, if the WT is used for image compression, to keep the compressed coefficients in memory.

IV. EXPERIMENTAL RESULTS

We have implemented the regular Wavelet Transform and our proposal with the B7/9 filter bank [5], using standard ANSI C language (as it is shown in Appendix) on a regular PC computer with 256 KB L2 cache. This implementation can be downloaded at http://www.disca.upv.es/joliver/WT.

We have used the standard Lena (512x512) and Woman (2048x2560) images. With six decomposition levels, our algorithm requires 40 KB for Lena and 162 KB for Woman, while the regular WT needs 1030 KB for Lena and 20510 KB for Woman, i.e., it uses 25 and 127 times more memory. In addition, Table 1 shows that our algorithm is much more scalable than the usual WT. An execution time comparison between both algorithms can be seen in Figure 2. It shows that, while our algorithm has a linear behavior, the regular WT approaches to an exponential curve. This behavior happens because our algorithm fits in cache for all the image sizes (162 KB for the 5-megapixel image). On the contrary, the usual WT rapidly exceeds the cache limits (1287 KB for the VGA resolution).

Further experiments have shown that the IWT has similar results in memory requirement and execution time.

TABLE 1

MEMORY REQUIREMENT (KB) COMPARISON.

Image size (megapixel)	Regular WT	Proposed WT
5 (2048 x 2560)	20510	162
4 (1856 x 2240)	16266	146
3 (1600 x 1984)	12423	126
2 (1280 x 1664)	9339	101
1 (1024 x 1280)	5135	80
VGA (512x640)	1287	40

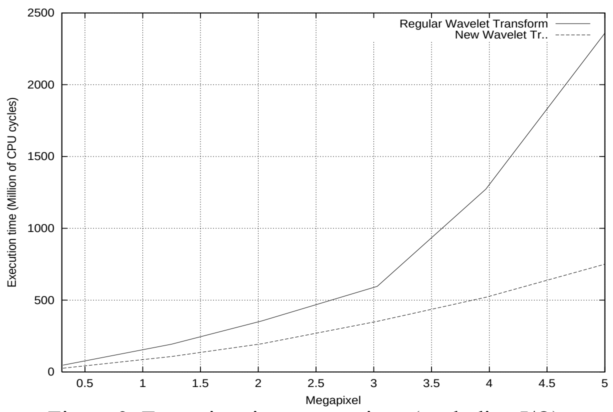

Figure 2: Execution time comparison (excluding I/O)

V. CONCLUSIONS

A line-by-line transform algorithm has been presented that solves the existing problem about different delay and rhythm among the buffers. It can be used as a part of compression algorithms such as JPEG 2000, speeding up its execution time and reducing its memory requirements.

VI. REFERENCES

[1] M. Vishwanath, "The recursive pyramid algorithm for the discrete wavelet transform," *IEEE Tr. Signal Proc.*, Mar. 1994

[2] C. Chrysafis, and A. Ortega, "Line-based, reduced memory, wavelet image compression," *IEEE Tr. Image Proc.*, Mar. 2000

[3] S. Mallat, "A theory for multiresolution signal decomposi-tion," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, July 1989

[4] Sweldens, "The lifting scheme: a custom-design construction of biorthogonal wavelets," *Appl. Comput. Harmon. Anal.*, 1996

[5] M.Antonini, Barlaud, P.Mathieu, and I.Daubechies, "Image coding using wavelet transform," *IEEE Tr. Image Proc.*, 1992

VII. APPENDIX

In this Section, an operative and optimized implementation of the proposed algorithm is given in ANSI C language.

In the first two Subsections, the FWT is implemented. Subsection A implements the recursive backward function that is used from Subsection B in order to compute the Wavelet Transform. In the following two Subsections, the inverse transform is implemented in a similar way. In Subsection E, some functions and variables that are used by the previous functions are given. Finally, some useful function headers are defined.

A. Backward Recursion Function

```
int GetLLlineBwd(int step, int width, float *Dest)
{
int f,g;
float **BuffLevel=BufferTransf[step];
if (step<0)
   return(ReadImageLine(Dest));
if (SymmetryPoint[step]<=NTaps)
   return EOL;
if (!BuffLevel[0])
{
   for (f=0;f<BufferSize;f++)
      BuffLevel[f]=
         (float *)malloc(width*sizeof(float));
   for (f=0;f<NTaps+1;f++)
   {
   GetLLlineFwd(step-1, width*2, BuffLevel[f+NTaps]);
   LineTransform(width,BuffLevel[f+NTaps]);
   }
   for(f=NTaps-1,g=NTaps+1;f>=0;f--,g++)
      LineCopy(width,BuffLevel[g],BuffLevel[f]);
}
else repeat(2)
{
   ShiftLines(BuffLevel);
   If (GetLLlineFwd(step-1,
            width*2,BuffLevel[BufferSize-1])!=EOL)
      LineTransform(width,BuffLevel[BufferSize-1]);
   else
   {
      LineCopy(width,BuffLevel[SymmetryPoint[step]-
            Radious[step]],BuffLevel[BufferSize-1]);
      Radious[step]++,SymmetryPoint[step]--;
   }
}
ColumnTransformLow(width, BuffLevel, Dest, HL);
ColumnTransformHigh(width, BuffLevel, LH, HH);
ProcessSubbands(HL,LH,HH,step);
return OK;
}
```

B. Implementation of the Wavelet Transform

```
int WaveletTransform(int Nsteps,int width,int height)
{
int f,g;
NTapsMax=NTaps>NTapsInv?NTaps:NTapsInv;
BufferSize=(NTapsMax<<1)+1;

CoefExt=(float *)
   malloc((width+(NTapsMax<<1))*sizeof(float));
CoefExtIni=CoefExt+NTapsMax;
BufferTransf=(float***)
   malloc(Nsteps*sizeof(float(**)));
for (f=0;f<Nsteps;f++)
   BufferTransf[f]=(float**)
      malloc(BufferSize*sizeof(float(*)));
for (f=0;f<Nsteps;f++)
   for (g=0;g<BufferSize;g++)
      BufferTransf[f][g]=NULL;
SymmetryPoint=(int *)malloc(Nsteps*sizeof(int));
for (f=0;f<Nsteps;f++)
   SymmetryPoint[f]=BufferSize-2;
Radious=(int *)malloc(Nsteps*sizeof(int));
for (f=0;f<Nsteps;f++)
   Radious[f]=1;
LL=(float *)malloc((width>>Nsteps)*sizeof(float));
LH=(float *)malloc((width>>1)*sizeof(float));
HL=(float *)malloc((width>>1)*sizeof(float));
HH=(float *)malloc((width>>1)*sizeof(float));

for (f=0;f<(height/(1<<Nsteps));f++)
{
   GetLLlineFwd(Nsteps-1, width/(1<<(Nsteps-1)), LL);
   ProcessLLSubband(LL);
}

for (f=0;f<Nsteps;f++)
   for (g=0;g<BufferSize;g++)
      free(BufferTransf[f][g]);
free(LL);free(HL);free(LH);free(HH);
free(Radious); free(SymmetryPoint);
for (f=0;f<Nsteps;f++)
   free(BufferTransf[f]);
free(BufferTransf); free(CoefExt);

return 0;
}
```

C. Forward Recursion Function

```
int GetLLlineFwd(int step, int width, float *Dest)
{
int f,g;
float **BuffLevel=BufferTransf[step];
int halfwidth=width/2;
if (step>lastStep) return ReadLLline(Dest);
if (LinesInBuffer[step])
{
   InvColumnTransformLow(width, BuffLevel, Dest);
   LinesInBuffer[step]=0;
   return OK;
}
if (SymmetryPoint[step]<=NTapsInv) return EOL;
if (!BuffLevel[0])
{
   for (f=0;f<BufferSize;f++)
      BuffLevel[f]=(float *)
         malloc(width*sizeof(float));
   for (f=NTapsInv;f<BufferSize;)
   {
      GetLLlineBwd(step+1, halfwidth, BuffLevel[f]);
      ReadSubbandLine(
            BuffLevel[f]+halfwidth,
            BuffLevel[f+1],
            BuffLevel[f+1]+halfwidth,
            step);
      InvLineTransform(width,BuffLevel[f++]);
      InvLineTransform(width,BuffLevel[f++]);
   }
   for (f=NTapsInv-1,g=NTapsInv+1;f>=0;f--,g++)
      LineCopy(width,BuffLevel[g],BuffLevel[f]);
}
else
{
   ShiftLines(BuffLevel);
   ShiftLines(BuffLevel);
   if (GetLLlineBwd(step+1,
      halfwidth, BuffLevel[BufferSize-2])==OK)
   {
     ReadSubbandLine(
            BuffLevel[BufferSize2]+halfwidth,
            BuffLevel[BufferSize-1],
            BuffLevel[BufferSize-1]+halfwidth
            step);
     InvLineTransform(width,BuffLevel[BufferSize-2]);
     InvLineTransform(width,BuffLevel[BufferSize-1]);
   }
   else
```

```
   {
      LineCopy(width,BuffLevel[SymmetryPoint[step]-
         Radious[step]-1],BuffLevel[BufferSize-2]);
      LineCopy(width,BuffLevel[SymmetryPoint[step]-
         Radious[step]-2],BuffLevel[BufferSize-1]);
      Radious[step]+=2;
      SymmetryPoint[step]-=2;
   }
}
InvColumnTransformHigh(width, BuffLevel, Dest);
LinesInBuffer[step]=1;
return OK;
}
```

D. Implementation of the Inverse Wavelet Transform

```
int InvWaveletTransform(int Nsteps,
                                int width,int height)
{
float *ImageLine;
int f,g;

NTapsMax=NTaps>NTapsInv?NTaps:NTapsInv;
BufferSize=(NTapsMax<<1)+1; lastStep=Nsteps-1;
CoefExt=(float *)
   malloc((width+(NTapsMax<<1))*sizeof(float));
CoefExtIni=CoefExt+NTapsMax;
BufferTransf=(float***)
   malloc(Nsteps*sizeof(float(**)));
for (f=0;f<Nsteps;f++)
   BufferTransf[f]=(float **)
      malloc(BufferSize*sizeof(float(*)));
for (f=0;f<Nsteps;f++)
   for (g=0;g<BufferSize;g++)
      BufferTransf[f][g]=NULL;
SymmetryPoint=(int *)
   malloc(Nsteps*sizeof(int));
for (f=0;f<Nsteps;f++)
   SymmetryPoint[f]=BufferSize-3;
Radious=(int *)malloc(Nsteps*sizeof(int));
for (f=0;f<Nsteps;f++)
   Radious[f]=0;
LinesInBuffer=(int *)malloc(Nsteps*sizeof(int));
for (f=0;f<Nsteps;f++)
   LinesInBuffer[f]=0;
ImageLine=(float *)malloc(width*sizeof(float));
for (f=0;f<height;f++)
{
   GetLLlineBwd(0, width, ImageLine);
   ProcessLine(ImageLine);
}
for (f=0;f<Nsteps;f++)
   for (g=0;g<BufferSize;g++)
      free(BufferTransf[f][g]);
free(ImageLine); free(LinesInBuffer);
free(Radious); free(SymmetryPoint);
for (f=0;f<Nsteps;f++)
   free(BufferTransf[f]);
free(BufferTransf); free(CoefExt);
return 0;
}
```

E. Auxiliary Functions and Global Variables

In order to get a complete module for the proposed Wavelet Transform, the previous functions can be appended to the functions and variables defined in this Subsection.

```
#include <stdlib.h>
#include <string.h>
#include “external_headers.h”

#define OK  0
#define EOL 1

static float ***BufferTransf;
static float *CoefExt,*CoefExtIni;
static int *SymmetryPoint, *Radious, *LinesInBuffer;
static int BufferSize,NTapsMax,lastStep;
static float *LL,*HL,*LH,*HH;

static int NTaps=4;
static int NTapsInv=3;
static float B79_AnLowPass[]=
   {0.85269868F, 0.37740286F,
   -0.11062440F, -0.02384947F, 0.03782846F};
static float B79_AnHighPass[]=
   {0.78848562F, -0.41809227F, -0.04068942F,
   0.06453888F};
static float B79_SynLowPass[]=
   {0.85269868F, 0.41809227F, -0.11062440F,
    -0.06453888F, 0.03782846F};
static float B79_SynHighPass[] =
   {0.78848562F,-0.37740286F,-0.04068942F,
   0.02384947F};
#define repeat(x) for (int __indx=0;__indx<x;__indx++)

#define LineCopy(n, source, dest) \
   memcpy(dest,source,n*sizeof(float))

inline void InterleaveLine
                 (float *Src,float *Dest, int width)
{
   float *HalfSrc=Src+(width>>1);
   for (int x=0;x<(width>>1);x++)
   { *Dest++=*Src++;  *Dest++=*HalfSrc++; }
}

inline void ShiftLines(float **Buff)
{
   float *Aux=Buff[0];
   for (int f=0;f<BufferSize-1;f++)
      Buff[f]=Buff[f+1];
   Buff[BufferSize-1]=Aux;
}

inline void SymmetricExt(float *ini,float *end)
{
   for (int x=1;x<=NTapsMax;x++)
   {
      ini[-x]=ini[x];
      end[x]=end[-x];
   }
}

inline float FourTapsFilter(float *c,float *t)
{
   return (
      t[0]*c[0]+
      t[1]*(c[1]+c[-1])+
      t[2]*(c[2]+c[-2])+
      t[3]*(c[3]+c[-3]));
}

inline float FiveTapsFilter(float *c,float *t)
{
   return (
      t[0]*c[0]+
      t[1]*(c[1]+c[-1])+
      t[2]*(c[2]+c[-2])+
      t[3]*(c[3]+c[-3])+
      t[4]*(c[4]+c[-4]));
}

void LineTransform(int width, float *Line)
{
float *CoefAuxL,*CoefAuxH;
LineCopy(width,Line,CoefExtIni);
SymmetricExt(CoefExtIni,CoefExtIni+width-1);
CoefAuxL=Line; CoefAuxH=Line+(width>>1);
for (int x=0;x<width;)
{
   *CoefAuxL++=
      FiveTapsFilter(CoefExtIni+x,B79_AnLowPass);
   x++;
   *CoefAuxH++=
      FourTapsFilter(CoefExtIni+x,B79_AnHighPass);
   x++;
}
}

void InvLineTransform(int width, float *Line)
{
InterleaveLine(Line,CoefExtIni, width);
SymmetricExt(CoefExtIni,CoefExtIni+width-1);
for (int x=0;x<width;)
{
   *Line++=
      FourTapsFilter(CoefExtIni+x,B79_SynHighPass);
   x++;
   *Line++=
      FiveTapsFilter(CoefExtIni+x,B79_SynLowPass);
   x++;
}
}

inline float FiveTapsColumnFilter
                 (float *t, float **B, int x)
{
   return (
      t[0]*B[4][x]+
      t[1]*(B[3][x]+B[5][x])+
      t[2]*(B[2][x]+B[6][x])+
      t[3]*(B[1][x]+B[7][x])+
      t[4]*(B[0][x]+B[8][x]));
}

inline float FourTapsColumnFilter
                 (float *t, float **B, int x)
{
  return (
      t[0]*B[5][x]+
      t[1]*(B[6][x]+B[4][x])+
      t[2]*(B[7][x]+B[3][x])+
      t[3]*(B[8][x]+B[2][x]));
}
```

```
void ColumnTransformLow(int width, float **BuffLevel,
float *LL, float *HL)
{
int x=0;
while (x<width/2)
   *LL++=FiveTapsColumnFilter
            (B79_AnLowPass, BuffLevel, x++);
while (x<width)
   *HL++=FiveTapsColumnFilter
            (B79_AnLowPass, BuffLevel, x++);
}

void ColumnTransformHigh(int width, float **BuffLevel,
float *LH, float *HH)
{
int x=0;
while (x<width/2)
   *LH++=FourTapsColumnFilter
            (B79_AnHighPass, BuffLevel, x++);
while (x<width)
   *HH++=FourTapsColumnFilter
            (B79_AnHighPass, BuffLevel, x++);
}

void InvColumnTransformHigh (int width, float
**BuffLevel, float *LL)
{
for (int x=0;x<width;x++)
   *LL++=FourTapsColumnFilter
            (B79_SynHighPass, BuffLevel-2, x);
}

void InvColumnTransformLow(int width, float
**BuffLevel, float *LL)
{
    for (int x=0;x<width;x++)
       *LL++=FiveTapsColumnFilter
            (B79_SynLowPass, BuffLevel, x);
}
```

F. External Headers

Some functions are dependant on the final application and are left as external and open functions. In general, these functions are used to read data (*ReadImageLine()*, *ReadLLline()* *ReadSubbandLine()*) or process the generated data (*ProcessSubbands()*, *ProcessLLSubband()*, *ProcessLine()*) line-by-line.

```
/* EXTERNAL FUNCTIONS USED BY THE FWT */

/* Read an image line on Dest */
int ReadImageLine(float *Dest);

/*  Process  or  store  one  line  achieved  from  every
wavelet subband (HL, LH and HH) at level=step */
void ProcessSubbands
   (float *HL,float *LH,float *HH,int step);

/*  Process  or  store  a  line  achieved  from  the  LL
subband */
void ProcessLLSubband(float *LL);

/* EXTERNAL FUNCTIONS USED BY THE IWT */

/* Read a line from the LL subband */
int ReadLLline(float *Dest);

/* Read a line from every wavelet subband (HL, LH and
HH) at level = step */
void ReadSubbandLine
   (float *HL, float *LH, float *HH, int step);

/* Process or store an achieved image line */
void ProcessLine(float *ImageLine);
```