UPEM - ESIEE PARIS

M2 SIS - Représentation et filtrage numérique 1D/2D : implémentation de l'algorithme jpeg2000

TOFFANIN Marc

8 janvier 2016

1 INTERPOLATION ET DECIMATION

```
//Interpolate (factor 2) in place a signal of size N
void interp2( double* x, int N )
{
    for (int i = N/2 ; i > 0 ; i--)
    {
        x[2*i] = x[i];
        x[i] = 0;
    }
}

//Decimate (factor 2) in place a signal of size N
void decim2( double* x, int N )
{
    for (int i = 1 ; i <= N/2 ; i++)
    {
        x[i] = x[2*i];
        x[2*i] = 0;
    }
}
```

2 FILTRAGE

2.1 CONVOLUTION

```
//Convolve a signal of size N with a filter of size K
// getIdx give back mirror symetry index
double* conv( double* x, int N, double* h, int K )
{
    double* new_sig = new double[N];
    int shift = K/2;
    for (int i = 0; i < N ; i++)
    {
        double sum = 0;
        for (int k = 0 ; k < K ; k++)
        {
            int index = i - (k - shift);
            sum += h[k] * x[getIdx(index,N)];
        }
        new_sig[i] = sum;
    }
    return new_sig;
}
```

2.2 BANC DE FILTRE DE HAAR

```
//Haar analysis of a signal of size p
void analyse_haar( double** x, int p)
{
    assert( (p%2) == 0 && "␣p␣must␣be␣a␣power␣of␣2␣" );
    double h0[3] = { 1/sqrt(2), 1/sqrt(2), 0 };
    double h1[3] = { 1/sqrt(2), -1/sqrt(2), 0};
    double* xb = conv(*x, p, h0, 3);
    double* xh = conv(*x, p, h1, 3);
    decim2( xb, p );
    decim2( xh, p );
    *x = concat( xb, p/2, xh, p/2);
}

//Haar synthesis of a signal of size p
double* synthese_haar(double* x, int p)
{
    assert( (p%2) == 0 && "␣p␣must␣be␣a␣power␣of␣2␣" );
    double g0[3] = { 0, 1/sqrt(2), 1/sqrt(2) };
    double g1[3] = { 0, -1/sqrt(2), 1/sqrt(2) };
    double * xbd = new double[p];
    double * xhd = new double[p];
    for ( int i = 0 ;
          i < p ;
          ++i)
    {
        if ( i < (p/2))
        {
            xbd[i] = x[i];
        }
        else
        {
            xhd[i -(p/2)] = x[i];
        }
```

```
    }
    interp2( xbd, p);
    interp2( xhd, p);
    double* yb = conv( xbd, p , g0, 3);
    double* yh = conv( xhd, p , g1, 3);
    add_tab( yb, p, yh, p);
    return yb;
}
```

Etudions les coefficients pour la fonction rampe, la plus simple à analyser. Regardons pour la première valeur.
Le coefficient d'approximation est égal à celui de détail, soit 0,7. Ceci parait tout à fait logique étant donné les coefficient du filtre utilisé de

$$\frac{1}{\sqrt{2}} = 0.7$$

La moyenne étant

$$0 * 0,7 + 1 * 0,7 = 0.7$$

. Les coefficients de détails sont tous égaux à 0,7 car sur la fonction rampe il y as toujours une différence de 1 entre n et n+1, ce qui donne

$$-n * 0,7 + (n+1) * 0,7 = 0,7$$

Pour la synthèse ont peu observer que

$$0 = 0 * 0.7 + 0 * 0,7 + 0.7 * 0.7 - 0.7 * 0.7$$

et

$$1 = 0 * 0,7 + 0.7 * 0,7 + 0.7 * 0.7 - 0 * 0.7$$

Après reconstruction, on trouve une erreur quadratique moyenne de 2.25617e-23 sur le signal leleccum, indiquant une reconstruction quasi-parfaite.

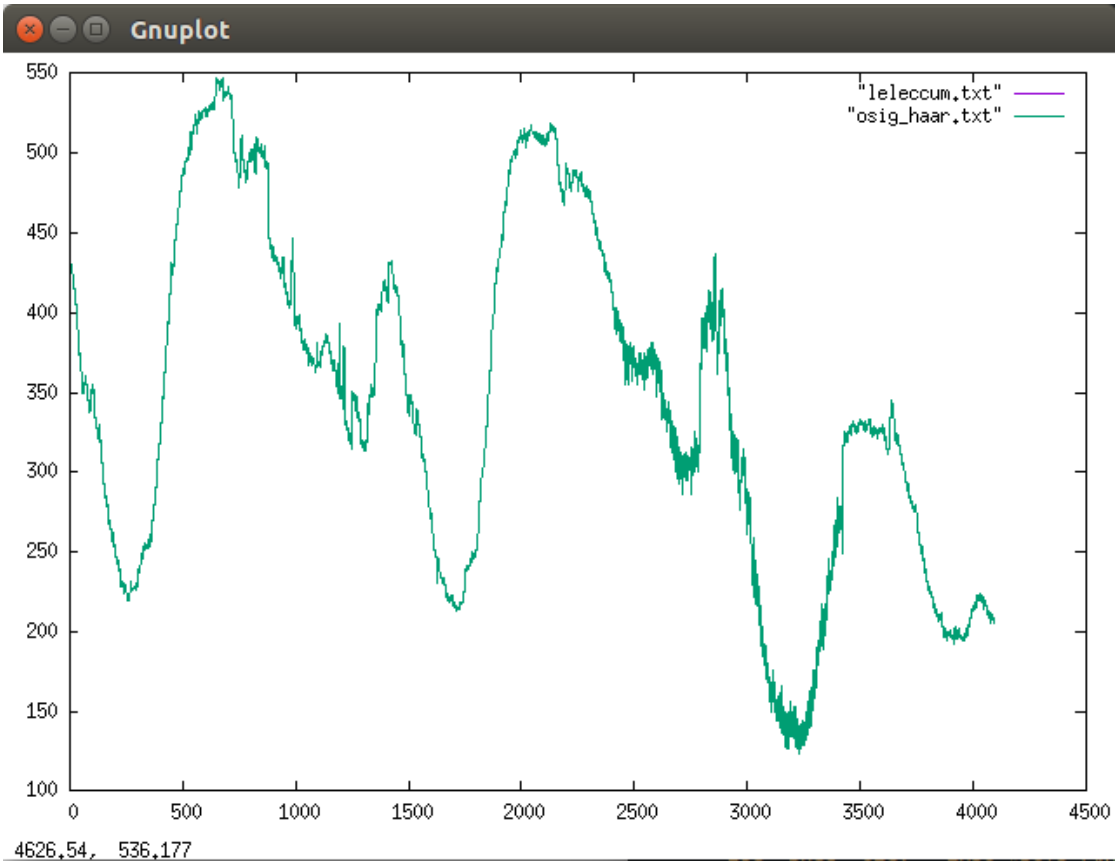

FIGURE 2.1 – La courbe initiale et reconstruite se recouvrent, indiquant une reconstruction quasi-parfaite avec le banc de filtres de Haar

2.3 BANC DE FILTRE BIORTHOGONAUX 9/7

```
void analyse_97( double** x, int p )
{
    assert( (p%2) == 0 && "␣p␣must␣be␣a␣power␣of␣2␣" );
    double* _h0 = new double[9];
    double* _h1 = new double[9];

    // Filtre biorthogonal 9/7 _h0 (longueur 9)
    _h0[0]=0.037828455507;
    _h0[1]=-0.023849465019;
    _h0[2]=-0.110624404418;
    _h0[3]=0.377402855613;
    _h0[4]=0.852698679009;
    _h0[5]=0.377402855613;
    _h0[6]=-0.110624404418;
    _h0[7]=-0.023849465019;
    _h0[8]=0.037828455507;
```

```
    // Filtre biorthogonal 9/7 _h1 (longueur 9)
    _h1[0]=0.064538882629;
    _h1[1]=-0.040689417610;
    _h1[2]=-0.418092273222;
    _h1[3]=0.788485616406;
    _h1[4]=-0.418092273222;
    _h1[5]=-0.040689417610;
    _h1[6]=0.064538882629;
    _h1[7]=0.000000000000;
    _h1[8]=-0.000000000000;

    double* xb = conv(*x, p, _h0, 9);
    double* xh = conv(*x, p, _h1, 9);
    decim2( xb, p );
    decim2( xh, p );
    *x = concat( xb, p/2, xh, p/2);
}

double* synthese_97( double* x, int p )
{
    assert( (p%2) == 0 && "␣p␣must␣be␣a␣power␣of␣2␣" );
    double* _g0 = new double[7];
    double* _g1 = new double[11];

    // Filtre biorthogonal 9/7 _g0 (longueur 7)
    _g0[0]=-0.064538882629;
    _g0[1]=-0.040689417610;
    _g0[2]=0.418092273222;
    _g0[3]=0.788485616406;
    _g0[4]=0.418092273222;
    _g0[5]=-0.040689417610;
    _g0[6]=-0.064538882629;

    // Filtre biorthogonal 9/7 _g1 (longueur 11)
    _g1[0]=0.000000000000;
    _g1[1]=-0.000000000000;
    _g1[2]=0.037828455507;
    _g1[3]=0.023849465019;
    _g1[4]=-0.110624404418;
    _g1[5]=-0.377402855613;
    _g1[6]=0.852698679009;
    _g1[7]=-0.377402855613;
    _g1[8]=-0.110624404418;
    _g1[9]=0.023849465019;
    _g1[10]=0.037828455507;

    double * xbd = new double[p];
    double * xhd = new double[p];

    for ( int i = 0;
          i < p ;
          ++i)
    {
        if ( i < (p/2))
```

```
        {
            xbd[i] = x[i];
        }
        else
        {
            xhd[i - (p/2)] = x[i];
        }
    }

    interp2( xbd, p);
    interp2( xhd, p);
    double* yb = conv( xbd, p , _g0, 7);
    double* yh = conv( xhd, p , _g1, 11);
    add_tab( yb, p, yh, p);
    return yb;
}
```

Pour les filtres biorthogonaux on peut observer que g1 = [0, 0, h0] et h1 = [g0, 0, 0]. Ceci est du à la biorthogonalité des filtres.

De manière analogue à celle vu précedemment, on trouve une erreur quadratique moyenne de 0.654779, ce qui peut paraitre peu mais est en fait énorme comparé à ce que nous aurions du trouver. En analysant le signal reconstruit il s'avère que très proche de la fin l'erreur augmente, surement du à une erreur dans le code, que nous n'avons pas réussi à trouver.

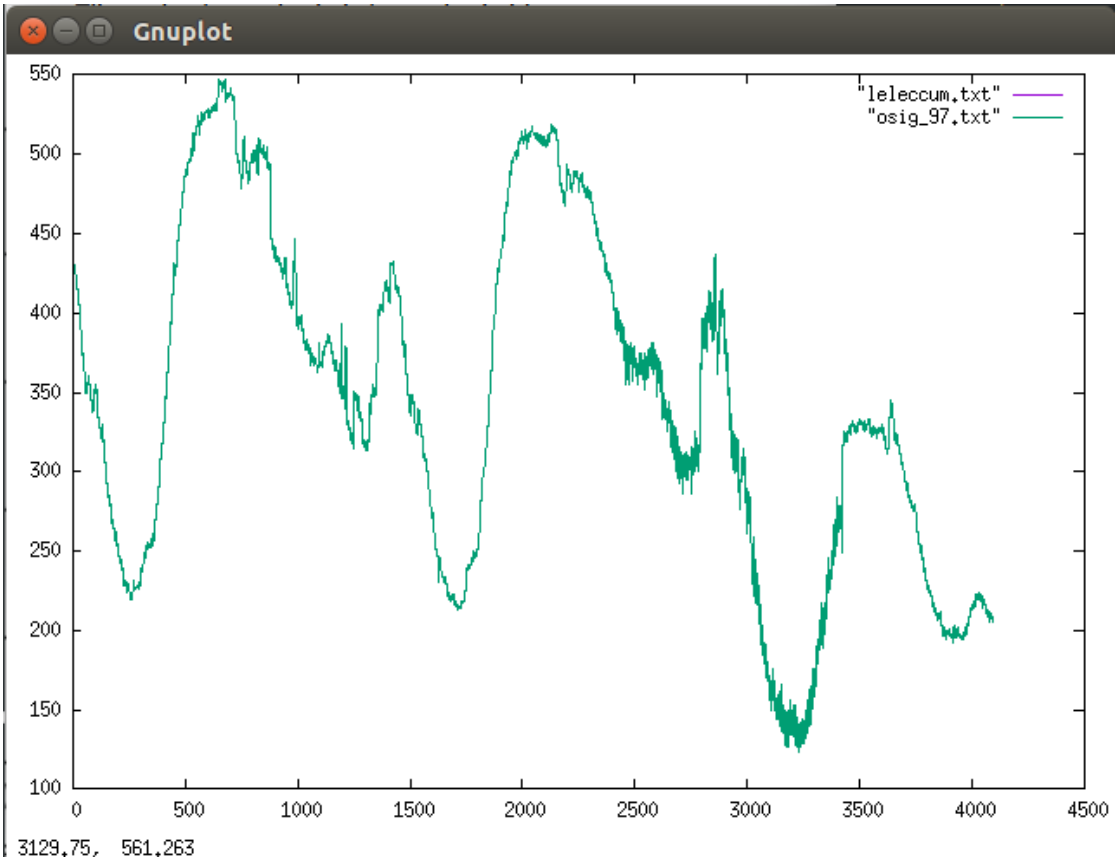

FIGURE 2.2 – La courbe initiale et reconstruite se recouvrent, indiquant une reconstruction quasi-parfaite avec le banc de filtres biorthogonaux 9/7

Par rapport au banc de filtre de Haar, les coefficients sont plus petit en moyenne et sont parfois égaux à zéro.

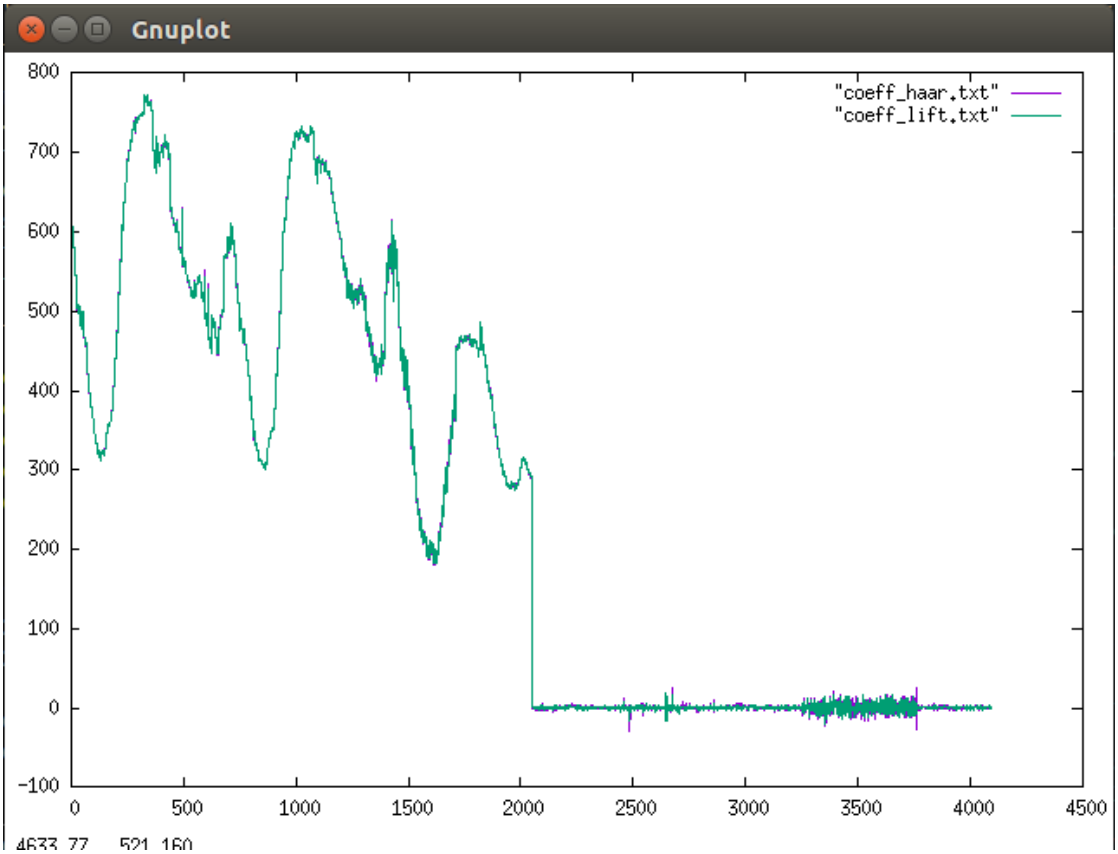

FIGURE 2.3 – Courbe de comparaisons entre les coefficients issus de Haar (violet) et des biorthogonaux 9/7 (vert)

2.4 LIFTING BIORTHOGONAUX 9/7

Les coefficients d'analyse ont l'air globalement identiques à ceux issus du banc de filtre. Cependant, je ne peux pas vérifier avec une erreur quadratique (4.69792e-10 dans mon cas) puisque ma version de banc de filtre 9/7 me parait fausse. Je pense qu'il n'y a quasiment pas de différence entre les deux en temps normal.
L'erreur entre l'image reconstruite et originale est de 2.13235e-23, soit moins qu'avec la décomposition de Haar.

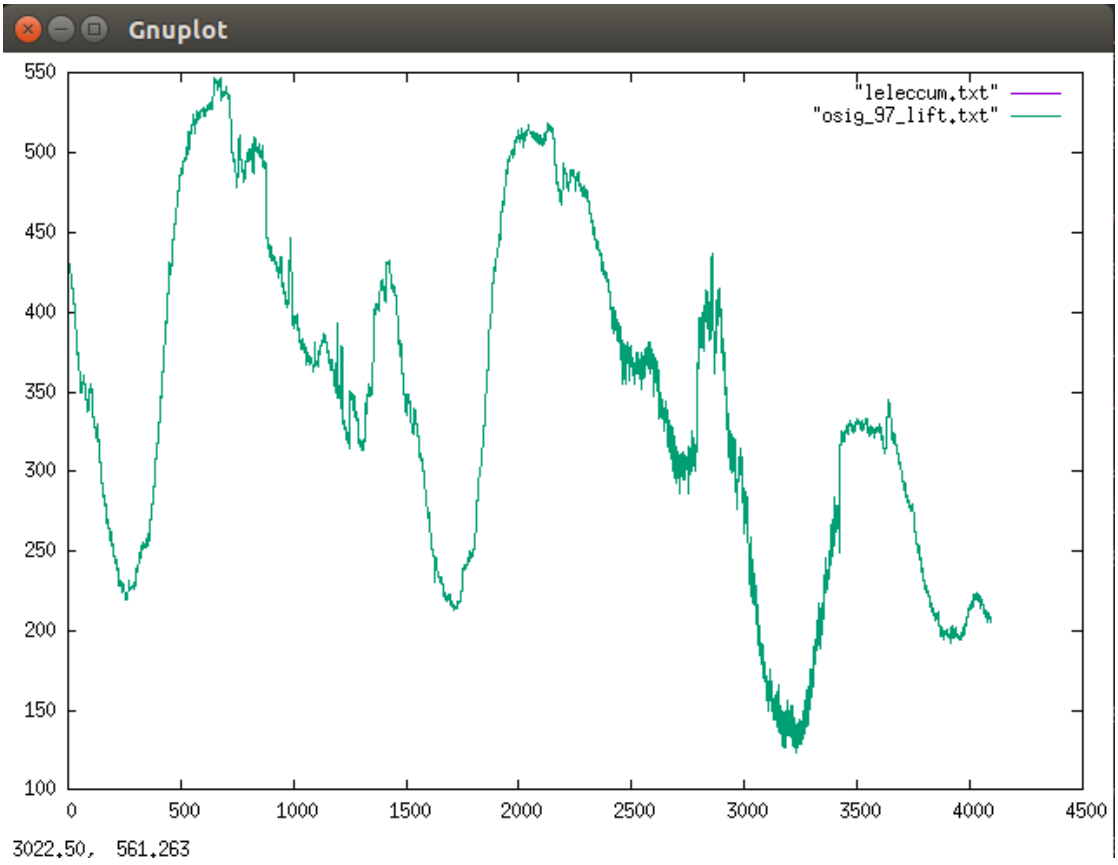

FIGURE 2.4 – La courbe initiale et reconstruite se recouvrent, indiquant une reconstruction quasi-parfaite avec le lifting biorthognaux 9/7

2.5 COMPARAISON

Comparons les 3 techniques sur le signal "test" issus d'une ligne d'image.

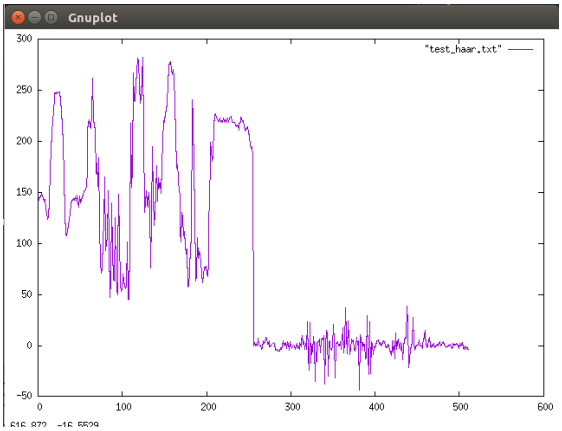

FIGURE 2.5 – Coefficients issus du banc de Haar

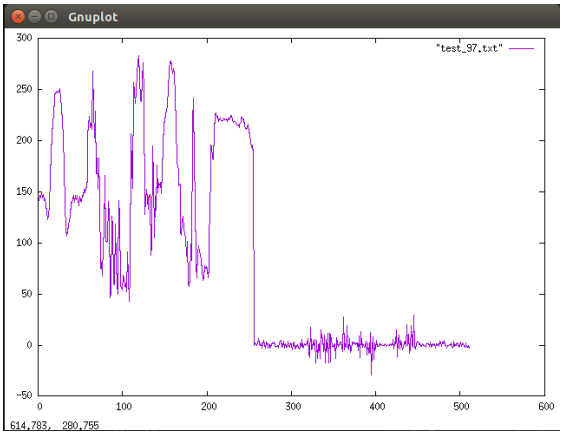

FIGURE 2.6 – Coefficients issus du banc 9/7

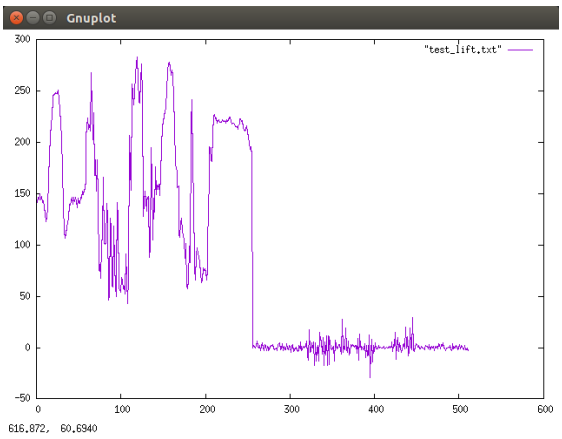

FIGURE 2.7 – Coefficients issus du lifting 9/7

FIGURE 2.8 – On peut remarquer une grande similarité entre les trois figures, dénotant la proximité des résultats des différentes techniques présentées

Je pense que le lifting a plusieurs interets. Au niveau mémoire, toute les transformations sont faite in-place, on a donc un coût constant. De plus, on économise du temps de calcul puisque l'on sous échantillone avant de filtrer (contrairement aux banc de filtre). Après mesurement, en moyenne le lifting est 10 micro secondes plus rapide sur un signal aussi petit que leleccum (4096 samples).

3 ONDELETTE

3.1 ANALYSE MULTI RÉSOLUTION

```
void amr (double* x, int p, int niveau )
{
    for ( int i = 1;
          i <= niveau;
          ++i )
    {
        analyse_97_lifting(x, p/pow(2,i-1));
    }
}

void iamr (double* x, int p, int niveau )
{
    for ( int i = niveau;
          i > 0;
          --i )
    {
        synthese_97_lifting(x, p/pow(2,i-1));
    }
}
```

Si un signal est de taille p, alors la décomposition maximale est égale à

$$log_2(p)$$

. Observons l'AMR de différents niveau sur le signal test.

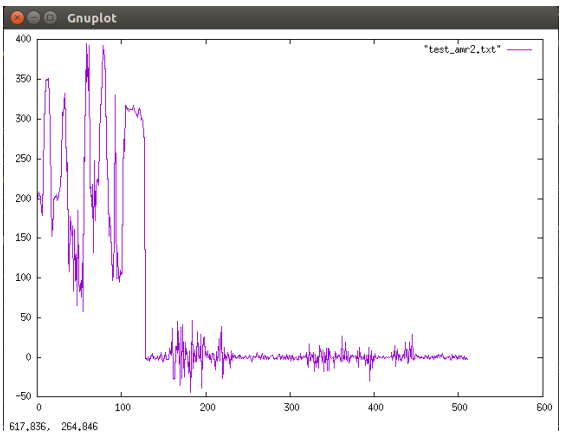

FIGURE 3.1 – AMR de niveau 2 sur le signal test

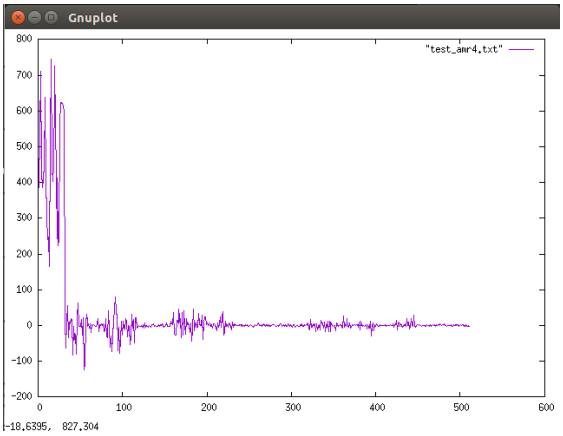

FIGURE 3.2 – AMR de niveau 4 sur le signal test

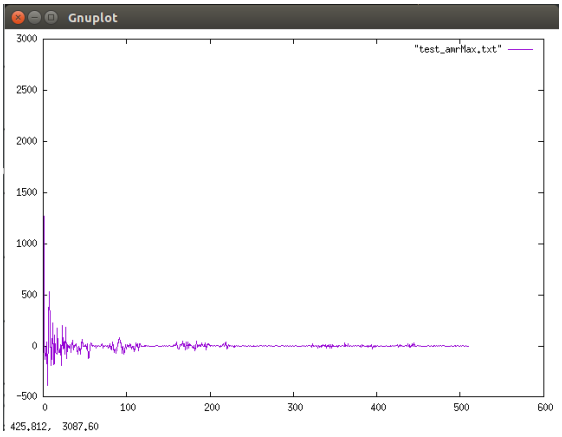

FIGURE 3.3 – AMR de niveau max sur le signal test

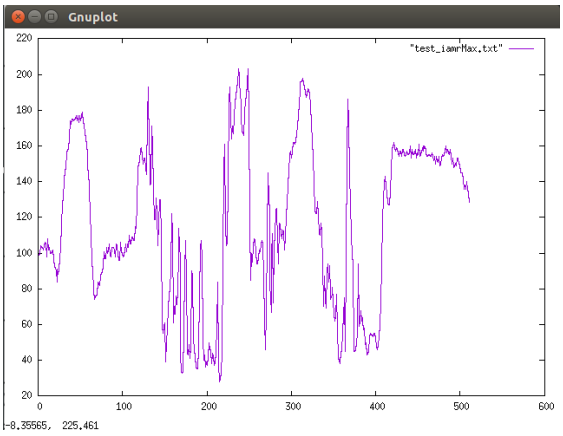

FIGURE 3.4 – Reconstruction à partir de l'AMR maximale. La reconstruction est parfaite (erreur : 3.23092e-24)

FIGURE 3.5 – A chaque niveau d'AMR, seulement les coefficients de détails (les plus petits) sont gardés, les coefficients d'approximation étant à nouveau analysés (la moitié devenant donc des coefficients de détails).

Pour vérifier ce résultat, nous pouvons calculer la valeur moyenne, minimale et maximale de chaque bande. Par exemple, pour une AMR de niveau 4 nous obtenons :

Moyenne (arrondi)	465	-9	-5	1	0
Minimum (arrondi)	165	-124	-78	-44	-29
Maximum (arrondi)	746	62	80	47	29

On peut bien observer l'intêret de l'AMR : de valeur qui de base ne peuvent pas être représenté par un char (signed short [-126, 127]) on obtient des valeur qui peuvent l'être. On peut donc abaisser le nombre de bits nécessaires à la représentation de certaines valeurs du signal,

ce qui est très intéressant pour la compression.

3.2 PASSAGE EN 2D

Tout a été fait jusqu'à maintenant en 1D, mais il est très simple de passer en 2D.

```
void analyse2D_97( double* m, int p, int size)
{
    for ( int i = 0 ;
          i < p;
          ++i)
    {
        analyse_97_lifting( m +(i*size), p);
    }

    for ( int i = 0 ;
          i < p;
          ++i)
    {
        double * column = new double[p];
        for ( int j = 0;
              j < p;
              ++j)
        {
            column[j] = m[ i + (j*size)];
        }
        analyse_97_lifting( column, p);
        for ( int j = 0;
              j < p;
              ++j)
        {
            m[ i + (j*size)] = column[j];
        }
    }
}

void synthese2D_97( double* m, int p, int size )
{
    for ( int i = 0 ;
          i < p;
          ++i)
    {
        double * column = new double[p];
        for ( int j = 0;
              j < p;
              ++j)
        {
            column[j] = m[ i + (j*size)];
        }
        synthese_97_lifting( column, p);
        for ( int j = 0;
              j < p;
              ++j)
        {
```

```
            m[ i + (j*size)] = column[j];
        }
    }
    for ( int i = 0 ;
          i < p;
          ++i)
    {
        synthese_97_lifting( m+(i*size) , p );
    }
}
```

Ce qui nous donne la possibilité de faire une AMR2D aussi, qui nous donne sur une image en niveau de gris :

FIGURE 3.6 – AMR2D de niveau 3 sur Lena, les coefficients ont été corrigés pour améliorer la visualisation

Après reconstruction on trouve une erreur de 1.85617e-21, ce qui est plus qu'avec l'AMR 1D, mais reste largement raisonnable, surtout une fois ramené à l'oeil humain qui ne vois pas la différence.
Regardons du coté des bandes en analysant leurs moyenne et variance. Les valeurs sont données dans l'ordre avec l'approximation en dernier.

Moyenne (arrondi)	0.008	0.008	0.04	0.07	-0.01	0.01	-0.32	-0.0008	-0.95	988
Variance (arrondi)	8.67	17.2	42.9	115.9	170.1	443.6	1064.7	1092.3	3484.4	127978

Même si ces valeur ne me paraissent pas tout à fait correct, on peut quand même observer que la moyenne augmente fortement dans les coefficients d'approximation et les derniers coefficients de détails. En fait, ce n'est peut être pas des valeur si aberrantes. Dans les coefficients d'approximation, ma valeur maximum est environ 1700 et ma moyenne 990, rien que cette valeur fait un écart de 700 au carré, ce qui donne vite un chiffre énorme.

3.3 Allocation de débit

Différents débits par bande en fonction du débit global demandé. Les résultats sont donnés dans l'ordre avec l'approximation en dernier.

0.25bit/px	3.3	4.3	5.6	7.1	7.6	9	10.3	10.3	12	17.2
0.5bit/px	3.6	4.6	6	7.3	8	9.2	10.5	10.6	12.2	17.5
1bit/px	4.1	5.1	6.4	7.8	8.4	9.7	11	11	12.7	18
2bit/px	5.1	6.1	7.4	8.8	9.4	10.7	12.05	12.09	13.7	19

Observons la relation entre le débit global demandé et le PSNR entre l'image originale et l'image reconstruite.

Bitrate	PSNR
0.5bit/px	53.6
2bit/px	61
4bit/px	74

On observe bien une diminution du PSNR quand on baisse le bitrate, cependant il reste très supérieur à 40, ce qui indique une très bonne reconstruction. La visualisation des image reconstruite ne permets pas à l'oeil humain de voir de différence notable.

Avec gzip on obtient un facteur de compression de 1,17 sur "lena.bmp". J'ai essayé d'utiliser gzip pour encoder les symboles donnés par la fonction quant1m_idx, mais je dois l'avoir mal utilisé : quelque soit mon débit global j'obtiens un fichier de 65kb (taille de l'image divisé par 4) et quand je le compresse avec gzip j'obtiens un fichier de 298 bytes. Je sais que le jpeg2000 compresse très bien avec des facteurs de compression de 20 et plus sans dégrader de trop l'image, mais là ça ferait un facteur de compression qui se rapproche des 1000, ce qui me parais totalement disproportioné.

4 Conclusion

Pour conclure, la compression jpeg2000 est extrèmement puissante. Même si je n'ai pas réussi à mettre en place entièrement la pipeline, rien que la décomposition par lifting et l'analyse multi-résolution sont des outils assez impressionnants. Je suis d'ailleurs très surpris de ne pas voir plus d'utilisation du format jpeg2000, qui, comparé au jpeg, est bien plus intéressant.