UNIVERSITAT DE BARCELONA

Tecnologies Multimèdia

Projecte final: Còdec

Vicent Roig / Igor Dzinka

08/01/2015

Índex

1. Introducció

A aquest projecte es posaran en pràctica els coneixements obtinguts sobre codificació de video i compressió a l'assignatura de Tecnologies Multimèdia. En particular ens centrarem en implementar un còdec propi que ens permeti obtenir una millor compressió respecte el DCT o compressió interquadre obtingut mitjançant el JPEG.

A mode introductori tractarem els conceptes inicials exposats a les classes magistrals i que son clau per el desenvolupament d'aquest projecte:

La compressió inter-quadre: Compensació de moviment

La compressió inter-quadre intenta detectar i eliminar la redundància temporal entre quadres (frames) successius. Aquesta compressió es realitza mitjançant un procés de codificació diferencial, i un altre anomenat "compensació de moviment". Amb aquestes tècniques s'aconsegueixen els nivells més elevats de compressió.

Técnica de compensació de moviment

La tècnica de compensació de moviment treballa sobre petites àrees de la imatge. En les imatges en moviment, és habitual que algunes zones de la imatge es desplacin en un fotograma pel que fa a l'anterior. El sistema de compensació del moviment, tracta de buscar el nou emplaçament d'aquestes regions, i calcular els vectors de desplaçament codificant solament aquests vectors.

Imatges "I" (intra-quadre)

Les imatges I són imatges que utilitzen només compressió intra-quadre. Cada quadre I és comprimit i processat de forma independent als altres, i conté per si sol tota la informació necessària per a la seva reconstrucció.
Les imatges I són les que més informació contenen, i per tant les que menys compressió aporten. Sempre inicien una seqüència i serveixen de referència a les imatges P següents.

Imatges "P" (predicció)

Les imatges P es generen a partir de la imatge I o P anterior més propera. El codificador compara la imatge actual amb l'anterior I o P, i codifica únicament els vectors de moviment i l'error de predicció. S'utilitza en aquest cas una predicció cap a endavant. Aquestes aporten un grau important de compressió.

GOP (Group of Pictures)

És un grup d'imatges successives dins d'un stream codificat de vídeo. Cada stream consisteix en successius GOPs, i és des de les imatges que contenen aquests GOPs des d'on es generen les imatges visibles.

Un GOP pot contenir els diferents tipus d'imatges (tipus I, tipus P o tipus B).Un GOP sempre començarà amb una imatge tipus I, i a continuació seguiran diverses imatges tipus P.

2. Objectiu

El principal objectiu de la pràctica és implementar un codificador i descodificador utilitzant tècniques de compensació de moviment i compressió intra-quadre per tal de millorar la compressió intra-quadre que obtenim mitjançant el JPEG.

El còdec ha de permetre reduir la mida total del vídeo emmagatzemat minimitzant la pèrdua de qualitat. En aquest sentit, els paràmetres que típicament han de ser modificables per tal d'optimitzar el procés han de ser la **mida de les tessel·les** (o el nombre d'aquestes), el **desplaçament màxim de les tessel·les**, un **factor de qualitat** del vídeo obtingut i la **distància entre dos frames de referència** (GOP).

Constarà de dues parts: **codificació** i **descodificació**.

Codificació

Pot ser un procés de pocs minuts. El resultat final seran dos fitxers: un zip d'imatges en jpg i un gzip amb la informació de les tessel·les. La mida total dels dos ha de ser menor que guardar directament les imatges en jpeg en un zip. La mecànica bàsica serà la seguirà el següent patró:

1. Agafar un frame de referencia i **segmentar-lo en tessel·les**.
2. Mitjançant un algorisme de correlació, comparar les tessel·les anteriors amb les subimatges properes (depenent del **desplaçament** elegit) dels següents frames.
3. En cas que el **factor de qualitat** ens indiqui que la tessel·la i la subimatge són equivalent, eliminar la subimatge del frame i guardar a d'on hem eliminat la tessel·la. *Cal tenir en compte que la DCT és més òptima amb imatges suaus sense canvis bruscs.*

Descodificació

A diferencia de la codificació, ha de ser un procés ràpid, al final del qual hem de recuperar una seqüència d'imatges amb una qualitat el més propera a la original possible.

El decoder haurà de llegir les imatges sense tessel·les i el fitxer de tessel·les i a partir d'aquesta informació, tornar a copiar les tessel·les on abans s'han eliminat per tal de recuperar la imatge original.

1. Escollir un **algoritme de correlació** per tal d'establir la **qualitat de la compressió**.
2. Suavitzar la imatge resultant abans de guardar-la a JPEG.
3. **Optimitzar els paràmetres** fins a trobar una relació decent entre qualitat i compressió per les imatges del cub, per tal d'obtenir **una millora d'almenys al voltant del 5%** respecte a la compressió inter-quadre (JPEG).

3. Descripció detallada de la implementació

Diàleg per a personalitzar paràmetres de compressió

Per tal de poder modificar de manera fàcil els paràmetres que es passen al algoritme de la compressió hem creat un diàleg a la interfície gràfica per tal de ajustar els valors de manera convenient. Els paràmetres que es poden modificar son:

- GOP.
- Offset: La compensació de moviment en el vídeo.
- La mida de les tessel·les.
- El factor correlació.

Funcionament Codificació

El nostre algoritme de codificació funciona de la següent forma:

Estructura de Builder Skeleton.

Per tal d'escriure la informació mínima i imprescindible al fitxer GZIP, hem estructurat les dades de la següent forma, optimitzant al màxim el nombre de bytes que desarem al fitxer binari :

BUILDER SKELETON STRUCTURE						
patchsize	**GOP**	**rows**	**P-frame**	**matches**	**(x, y)**	**I-frame N**
1byte	1byte	1byte	1byte	2bytes	4bytes	1byte

Com podem comprovar el fitxer tindrà una capçalera / header de mida fixa. On s'especifica la mida de la tessel·la (patchsize) i el GOP, així com el nombre de files total de tessel·les al I-frame, per tal de minimitzar l'espai a desar a aquesta estructura (és més eficient desar directament el nombre de files en lloc del width de la imatge donat que podem arribar a optimitzar 1 byte de representació numèrica).

A partir d'aquí trobar l'índex del P-frame a tractar, és important tenir en compte que no cal tenir l'índex de a quin I-frame fa referència ja que en funció del P-frame i el GOP podem deduir-ho directament amb una simple operació (divisió entera de l'índex del I-frame entre el nombre GOP). A partir d'aquí desarem el nombre de matches (hits o tessel·les a substituir a aquest P-frame) per tal de poder iterar sobre l'estructura i tenir coneixement de quan haurem d'aturar la seva reconstrucció.

A partir d'aquest punt la informació desada que segueix es més trivial, simplement cal ajustar l'espai que ocuparem (bytes). Per cadascun dels matches, seguidament comptarem amb les dues coordenades (x, y) de la tessel·la a substituir al P-frame (2 bytes per coordenada, que posteriorment parsejat passarà a emmagatzemar-se a un tipus short) i a més, l'índex de la tessel·la corresponent del I-frame (tenint en comte que les tessel·les aniran numerades d'esquerra a dreta / amunt - baix)

Correlació

La funció de correlació que hem seleccionat per a nostre còdec es la de la distancia Euclidiana On els valors de color de cada canal son equivalents a les coordenades espacials.

$$d_E(P,Q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_n - q_n)^2} = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2}.$$

Per calcular la correlació d'una tessel·la amb una part de la imatge, realitzem la suma de les distancies euclidianes de tots els píxels (píxel 1 de la tessel·la amb píxel 1 del P-frame, píxel 2 de la tessel·la amb píxel 2 del P-frame...) i trobem el valor mig. Aquest valor es el valor de correlació entre una tessel·la i un àrea del pframe.

Matching

El factor de qualitat de compressió (q) servirà per discriminar aquelles tessel·les que s'han d'eliminar del P-frame d'aquelles que s'han de mantenir. Cada una de les tessel·les que eliminem seran substituïdes per un valor promig dels valors que conformaven aquella regió, d'aquesta forma la DCT permetrà al JPEG comprimir millor, donat que minimitzarem els canvis bruscs. Cal a dir que a més d'això, al acabar el tractament d'un P.-frame (es a dir, un cop substituïdes totes les tessel·les que han passat el test de correlació pel seu corresponent color promig) s'aplicarà un filtratge de mitjana per tal de reduir encara més les freqüències altres introduïdes al emplenar tota una tessel·la amb un sol color.

El procés de filtratge es realitzarà per totes les imatges que composen el video, un cop hagi finalitzat el procés de matching i la substitució de tessel·les. Posteriorment es comprimirà el **video en format zip**, i el fitxer de construcció **(buider) en format gzip.**

Artefacte generat

En el nostre cas concret, hem decidit empaquetar el **video.zip** i el **builder.gz** a un fitxer contenidor **.tar**, el qual no ens proporciona compressió, però ens ofereix la possibilitat de comptar amb un sol artefacte i manipular-ho més fàcilment, de cara a obrir-ho de nou i assegurar millor integritat al no perdre el builder, d'aquesta manera evitarem errors.

Un altre alternativa seria desar-ho en un format amb extensió pròpia, o inclús desar-ho en format zip (el qual a més d'empaquetar-ho sí que ens proporcionaria de nou un poc més de compressió) i canviar l'extensió.

Realitzant diverses proves hem decidit desar-ho amb **.tar** perquè un cop així podríem passar-hi una capa de compressió amb un compressor de millor qualitat com ara el propi **gzip**, obtenint així fins a 10Kb més de compressió en un fitxer **.tar.gz o .tgz**, o sacrificar un poc més de temps de compressió per tal d'obtenir una compressió encara superior mitjançant **bzip2** generant així un **tar.bz2** o **tbz2**.

De tota manera, no hem introduït aquesta segona capa de compressió donat que l'objectiu del projecte es obtenir compressió mitjançant l'algorisme inter-quadre desenvolupat gràcies a la compressió pròpia de la DCT i pas encadenar algoritmes de compressió mentre aquests encara ofereixin aquesta capacitat.

Funcionament Descodificació

El procés de descodificació es prou més senzill donat que bàsicament anirà llegint la informació del builder i en funció de les tessel·les a reconstruir les substituirà per la tessel·la procedent del I-frame indicat.

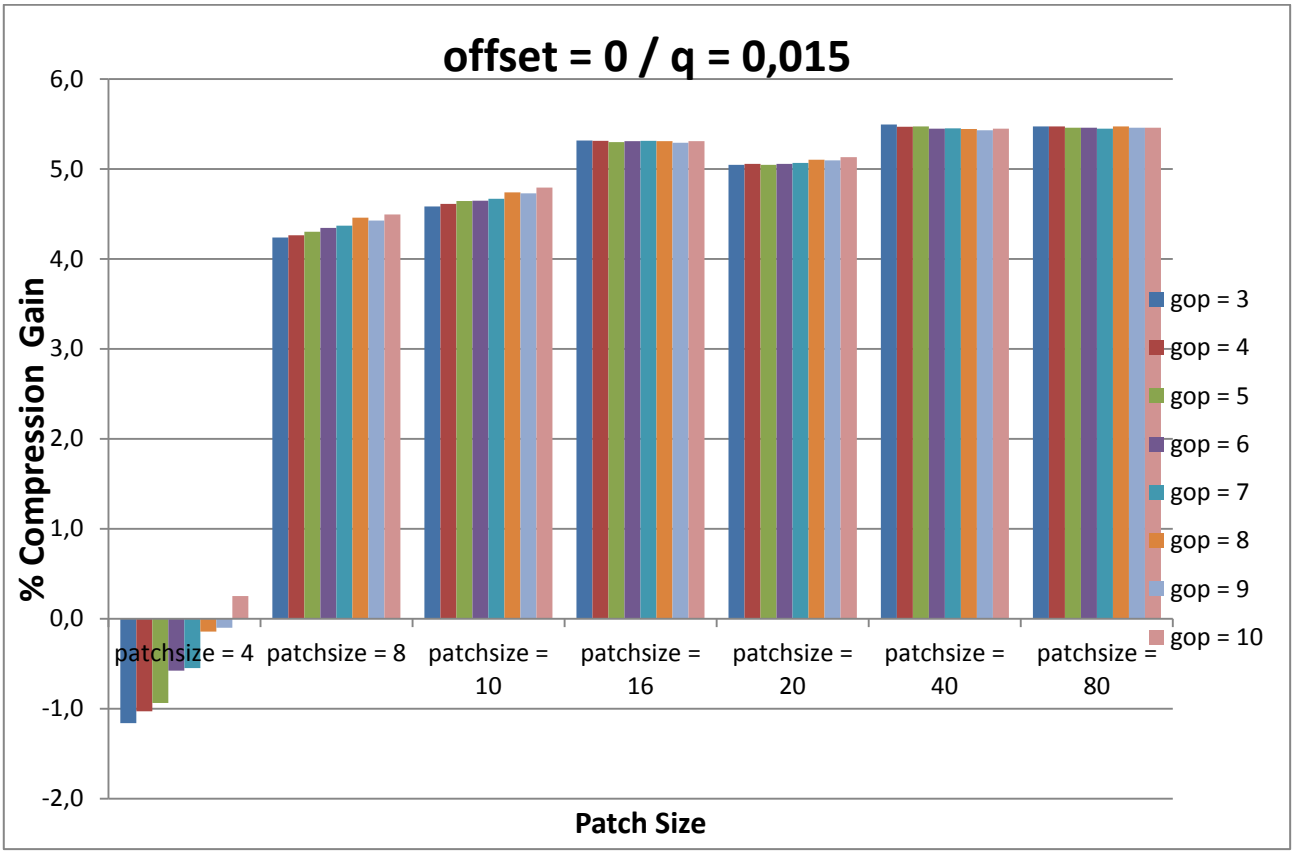
En aquest cas com comptem amb un fitxer .tar per tant, dintre del procés de descodificació **encode()** es desempaquetarà temporalment el builder i ràpidament es descomprimirà i carregarà en memòria per tal de ser seguidament eliminat, el mateix farem amb el video.zip, un cop descomprimit i carregat l'esborrarem. Aquest procés es molt ràpid i no afecta al compressed.tar original, observem que en cap moment tindrem alhora extrets ambdós fitxers sino que és un procés seqüencial i aquests fitxers passen a ser temporals volàtils.

Un cop obtenim l'array de bytes carregat al builder i el conjunt de frames del video a tractar. Es realitza la reconstrucció del video (recorregut del Builder tractant les tessel·les i substituint-les) dintre de la funció **build().** Aquest procés esdevé molt ràpidament.

4.Resultats de funcionament i discussió

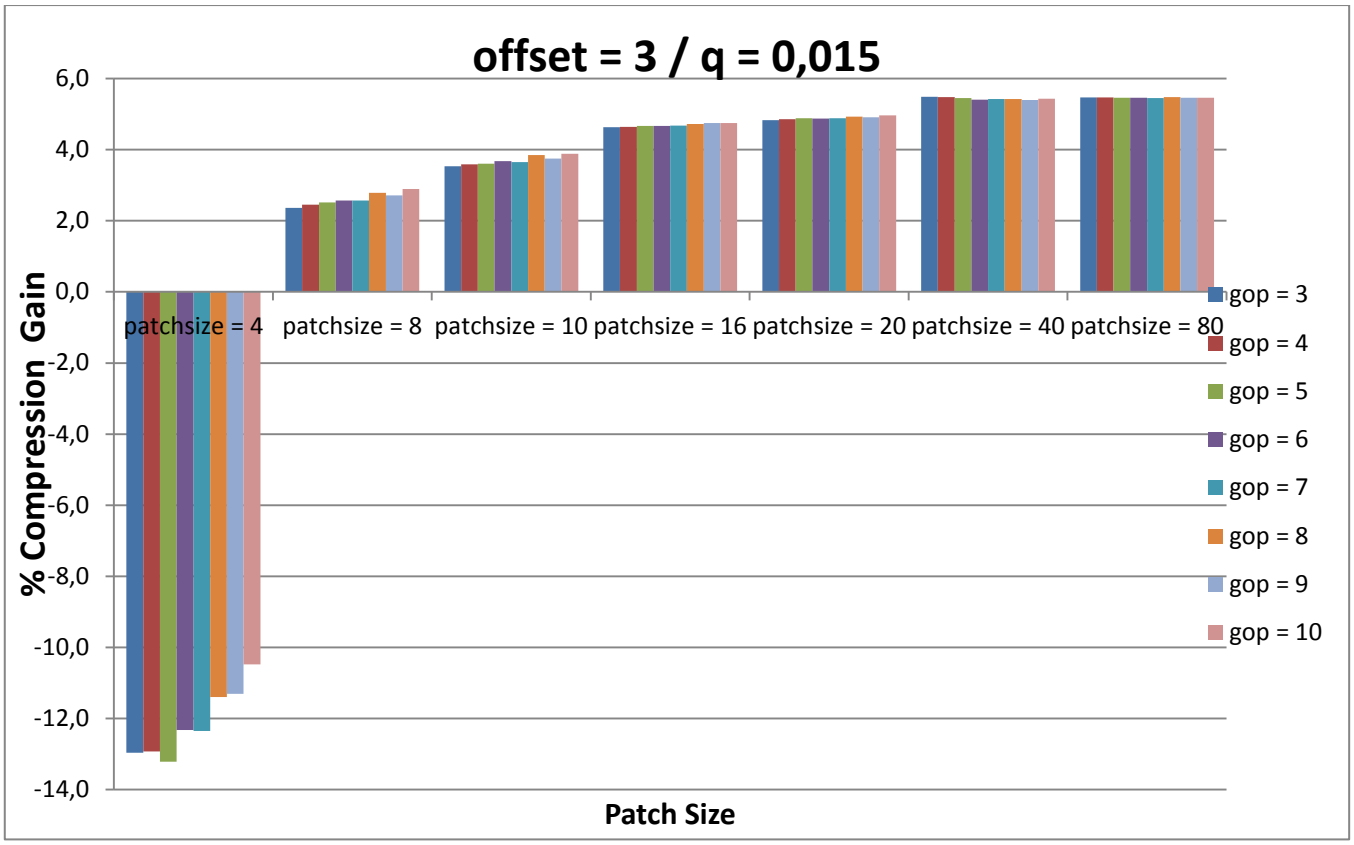
Per veure l'anàlisi exhaustiu del rendiment del nostre còdec, consulteu **MediaPlayer_stats.xlsx**

A continuació mostrarem un conjunt de dades amb informació rellevant sobre els resultats d'execució per la compressió del fitxer "imagenes.zip" amb diferents paràmetres.



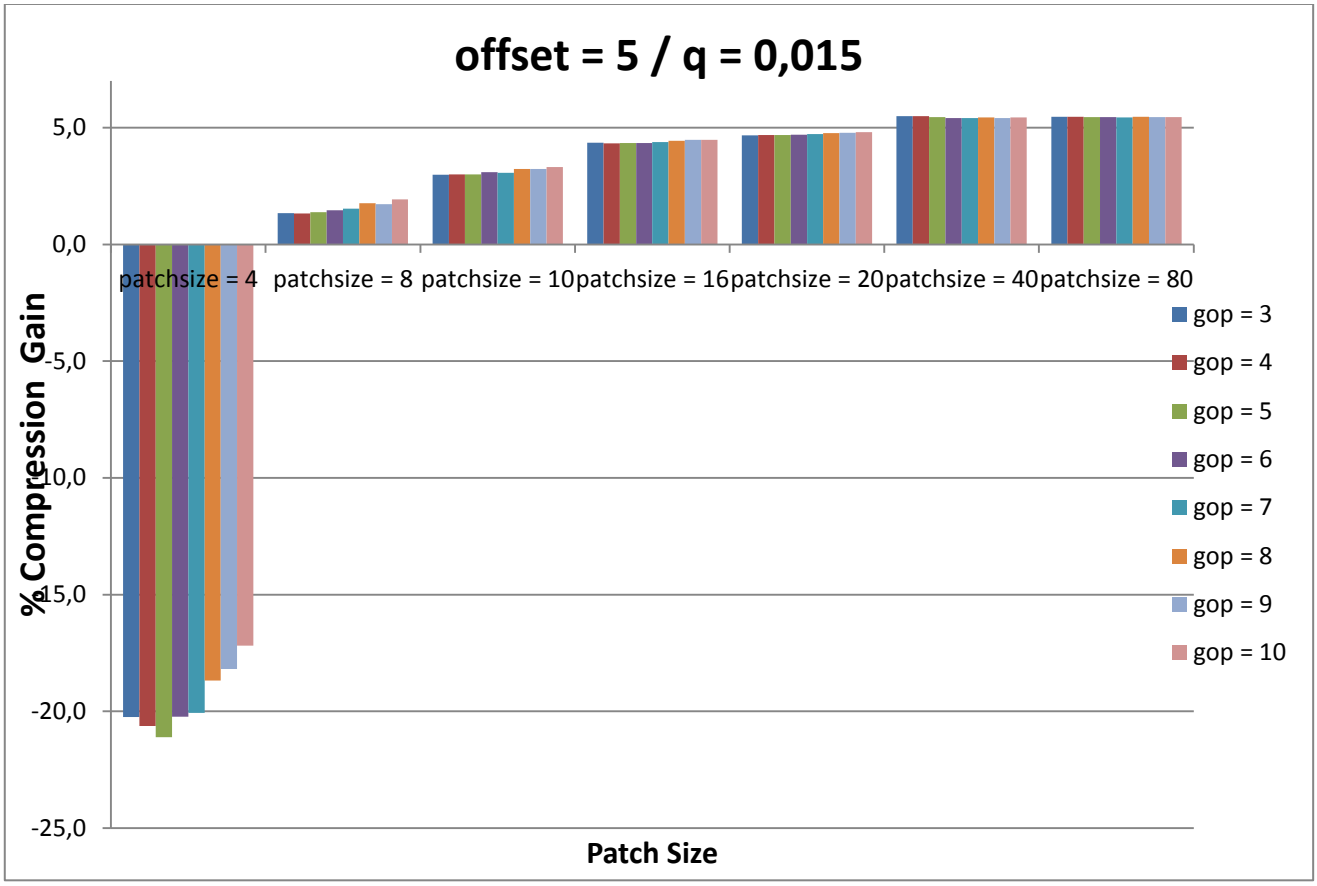
offset =0

gop	patchsize						
	4	8	10	16	20	40	80
3	1049,047	993,068	989,494	981,882	984,675	980,019	980,259
4	1047,666	992,814	989,188	981,902	984,558	980,308	980,254
5	1046,726	992,401	988,846	982,055	984,678	980,251	980,405
6	1042,996	991,938	988,829	981,949	984,555	980,494	980,406
7	1042,679	991,689	988,581	981,931	984,447	980,478	980,496
8	1038,482	990,785	987,852	981,968	984,074	980,565	980,246
9	1038,051	991,089	987,955	982,134	984,181	980,708	980,412
10	1034,405	990,414	987,319	981,933	983,796	980,504	980,417



offset =3

gop	patchsize						
	4	8	10	16	20	40	80
3	1171,447	1012,462	1000,396	988,942	986,901	980,100	980,259
4	1171,010	1011,575	999,764	988,919	986,622	980,230	980,254
5	1174,039	1010,917	999,585	988,647	986,338	980,445	980,405
6	1164,764	1010,316	998,887	988,608	986,485	980,930	980,406
7	1165,027	1010,364	999,165	988,468	986,392	980,798	980,496
8	1155,204	1008,101	997,136	988,069	985,869	980,758	980,246
9	1154,202	1008,865	998,165	987,745	986,053	980,998	980,412
10	1145,626	1007,003	996,738	987,731	985,512	980,696	980,417



offset =5

gop	patchsize						
	4	8	10	16	20	40	80
3	1246,990	1023,103	1006,027	991,726	988,517	979,965	980,259
4	1250,879	1023,287	1005,853	992,105	988,436	979,967	980,254
5	1255,936	1022,721	1005,803	991,876	988,368	980,408	980,405
6	1246,859	1021,738	1004,930	991,895	988,257	980,791	980,406
7	1245,018	1021,069	1005,174	991,574	988,019	980,793	980,496
8	1230,669	1018,723	1003,411	990,950	987,551	980,600	980,246
9	1225,663	1019,093	1003,449	990,558	987,393	980,904	980,412
10	1215,271	1017,021	1002,534	990,507	987,073	980,616	980,417

5. Conclusions

Després de realitzar unes proves exhaustives amb el còdec, veiem que amb la compressió podem arribar a obtenir una compressió de entre 5% i 6%. Mantenint un resultat acceptable, a partir d'aquí obtindrem problemes en termes de fluència, tessel·les massa grans, etc. Nosaltres no ho hem considerat acceptable i em limitat el factor de qualitat i ajustat els màxims dels paràmetres per tal de ajustar valors que permetin una compressió viable.

Un factor important a tenir en compte es que amb la mida de tessel·la massa petita en general no obtindrem compressió. Al fer *templateMatching* l'algorisme substitueix la zona trobada per un promig de color d'aquella zona. Quan la zona substituïda es prou gran, traiem gran part de la informació i aconseguim que la compressió DCT sigui millor. Però si la zona substituïda es massa petita, aquesta substitució acaba introduint mes informació de la que s'ha tret (els *edges* de la zona substituïda aporten molta informació a la imatge i fa que la compressió sigui pitjor).

Per altra banda una mida de tessel·la massa gran cada cop es mes difícil que el *templateMatching* trobi coincidències als *P-Frames*, ja que en un àrea del vídeo gran es mes fàcil que hi hagi variàncies

Observem una tendència natural d'augmentar la compressió en funció de la mida de la tessel·la, això però comença a estabilitzar-se al arribar a un 5% de compressió.

Al incrementar el GOP aconseguim una lleugera millora a la compressió. Això es deu a que com mes gran es el GOP, menys quadres de referencia (I-Frames) tenim i mes quadres son comprimits(P-Frames). En quant a la qualitat de vídeo, com mes gran sigui GOP pitjors resultats obtenim. Es deu a que amb un GOP gran per trobar coincidències hem de mirar a un frame molt llunyà i si al video un objecte va canviant de posició, amb aquest problema pot semblar que s'ha congelat. A mes a mes podem tenir problemes de fluència.

6. Referències

- Material docent facilitat a classe.
- Escuela de Ingenieros Industriales (UCLM) http://edii.uclm.es
- Wikipedia The Free Encyclopedia http://en.wikipedia.org
- BatchFrame http://www.batchframe.com/

7. Codi font

Encoder.java

```
package codec;

/**
 *
 * @author Vicent Roig & Igor Dzinka
 */
public class Encoder {
    public final static String BUILDER_FNAME = "builder_skeleton.gz";
    public final static String VIDEO_FNAME = "video.zip";
    public final static String COMPRESSED_FNAME = "compressed.tar";

    private ArrayList <Byte> builder;
    private ArrayList<BufferedImage> raw;
    private final int cols;
    private final int rows;
    private final float quality;
    private final short gop;
    private final short brick;    // breaking block size
    private final short offset;   // Defines Block neighbourhood to seek/match

    public Encoder(ArrayList video, float quality, short gop, short block_size, short offset){
        this.raw = video;
        this.quality = quality;
        this.gop = gop;
        this.brick = block_size;
        this.rows = raw.get(0).getWidth()/brick;
        this.cols = raw.get(0).getHeight()/brick;
        this.offset = offset;
    }


    /**
     * Function that recieves a Buffered image and chunks it into a List of subimages of
     * each one of the same size and containing a different region of the image
     * @param frame Input Buffered Image
     * @return List of Buffered Images that are chunks of the input image
     */
    private BufferedImage[] chunkFrame(BufferedImage frame){
        int chunks = rows * cols;
        int count = 0;

        BufferedImage imgs[] = new BufferedImage[chunks]; //Image array to hold image chunks
        for (int y = 0; y < frame.getHeight(); y+=this.brick) {
            for (int x = 0; x < frame.getWidth(); x+=this.brick) {
                imgs[count] = frame.getSubimage(x, y, this.brick, this.brick);
                //System.out.println("Brick "+count+": Coordenades: "+x+" "+y);
                count++;

            }
        }
        return imgs;
    }


    /**
     * Determines how different two identically sized regions are.
     * @param im1
     * @param im2
     * @return
     */
    private double compareImages(BufferedImage im1, BufferedImage im2){
        assert(im1.getHeight() == im2.getHeight() && im1.getWidth() == im2.getWidth());
        double variation = 0.0;
        for(int y = 0;y < im1.getHeight();y++){
            for(int x = 0;x < im1.getWidth();x++){
```

```
                variation += compareARGB(im1.getRGB(x,y),im2.getRGB(x,y))/Math.sqrt(3);
            }
        }
        return variation/(im1.getWidth()*im1.getHeight());
    }


    /**
     * Calculates the difference between two ARGB colors (BufferedImage.TYPE INT ARGB).
     * @param rgb1 RGB value to compare
     * @param rgb2 RGB value to compare
     * @return The difference between two colors
     */
    private double compareARGB(int rgb1, int rgb2){
        double r1 = ((rgb1 >> 16) & 0xFF)/255.0;
        double r2 = ((rgb2 >> 16) & 0xFF)/255.0;
        double g1 = ((rgb1 >> 8) & 0xFF)/255.0;
        double g2 = ((rgb2 >> 8) & 0xFF)/255.0;
        double b1 = (rgb1 & 0xFF)/255.0;
        double b2 = (rgb2 & 0xFF)/255.0;
        double a1 = ((rgb1 >> 24) & 0xFF)/255.0;
        double a2 = ((rgb2 >> 24) & 0xFF)/255.0;

        // if there is transparency, the alpha values will make difference smaller
        return a1*a2*Math.sqrt((r1-r2)*(r1-r2) + (g1-g2)*(g1-g2) + (b1-b2)*(b1-b2));
    }

    /**
     * Method that Serializes a short type value and adds it to builder byte array
     * @param n Value to serialize
     */
    private void serializeShort(short n){
        this.builder.add((byte)(n & 0xff));
        this.builder.add((byte)((n >> 8) & 0xff ));
    }

    /**y
     * Method that recieves a coordinade(x,y), serializes it and puts it into the builder
     * @param x X coordinate
     * @param y Y coordinate
     */
    private void serializeCoord(short x, short y){
        serializeShort(x);
        serializeShort(y);
    }


    /**
     * Encoding function.
     *
     * BUILDER SKELETON STRUCT. (each X represents 1 byte)
     *    X   /   X   /   X   |    X    /    XX    /     XX XX           XX      [...] |    X    (etc)
     *< brick    gop    rows  |  pframe   matches    pf match (x,y)   iF N-chunk  [...] |  pframe (etc)>
     */
    public void encode(){
        short matches;
        ArrayList <int[]> coords;
        BufferedImage[] iframe = null;


        this.builder = new ArrayList();
        // save builder skeleton header:
        this.builder.add( (byte) (this.brick & 0xff) );
        this.builder.add( (byte) (this.gop & 0xff) );
        this.builder.add( (byte) (this.rows & 0xff) );

        System.out.println("@encode STARTING (" + raw.size()+ " frames)...");
        //loop over all video frames
        for (int k = 0; k < raw.size(); k++) {
            matches = 0;
```

```
            BufferedImage pframe = raw.get(k);

            // We only take as a reference frame the first picture from every GOP
            if( k % this.gop == 0 ){
                // Imágenes codificadas intracuadro (origen)
                iframe = chunkFrame(pframe);
                System.out.println("-> New _IFRAME " + k + " splitted into " + iframe.length + " chunks" );
            } else {
                // Imágenes codificadas intercuadro (destino)
                coords = new ArrayList();
                this.builder.add( (byte) (k & 0xff) );
                System.out.print("\t_PFRAME "+ k + " ..." );

                this.builder.add((byte) 0);
                this.builder.add((byte) 0);
                int m_index = this.builder.size() - 2;
                /* we get the m_index as a pointer as it reserves the needed bytes positions in the array,
                Once we have computed matches, we can update it directly */

                //template-matching for each block of IFRAME into PFRAME
                for (int h=0; h < iframe.length; h++) {
                    //System.out.println("\t\t@debug tenplate matching iteration "+ h );
                    if( templateMatching(h, iframe[h], pframe, coords) ) matches++;
                }

                System.out.println("\t\t-> total matches ["+ matches +"]");
                if(matches > 0){
                    //Let's update matches field with a valid value
                    this.builder.set(m_index, (byte)(matches & 0xff));
                    this.builder.set(m_index+1, (byte)((matches >> 8) & 0xff ));

                    setPatchColor(pframe, coords );
                }
            }
        }
        System.out.println("@encode FINISHED (" + raw.size()+ "frames)");

        //builder GZIP compression:
        if(!compressGzipFile( toByteArray(this.builder) )){
            System.out.println("GZIP compression failed!");
            return;
        }
        System.out.println("[Builder GZIP compression succeded]");

        //average filtering:
        System.out.println("Aplying average filter...");
        this.raw = new Convolution().applyFilterToVideo(Convolution.kernel.AVERAGE, this.raw);

        // Files to tar
        File[] files = new File[2];
        files[0] = new File(VIDEO_FNAME);
        files[1] = new File(BUILDER_FNAME);

        //images JPEG compression:
        saveZip(files[0]);

        //pack GZIP builder & compressed JPEG video togheter
        try {
            tar(files);
        } catch (FileNotFoundException ex) {
            System.out.println("A problem happened while generating tar");
            Logger.getLogger(Encoder.class.getName()).log(Level.SEVERE, null, ex);
        }

    }


    /**
     * Saves current BufferedImage list raw into Compressed ZIP file in JPEG format.
     * @param file File on which we want to save our content.
     */
```

```
public void saveZip(File file){
    ZipSaveWorker zp = new ZipSaveWorker(this.raw, file);
    zp.run();
}


/**
 * Method that calculates the mean color value of an Image
 * @param image Input Image
 * @return Mean color value of the input Image
 */
private double meanValue(BufferedImage image){
    Raster raster = image.getRaster();
    double sum = 0.0;

    for(int y = 0; y < image.getHeight(); ++y)
        for(int x = 0; x < image.getWidth(); ++x)
            sum += raster.getSample(x,y,0);

    return sum / (image.getWidth() * image.getHeight() );
}


/**
* Template matching Function.
* -------------------------------
* @param h           patch ID (ordered integer number) - index of the chunk, which is used to calculate
* the zone of the Pframe where template matching will be done.
* @param template    iframe template to match - template chunk extracted from a IFrame
* @param pframe      image to match on - A Pframe, Image where we will look for matches.
* @param coords      Array of coords used to set patchColor once full pframe matching has finished
* @return true if a match has been founded. False otherwise
*/
private boolean templateMatching(int h, BufferedImage template, BufferedImage pframe,
        ArrayList <int[]> coords){

    int xmin, xmax, ymin, ymax;
    ymin = (h%rows)*brick - offset;
    if( ymin < 0 ) ymin = 0;

    ymax = brick *((h%rows) + 1) + offset;
    if( ymax  > raw.get(0).getWidth()) ymax = raw.get(0).getWidth();

    xmin = (h/rows)*brick - offset;
    if( xmin < 0 ) xmin = 0;

    xmax = brick * ((h/rows)+1) + offset;
    if( xmax > raw.get(0).getHeight()) xmax = raw.get(0).getHeight();

    //System.out.println("Brick "+h+": Coordenades: of search"+xmin+" "+xmax+" "+ymin+" "+ymax);

    for(int i = ymin; i <= ymax-brick; i++){
        for(int j = xmin; j <= xmax-brick; j++){
            /* matching evaluation */

            double corr = compareImages( pframe.getSubimage(i, j, this.brick, this.brick), template);
            if(corr < this.quality){
                //System.out.println("\t\tINDEXED brick @("+i+","+j+") with corr:"+corr );
                int[] coord = { i, j };
                coords.add(coord);
                serializeCoord((short)i, (short)j);
                serializeShort((short)h);
                //System.out.println("\t\t(x, y) " + coord[0] + ", " + coord[1] + " / h " + h);
                return true;
            }
        }
    }
    return false;
}
```

```
    /**
     * Method that recieves a Image and an Array of coordinades That represent patches we want to recolor.
     * Sets a color of every patch to the mean value of pixels the patch contains.
     * @param pframe Input image
     * @param coords array of coordinades
     * @return
     */
    private void setPatchColor(BufferedImage pframe, ArrayList <int[]> coords ){
        int x, y;
        for (int[] coord : coords) {
            x = coord[0];
            y = coord[1];
            // Compute average patch color
            int[] colors = pframe.getRGB(x, y, this.brick, this.brick, null, 0, this.brick);
            int r = 0;
            int b = 0;
            int g = 0;
            for (int c : colors){
                r += ((c >> 16) & 0xFF);
                g += ((c >> 8) & 0xFF);
                b += (c & 0xFF);
            }
            int R = r / colors.length;
            int G = g / colors.length;
            int B = b / colors.length;
            //System.out.println("\tAVG PATCH RGB: " + R + " , " + G + " , " + B);

            // Set color patch
            int[] rgbArray = new int[(this.brick-2) * (this.brick-2)];
            Color c = new Color(R,G,B);
            //Color c = new Color(255,255,255);
            Arrays.fill(rgbArray, c.getRGB());
            pframe.setRGB(++x, ++y, this.brick-2, this.brick-2, rgbArray, 0, 0);
        }
    }


////////////////////////////////////////////////////
//              GZIP compression                  //
////////////////////////////////////////////////////


    /**
     * @param data Byte array to compress
     * @return true if compression is done correctly. false otherwise.
     */
    private boolean compressGzipFile( byte[] data ){
        try{
            ByteArrayOutputStream byteStream = new ByteArrayOutputStream(data.length);
            try{
                GZIPOutputStream gos;
                gos = new GZIPOutputStream(byteStream, false);
                try{
                    gos.write(data, 0, data.length);
                } finally { gos.close(); }
            } finally { byteStream.close(); }

            byte[] compressedData = byteStream.toByteArray();
            FileOutputStream fileStream = new FileOutputStream(BUILDER_FNAME);
            try{ fileStream.write(compressedData); } finally {
                try{ fileStream.close(); }catch(Exception e){
                    /* We should probably delete the file now? */
                    return false;
                }
            }
        } catch(IOException ex){
            return false;
        }
        return true;
    }
```

```
    /**
     * Method that casts an  Array List into List
     * @param in
     * @return
     */
    private byte[] toByteArray(ArrayList<Byte> in) {
        int n = in.size();
        byte ret[] = new byte[n];
        for (int i = 0; i < n; i++) ret[i] = in.get(i);
        return ret;
    }


    /**
     * Method that recieves a List of files  and puts them into a .tar file
     * @param filesToTar List of files
     * @throws FileNotFoundException
     */
    private void tar(File[] filesToTar) throws FileNotFoundException{
        // Output file stream
        FileOutputStream dest = new FileOutputStream( COMPRESSED_FNAME );

        // Create a TarOutputStream
        TarOutputStream out = new TarOutputStream( new BufferedOutputStream( dest ) );
        try{
            for(File f : filesToTar){
                out.putNextEntry(new TarEntry(f, f.getName()));
                BufferedInputStream origin = new BufferedInputStream(new FileInputStream( f ));

                int count;
                byte data[] = new byte[2048];
                while((count = origin.read(data)) != -1) {
                    out.write(data, 0, count);
                }

                out.flush();
                origin.close();
                f.delete();
            }
        } catch (IOException ex) {
                Logger.getLogger(Encoder.class.getName()).log(Level.SEVERE, null, ex);
        } finally {
            try {
                out.close();
            } catch (IOException ex) {
                Logger.getLogger(Encoder.class.getName()).log(Level.SEVERE, null, ex);
            }
        }
    }
}
```

Decoder.java

```
package codec;

/**
 *
 * @author Vicent Roig & Igor Dzinka
 */
public class Decoder {
    public static final int BUFFER_SIZE = 1024;
    private static ArrayList <Byte> builder;
    private static ArrayList<BufferedImage> compressed;
    private static int index;

    public Decoder(){ /*...*/ }
```

```
    /**
     * Decoding process
     *
     * @param packed Tar file that contains the compressed video  file
     *               and the  gzip with a info neccessary to decode the video
     * @return Array of decoded video frames
     */
    public static ArrayList <BufferedImage> decode(File packed){
        System.out.println("@decode receives "+ packed.getName());
        try {
            untar(packed);
        } catch (Exception ex) {
            Logger.getLogger(Decoder.class.getName()).log(Level.SEVERE, null, ex);
        }
        //printBuilder();
        build();
        return compressed;
    }

    /**
     * Method that reconstructs the video frames using the builder skeleton
     *
     * BUILDER SKELETON STRUCT. (each X represents 1 byte)
     *    X   /   X   /   X   |    X    /    XX    /     XX XX            XX      [...] |    X    (etc)
     *< brick    gop    rows  |  pframe   matches    pf match (x,y)   iF N-chunk  [...] |  pframe (etc)>
     */
    private static void build(){
        index = 0;

        System.out.println("@build -> Reconstructing video...");
        // load builder skeleton header:
        byte brick = builder.get(index++);
        byte gop = builder.get(index++);
        byte rows = builder.get(index++);

        while(index < builder.size()){
            byte k = builder.get(index++);
            short matches = getNextShort();

            //loop over matches
            for(int i=0; i < matches; i++){
                short coord[] = getNextCoord();
                short h = getNextShort();

                BufferedImage chunk = compressed.get((k/gop)*gop).getSubimage((h%rows)*brick, (h/rows)*
brick, brick, brick);
                Graphics2D g2 = compressed.get(k).createGraphics(); //pframe drawer
                g2.drawImage(chunk, coord[0], coord[1], null);
            }
        }
        builder.clear();
        System.out.println("@build -> Reconstruction DONE!");
    }

    /**
     * Method that prints the content of builder byte array     */
    private static void printBuilder(){
        for (byte b : builder) {
            System.out.print(String.format("%02X ", b));
        }
    }
```

```
    /**
     *
     * @see /codec/Encoder.serializeCoord()
     * @return
     */
    private static short[] getNextCoord(){
        short coord[] = new short[2];
        for(int i=0; i<2; i++) coord[i] = getNextShort();
        return coord;
    }

    /**
     * Method to get next short value from the builder byte array
     * @return short  extracted from builder
     */
    private static short getNextShort(){
        //(short) ( ((hi & 0xff) << 8) |  (lo & 0xff) )
        return (short) ( (builder.get(index++) & 0xff) |  ((builder.get(index++) & 0xff) << 8 ) );
    }

    /**
     * Recovery for builder skeleton & base JPEG sequence of frames
     *
     * @param encoded   File .tar codified with Encode.java
     * @return          result decompressed frame sequence
     */
    private static File[] untar(File encoded) throws Exception{
        String tarFile = encoded.getName();
        File[] output = new File[2];

        // Create a TarInputStream
        TarInputStream tis;
        try {
            tis = new TarInputStream(new BufferedInputStream(new FileInputStream(tarFile)));

            TarEntry entry;
            int id = 0;
            while((entry = tis.getNextEntry()) != null) {

                byte data[] = new byte[BUFFER_SIZE];
                FileOutputStream fos;
                fos = new FileOutputStream(entry.getName());
                BufferedOutputStream dest = new BufferedOutputStream(fos);

                int count;
                while((count = tis.read(data)) != -1) {
                  dest.write(data, 0, count);
                }

                dest.flush();
                dest.close();

                //Once extracted read as tmp files. We should delete them after processing.
                System.out.print("\t File " + entry.getName() + " extracted" );
                File tmp = new File(entry.getName());
                if(id==0){
                    System.out.println("\t\t-> @readZip");
                    compressed = ZipSaveWorker.readZip(tmp);
                } else if(id==1 && entry.getName().equals(Encoder.BUILDER_FNAME)){
                    System.out.println("\t-> @decompressGzipFile");
                    builder = decompressGzipFile(entry.getName(), BUFFER_SIZE);
                }
```

```
                tmp.delete();
                id++;
            }
            tis.close();

        } catch (FileNotFoundException ex) {
            Logger.getLogger(Decoder.class.getName()).log(Level.SEVERE, null, ex);
            return null;
        } catch (IOException ex) {
            Logger.getLogger(Decoder.class.getName()).log(Level.SEVERE, null, ex);
            return null;
        }

        return output;
    }



    ////////////////////////////////////////////////////
    //               GZIP skeleton recover            //
    ////////////////////////////////////////////////////

    /**
     * Get byte array from GZip file
     * -----------------------------
     * @param file .tar codified with Encode.java
     * @param bufferlength size for buffering bytes. Recommended 1024/2048
     *
     * @return decompressed frame sequence
     */
    private static ArrayList <Byte> decompressGzipFile(String filename, int bufferlength) {
        ArrayList <Byte> data = new ArrayList<>();
        byte[] buffer = new byte[bufferlength];
        try {
            FileInputStream fis = new FileInputStream(filename);
            GZIPInputStream gis = new GZIPInputStream(fis);

            int bytes_read = -1;
            while ((bytes_read = gis.read(buffer)) != -1){
                for(int i=0; i<bytes_read; i++) data.add(buffer[i]);
            }

            //bos.close();
            gis.close();
            fis.close();
            System.out.println("The file was decompressed successfully!");
        } catch (IOException e) {
            return null;
        }

        return data;
    }


}
```