UNIVERSIDAD AUSTRAL DE CHILE.

CAMPUS MIRAFLORES.

“PROYECTO1:INFO 185- COMUNICACIONES “

PRESENTAN:

DIEGO ROJAS ASENJO
DANIEL GARCIA ARCE

ING. CIVIL EN INFORMÁTICA

FECHA DE ENTREGA :23/ABRIL/2018

Informe proyecto 1: INFO 185: Comunicaciones

Índice:

PROBLEMÁTICA:

Para el desarrollo de este proyecto se planteó una problemática con el CENCO (Centro de Comunicaciones (CENCO) de carabineros de Valdivia sin embargo en la municipalidad de Valdivia se instalaron cámaras de vigilancia con la finalidad de detectar situaciones que requieran de presencia policial.

Por otro lado la municipalidad de Valdivia nos solicitó el desarrollo de un par codificador y decodificador para el flujo de imágenes crudas. Con los siguientes requisitos :

La salida del codificador debe ser un código de largo de palabra variable con el enlace entre la cámara y el CENCO es de 20 Mbps.

Especificaciones de la cámara.

- Sensor CCD con 3 canales RGB
- Resolución de 1920x1080 píxeles.
- Tasa de 10 cuadros por segundo

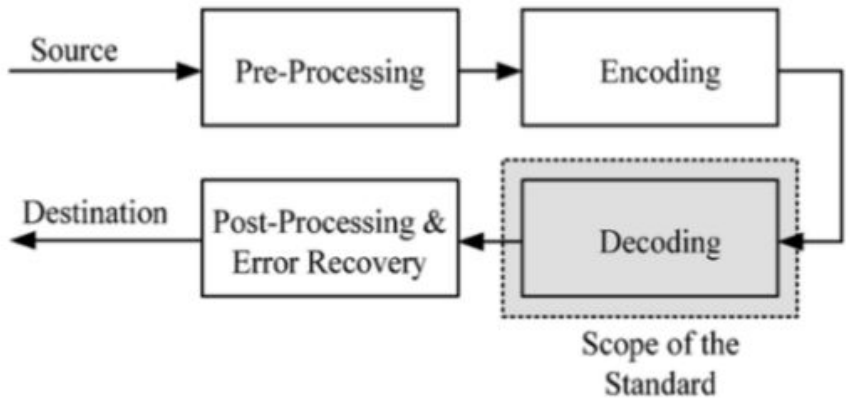
Para ello se tiene que realizar un programa en python que aplique estos requisitos al codificador basado con los temas vistos tales como : DCT (transformada discreta de fourier).

OBJETIVO:

Desarrollar y programar en python un par codificador y decodificador para el flujo de imágenes crudas de la cámara de vigilancia. Donde involucre la compresión de imágenes de transformación, cuantización y codificación, que permitan cumplir con los requerimientos para el enlace hasta CENCO.

METODOLOGÍA:

Para el desarrollo del software se utilizó el siguiente modelo que fue presentado en avances preliminares estas etapas son las siguientes:



metodología implementada.

DESARROLLO:

ETAPA 1: Pre-Processing y Encoding.

1.- Se inició con la investigación teórica con las técnica de compresión de imágenes , realizando búsquedas en la internet y paper en pdf acerca del tema.

2.- Posteriormente se llevó a cabo las primeras codificaciones de compresión de imagen utilizando funciones básicas en python.

3.- Se consultó en algunos libros y en sitios web acerca de la transformada discreta coseno de fourier que nos ayudaron a implementar una codificación básica de la transformación de imágenes de RGB a YCbCr.

4.- Como se muestra en la figura 1.1-"código Ycbcr en python". aquí fue donde iniciamos a entender cómo funciona la función y que valores contiene.

```
    y = int(.299*r + .587*g + .114*b)
    cb = int(128 -.168736*r -.331364*g + .5*b)
    cr = int(128 +.5*r - .418688*g - .081312*b)
    return y, cb, cr
```

fig 1.1- función YCbCr en python".

5.- Posteriormente se codificó el preprocessing de una imagen de entrada (imagen rgb), dicha imagen se llevó a cabo la conversión a ycbcr y como resultado se obtuvo la **DCT(transformada discreta de coseno)** en la siguiente imagen. (véase fig.1.2).

Código en python.

```
def dctTransform(matrix):


    dct1 = np.zeros((8,8))

    m = len(matrix)
    n = len(matrix)
    for u in range(m):
        for v in range(n):

            if (u == 0):
                cu = 1 / sqrt(m)
            else:
                cu = sqrt(2) / sqrt(m)


            if (v == 0):
                cv = 1 / sqrt(n)
            else:
                cv = sqrt(2) / sqrt(n)


            sm = 0
            for x in range(m):
                for y in range(n):
                    dct2 = matrix[x][y] * cos(((2 * x + 1) * u * pi) / (2 * m)) * cos(((2 * y + 1) * v * pi) / (2 * n))
                    sm = sm + dct2


            dct1[u][v] = cu * cv * sm

    return dct1
```

fig. 1.2- Conversión de imagen real RGB (3.9mb.) - YCbCr(239KiB).

ETAPA 2: Decoding.

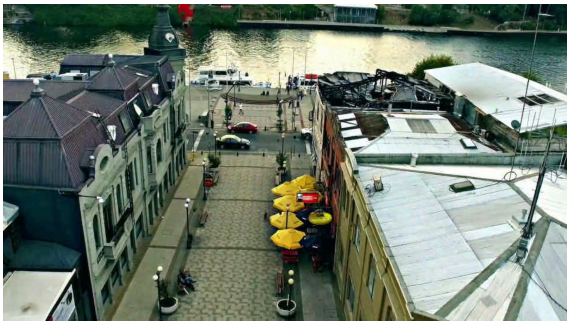
6.- Una vez obtenido la imagen en DCT en comprensión YCbCr procedimos a la transformación discreta de fourier inversa de la DCT. a IDCT para obtener la imagen original de entrada.

Código en python.

```
################## proceso completo de transformacion#########################
def proces(imagen):

    ancho,altura,pixels,im = cargar(imagen)
    i = 0
    j = 0

    while i < altura: ##recorre la imagen
        while j < ancho:
            if ancho - j >= 8 and altura - i >= 8:

                mat = []
                for k in range(8): ##bloques de ocho
                    mat.append([])
                    for l in range(8):

                        mat[k].append(pixels[l+j,k+i][0])


                mat = temmat(mat)
                mat = trans(mat)
                finaly = regr(mat)


                mat1 = []
                for k in range(8): ##bloques de ocho
                    mat1.append([])
                    for l in range(8):

                        mat1[k].append(pixels[l+j,k+i][1])


                mat1 = temmat(mat1)
                mat1 = trans(mat1)
                finaly1 = regr(mat1)


                mat2 = []
                for k in range(8): ##bloques de ocho
                    mat2.append([])
                    for l in range(8):

                        mat2[k].append(pixels[l+j,k+i][2])


                mat2 = temmat(mat2)
                mat2 = trans(mat2)
                finaly2 = regr(mat2)

                for k in range(8): ##pintamos nuevamente
                    for l in range(8):
                        pixels[j+l,i+k] = (int(finaly[k][l]),int(finaly1[k][l]),int(finaly2[k][l]))
```

fig. 1.3- “conversión DCT(239KiB) - IDCT(238.2KiB)”.

fig. 1.4- “conversión de IDCT(238.2KiB) - RGB final(267.7KiB)”.



Tiempo de ejecución DCT-PYTHON 1:09ms, 93% Compresión, ERR(rojo): 41%, ERR(verde): 46%, ERR(azul): 38%
Tiempo de ejecución DCT-MANUAL 32:45ms, 93% Compresión, ERR(rojo): 42%, ERR(verde): 46%, ERR(azul): 38%

ETAPA 3: Post-processor & Error Recovery.

7.- Para determinar el margen de error de la imagen obtenida se creó un código que calcula en porcentajes de error los valores de la imagen en RGB para saber cuales son las posibles pérdidas de la imagen por color.

Cálculo de pérdida de imagen

```
def calc(image):
    im = Image.open(image)
    im1 = Image.open("rgb.jpg")

    a = im.load()
    b = im1.load()

    dm = im.size

    acm=0
    rojo=0
    verde=0
    azul=0
    for i in range(dm[0]):
        for j in range(dm[1]):
            if a[i,j][0]!=0:
                rojo = ((a[i,j][0]-b[i,j][0])/a[i,j][0])*100 + rojo
            if a[i,j][1]!=0:
                verde = ((a[i,j][1]-b[i,j][1])/a[i,j][1])*100 + verde
            if a[i,j][2]!=0:
                azul = ((a[i,j][2]-b[i,j][2])/a[i,j][2])*100 + azul


    rojo = abs(rojo/(dm[0]*dm[1]))
    verde = abs(verde / (dm[0]*dm[1]))
    azul = abs(azul / (dm[0]*dm[1]))

    print("Porcentaje error por color a nivel gral")
    print("{0}%: Rojo - {1}%: Verde - {2}%: Azul".format(rojo,verde,azul))
    print(time.strftime("%H:%M:%S"))
    return time.strftime("%H:%M:%S"), rojo, verde, azul
```

Observaciones y Resultados

Como parte del proyecto en la etapa anterior se llevaron acabo datos estadísticos donde se determina los siguientes aspectos de comprensión que nosotros estimamos:

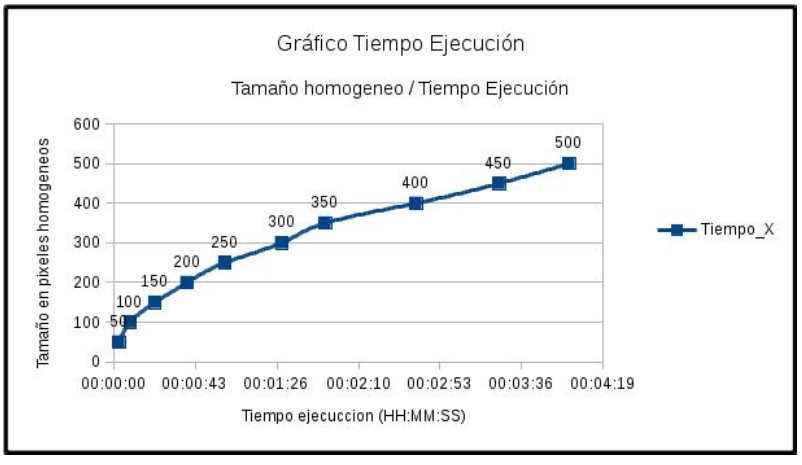
- **Ejecución**
- **Tasa Compresión**
- **Error RGB**

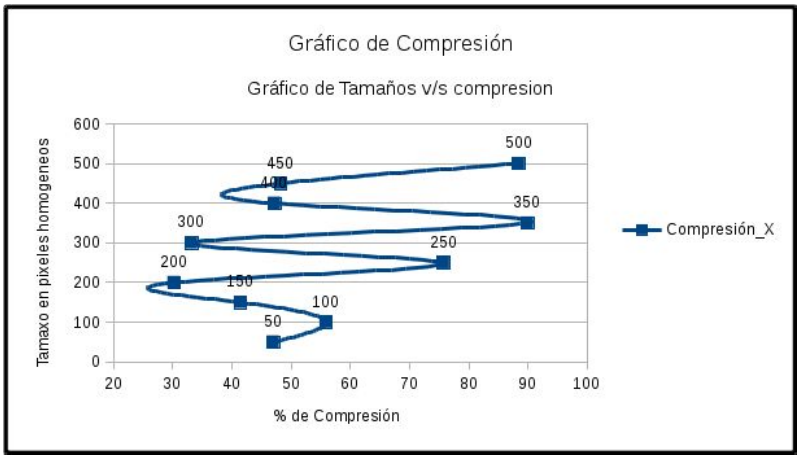
Con el fin de investigar cual seria la forma más adecuada de implementar , optimizar y mejorar el software.

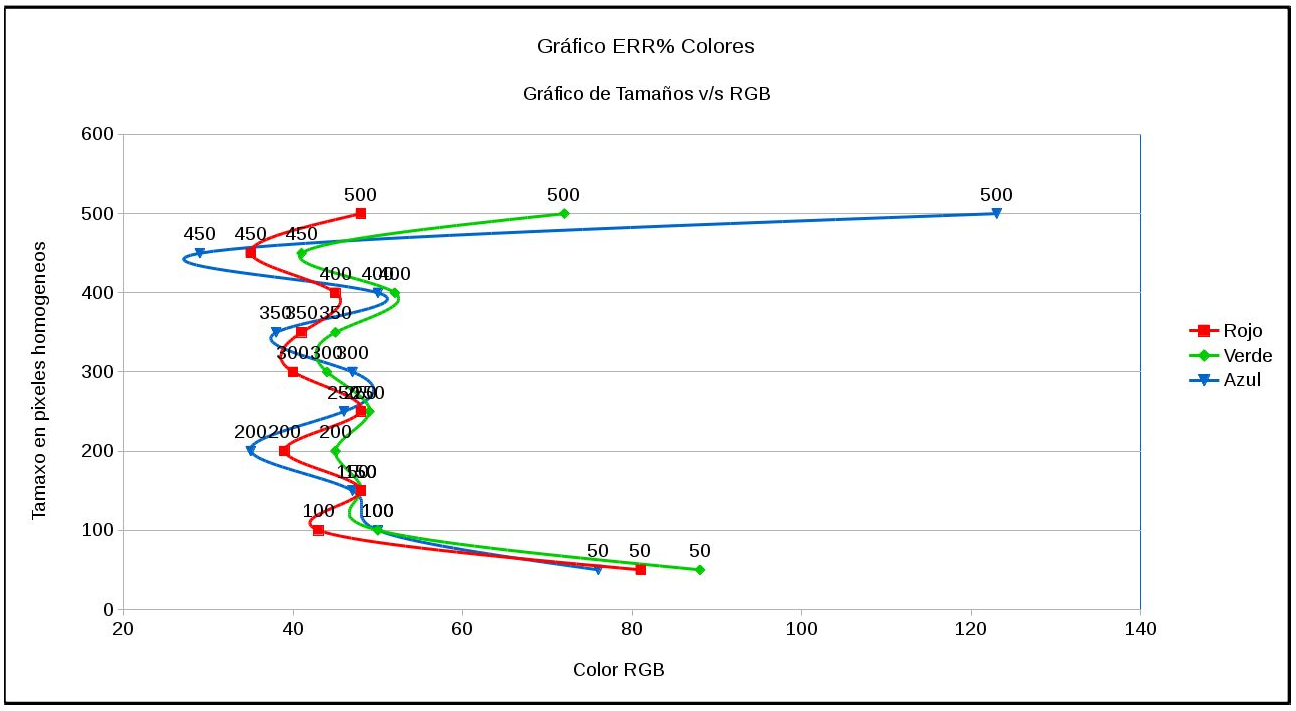
Resultados No.1:

Gráficos corresponden a experimentos tomados con lotes de imágenes con DCT manual.

TAMANO	Tiempo Inicia	Tiempo Final	tamaño inicia	tamaño final	Compresión_	Rojo	Verde	Azul		Tiempo_X
50	16:29:45	16:29:48	2783	1475	46,99964068	81	88	76		00:00:03
100	16:14:09	16:14:18	5663	2496	55,92442168	43	50	50		00:00:09
150	16:25:11	16:25:33	7738	4529	41,47066425	48	48	47		00:00:22
200	16:14:38	16:15:17	7858	5480	30,26215322	39	45	35		00:00:39
250	16:15:17	16:16:16	33738	8182	75,74841425	48	49	46		00:00:59
300	16:16:17	16:17:46	22320	14902	33,23476703	40	44	47		00:01:29
350	16:17:46	16:19:38	208241	20973	89,92849631	41	45	38		00:01:52
400	16:19:38	16:22:18	47457	25038	47,24065997	45	52	50		00:02:40
450	16:22:19	16:25:43	41700	21580	48,24940048	35	41	29		00:03:24
500	16:25:44	16:29:45	562517	65112	88,42488316	48	72	123		00:04:01





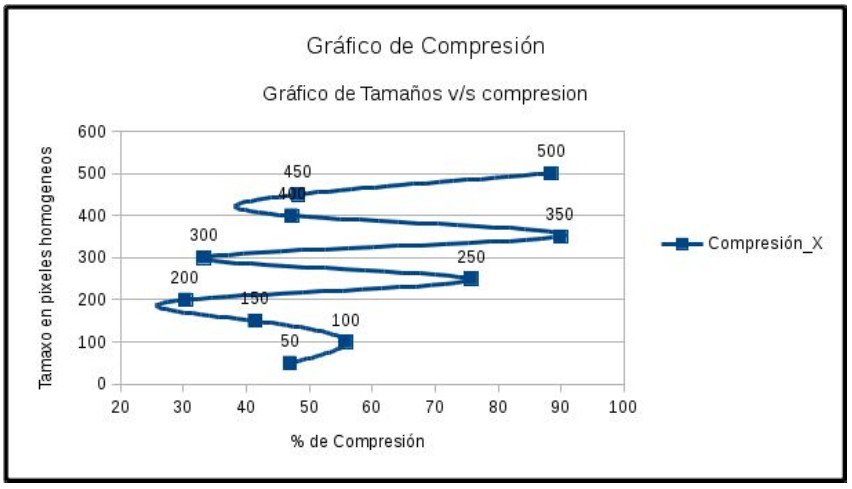


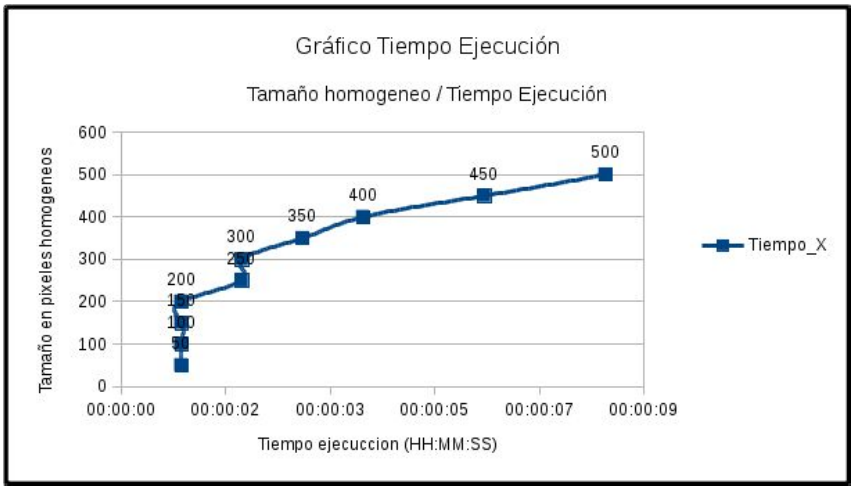
DATOS ESTADÍSTICOS Y GRÁFICOS DE COMPRESIÓN DE IMÁGENES.

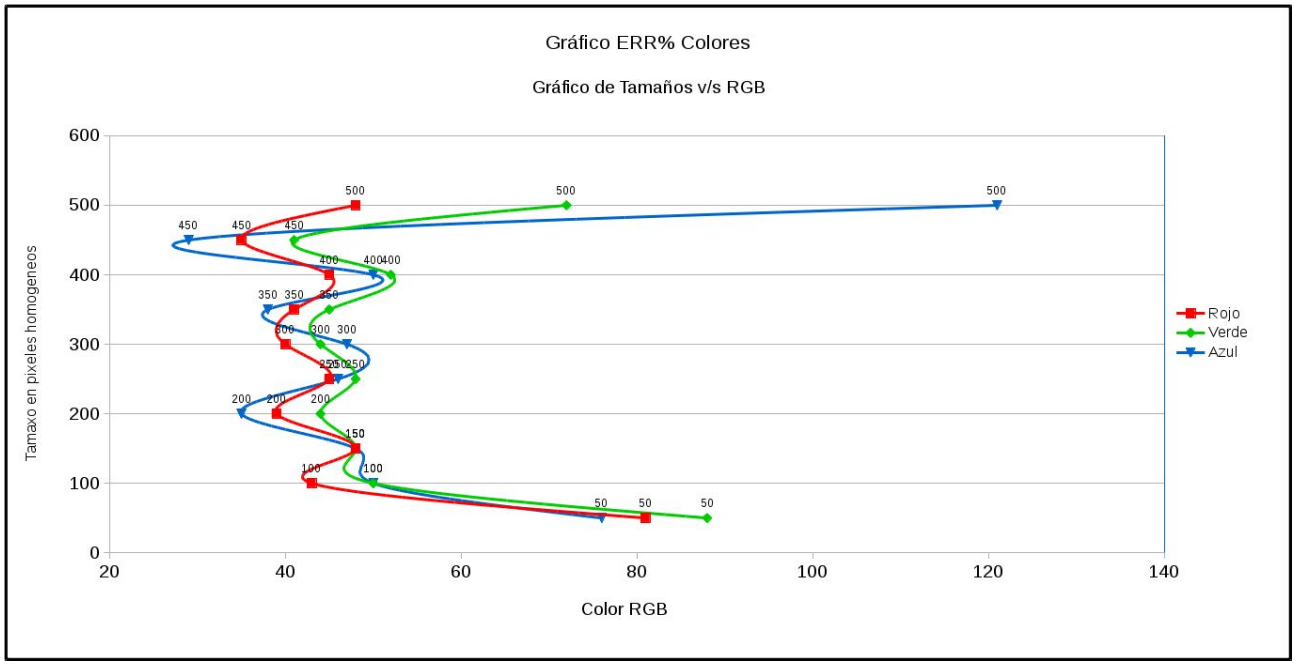
Resultados No.2:

Gráficos corresponden a experimentos con imágenes pasando con la DCT de python.

TAMAÑO	Tiempo Inicia	Tiempo Final	tamaño inicia	tamaño final	Compresión_X	Rojo	Verde	Azul		Tiempo_X
50	18:32:34	18:32:35	2783	1476	46,96370823	81	88	76		00:00:01
100	18:32:02	18:32:03	5663	2502	55,81847078	43	50	50		00:00:01
150	18:32:03	18:32:04	7738	4535	41,39312484	48	48	48		00:00:01
200	18:32:04	18:32:05	7858	5473	30,35123441	39	44	35		00:00:01
250	18:32:06	18:32:08	33738	8180	75,75434228	45	48	46		00:00:02
300	18:32:08	18:32:10	22320	14892	33.2795698925	40	44	47		00:00:02
350	18:32:11	18:32:14	208241	21006	89,91264929	41	45	38		00:00:03
400	18:32:15	18:32:19	47457	25033	47,25119582	45	52	50		00:00:04
450	18:32:20	18:32:26	41700	21609	48,17985612	35	41	29		00:00:06
500	18:32:26	18:32:34	562517	65149	88,41830558	48	72	121		00:00:08







DATOS ESTADÍSTICOS Y GRÁFICOS DE COMPRESIÓN DE IMÁGENES.

Conclusión:

Se llegó a la conclusión que, durante el desarrollo de este tipo de software, logramos notar la complejidad de implementar los procedimientos matemáticos en la programación, y encontrar la forma de optimizar y mejorar el software de manera manual, debido que estaba orientado a desarrollarse sin alguna librería disponible de cálculo, ya que el tiempo de desarrollo fue corto como equipo aplicado metodología de programación extrema con fases de prototipado, y buscar herramientas factibles para la implementación de compresión de imágenes, gracias a la literatura.

Mediante los resultados pudimos encontrar que las curvas de compresión, error por color y tiempo de ejecución. Podemos decir que la curva por “compresión” tanto en DCT-PYTHON y DCT-MANUAL se aprecia una leve tendencia “lineal” por el cómo se eleva en zig-zag la curva a medida que aumenta el tamaño de la imagen.

Con respecto a las gráficas de error por colores RGB en DCT-PYTHON y DCT-MANUAL tienen el mismo comportamiento, indicando que se genera una cierta estabilidad de rango de $52 \geq ERR(RGB) \geq 38$ entre 450x450px ≥ tamaño ≥ 100x100px, descartando los puntos fronteras, siendo éstos los límites del experimento en estos gráficos de RGB.

Finalizando con los gráficos “Tiempos de Ejecución”, apreciamos que posee un comportamiento logarítmico en los 2 gráficos, pero notoriamente distintos en valores que es de esperar, siendo el más costoso DCT-MANUAL con un tiempo máximo de ejecución 4:19 minutos de una imagen de 500x500, a diferencia de DCT-PYTHON con solo 9 segundos.

En base los datos “Tiempos de Ejecución” con respecto al código DCT-MANUAL, es inevitable no poder tener un tiempo de ejecución tan costoso, debido que para realizar la éste cálculo, debe realizar 4 instrucciones “FOR” anidadas, que aumenta cuadráticamente su orden de ejecución por la estructura que posee, aunque el gráfico refleja otra condición. Siguiendo la misma hipótesis, pudimos también ver la diferencia en ejecución del frame0677.PNG por nuestro codificador/decodificador, entregando un tiempo muy alto de ejecución comparado a la DCT-PYTHON, siendo esto incompatible con el requerimiento solicitado para el enlace.

Finalmente con respecto al procesamiento del frame0677.PNG, pudimos notar su alto porcentaje de compresión llegando a un 93%, perdiendo calidad sin sobrepasar el 50% de distorsión por color, que si en tales casos necesite pasar 10 frames x segundo a una tasa de 20mbps en el enlace llegaríamos que:

$[(267.7 * 10) / 1024] * 8 = 20.91$mbps para el enlace

Que para lograr el objetivo, estamos pasando el requerimiento de transferencia por enlace, pero aún se puede disminuir esa tasa, lo que podemos decir que ésto es lograble si cambiaramos el sistema de cuantización por uno dinámico, probablemente usando algunos filtros pasa altos que permitan disminuir aún más la compresión por frame, usando algoritmos de codificación para redundancias y recuperación, para disminuir errores en las escalas de recuperación.

Bibliografía:

[Repositorio de Experimento] - https://github.com/hackerter/ConversorImagenes.git.

- Video Compression—From Concepts to the H.264/AVC Standard - GARY J. SULLIVAN , SENIOR MEMBER, IEEE AND THOMAS WIEGAND
- The JPEG Still Picture Compression Standar – Gregory K. Wallace
- The H.264 Video Compression Standard - Till Halbach