Application Acceleration with High-Level-Synthesis

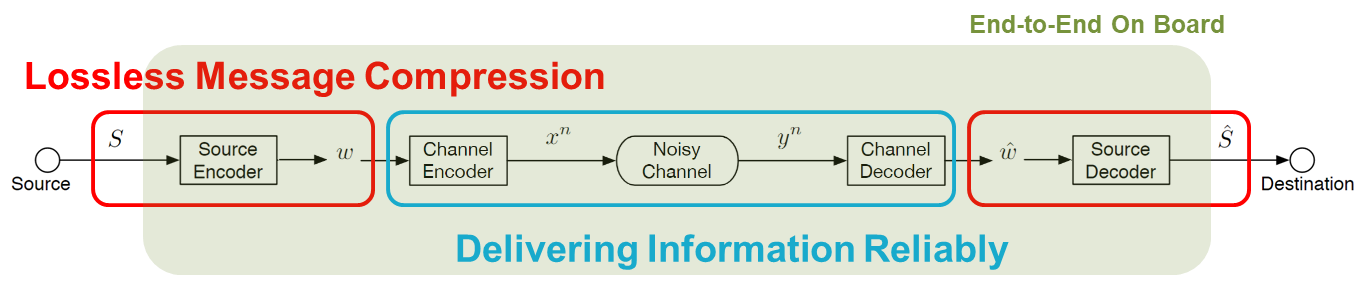
Final Project

Team 06 組員: 王景平 / 黃其澤 / 陳維隆

Title: Message Communication

Project Goal:

Full run including **Encoding**, **Channel Simulation**, and **Decoding**



圖一、Final Project 系統架構示意圖

此Project包含以下三個模組，後續內容將介紹下述演算法與硬體實作技術。

1. Source Coding (Image Compression) – QOI (Quite OK Image Format)
2. Channel Coding (Error Correction Code) – Polar Code
3. Simulated AWGN Channel

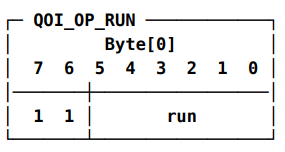
**Source Coding (Image Compression) – QOI (Quite OK Image Format)**

1. Algorithm

QOI透過分析不同資料特性，對於不同的pattern採用不一樣的壓縮資料格式，以達到較好的壓縮比。QOI定義了以下5種case以利資料壓縮，這五個case是sequential的去比對，從壓縮率最高的case比較到需要額外overhead的case，為了使解碼方便，所以OQI定義均以一個Byte為單位。

1. RUN

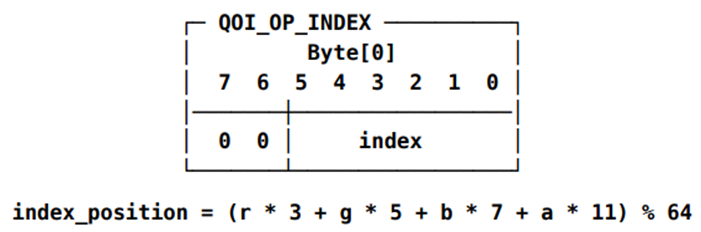
RUN代表目前的pixel與前一個pixel數值完全相同，將連續相同pixel數值用run來表示，對於較低頻，有大塊色塊的圖片能夠帶來很好的壓縮效果。



圖二、QOI RUN 資料格式

1. INDEX

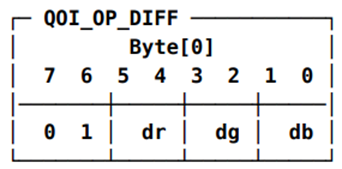
INDEX代表與為先前看過的pixel值相同，透過hash table有效的搜索相同的pixel，以達到高效率的壓縮。



圖三、QOI INDEX 資料格式與雜湊(Hash)函數

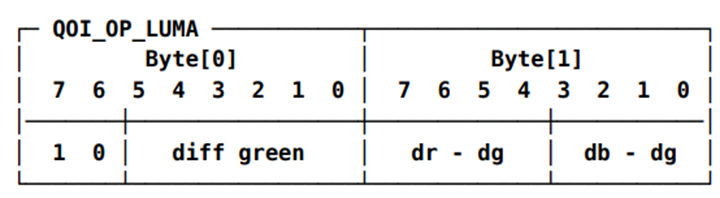
1. DIFF

DIFF代表與前一個pixel各個顏色色值都小於2，則可以透過去紀錄與前面之pixel的difference以減少整體儲存空間。



圖四、QOI DIFF資料格式

1. LUMA

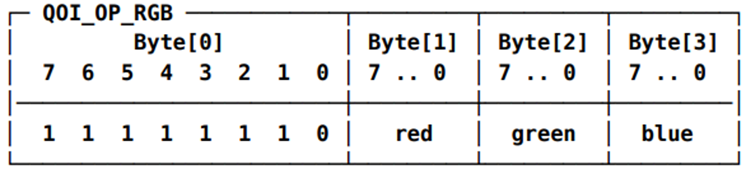


圖五、QOI LUMA資料格式

LUMA代表與前一個pixel綠色色值差值絕對值小於32，而紅藍的差值也與綠色差值相近，則可以透過去紀錄與前面之pixel的difference及各顏色之間的差異以減少整體儲存空間。

1. RGB

若上述case均不符合，即會記錄原始的RGB，也代表這個pixel相較於其他pixel較無規律可以去壓縮，亦顯示對於這種不平常的pixel沒有壓縮的必要。

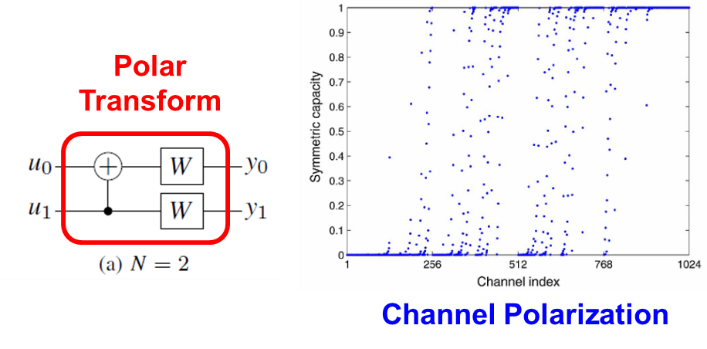


圖六、QOI RGB資料格式

**Channel Coding (Error Correction Code) – Polar Code**

1. Algorithm

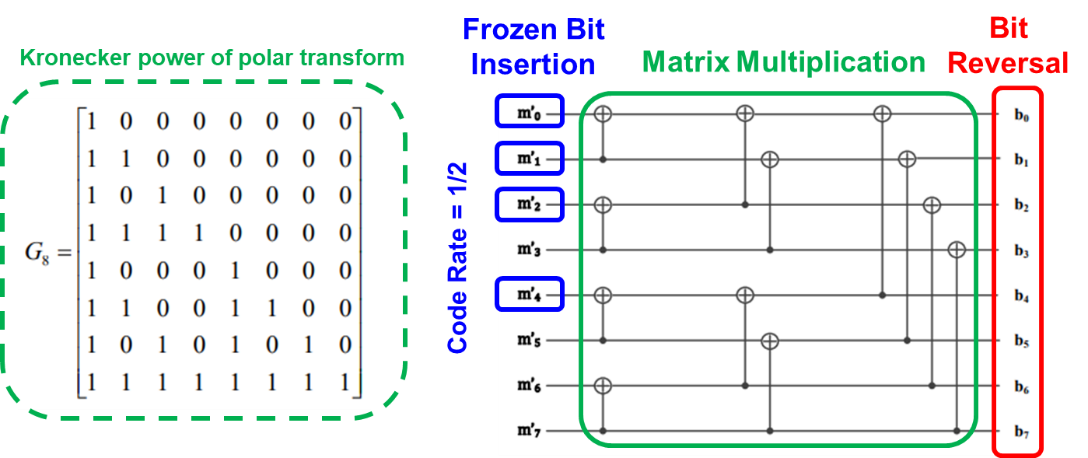
Polar Code 透過polar transform將上半部的channel資訊分布至下半部的channel，並利用多個channel的階層式轉換，分出reliable和unreliable的channels。



圖七、Polar Transform與Channel Polarization效果

Polar Code透過在unreliable channels傳輸固定的數值(0)，以去除noise對於這些channel嚴重的影響，而Encoder演算法實作上主要就分成三個部分。

1. 插入frozen bits於unreliable channels。
2. 與Generator Matrix進行矩陣乘法。
3. 對於output channel進行Bit Reversal order的排列。



圖五、Polar Encoder架構示意圖

Decoder部分我們則是採用belief propagation去實作，利用收到的LLR(Log Likelihood Ratio)去計算每個node的maximum likelihood，Decoder即可利用Encoder原有之架構進行forward和backward pass去更新並透過iterative的decoding解出可能的message。

1. Hardware

Encoder

1. 將矩陣乘法與bit reversal的部分fold起來以加速整體encode時間。
2. Array partition / Pipeline / Unroll優化polar encoder

首先，我們會透過額外生成的reliability sequence去排列message實際上的位置，由於前半部QOI壓縮出來的數值為一個Byte，故我們會透過shift-register和bit-wise and去取出每個bit的資訊。後半部則是採用過去Lab實作過的矩陣乘法加速，並透過array partition / pipeline / unroll達到hardware resource和efficiency較好的trade-off。

Polar Encode Code

**Bit** dst\_tmp[N];

**Bit** bit\_stream[N];

**#pragma** HLS ARRAY\_PARTITION variable=bit\_stream complete

**Byte** data;

**int** count\_1 = 0;

**int** count\_2 = 0;

**int** count\_3 = 0;

**for** (**int** i = 0; i < BUF\_SIZE; i += num\_load) {

**for**(**int** j = 0; j < N; j++){

**#pragma** HLS PIPELINE

**if**(j < K){

count\_1 = j%8;

**if**(count\_1 == 0){

data = src[count\_2++];

}

bit\_stream[reliability\_sequence[N-K+j]] = data & 1;

data >>= 1;

}

**else**{

bit\_stream[reliability\_sequence[j-K]] = 0;

}

}

// polar Transformation

**for**(**int** j=0; j<N; j++){

**Bit** tmp = 0;

**#pragma** HLS PIPELINE

**for**(**int** k=0; k<N; k++){

tmp ^= bit\_stream[k] & polar\_transform[k][j];

}

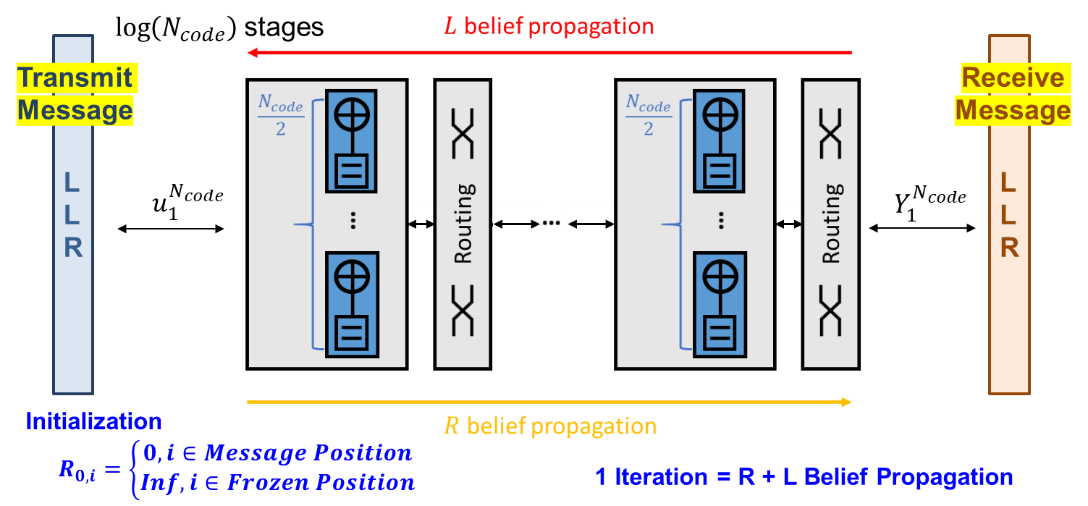
dst[count\_3++] = tmp;

}

}

Decoder

1. 實踐常見論文中PE Reusing的架構，透過Routing改變每一層的接線以達到不同stage間不同資料流向更新的功能。
2. Unified PE 將左右的資料更新統一fold到routing當中以降低整體PE複雜度，並參考多篇paper選用優化後的minsum機制以達到較好的解碼效果。
3. 透過partition / unroll利用HLS達到與過去文獻相同的硬體架構實現。



圖五、Polar Decoder (Belief Propagation) 架構示意圖

基於上述架構，我們會將硬體實作分成下列三部份。主要為PE Data Fetching、PE Computing、Routing以實作不同layer間belief propagation的llr更新。由於每一個stage的接線較為繁雜，用變數儲存並取用會造成整體latency較長，故我們有一大部分code使用python code去展開並將routing事先生成好利用硬體資源減少整體運行時間。後續則使用array partition和unroll達到平行化的硬體架構。

PE Data Fetching

**#pragma** HLS ARRAY\_PARTITION variable=left\_up complete

**#pragma** HLS ARRAY\_PARTITION variable=right\_up complete

**#pragma** HLS ARRAY\_PARTITION variable=left\_down complete

**#pragma** HLS ARRAY\_PARTITION variable=right\_down complete

**switch**(depth){

**case** 0:

**for**(**int** pe\_num=0; pe\_num < N\_div2; pe\_num++){

**#pragma** HLS UNROLL

left\_up[pe\_num] = R\_tx\_llr[(pe\_num << 1)];

left\_down[pe\_num] = R\_tx\_llr[(pe\_num << 1)+1];

right\_up[pe\_num] = shared\_memory\_1[(pe\_num << 1)];

right\_down[pe\_num] = shared\_memory\_1[(pe\_num << 1)+1];

}

**break**;

PE Computing

**for**(**int** pe\_num = 0; pe\_num < N\_div2; pe\_num++){

**#pragma** HLS UNROLL factor=32

**xor\_rep** result;

**float** input\_u1, input\_u2, input\_d1, input\_d2;

result = PE(right\_up[pe\_num], right\_down[pe\_num], left\_up[pe\_num], left\_down[pe\_num]);

result\_xor[pe\_num] = result.r\_xor;

result\_rep[pe\_num] = result.r\_rep;

}

Routing (Partial)

**switch**(depth){

**case** 0 :

shared\_memory\_1[0]=result\_xor[0];

shared\_memory\_1[32]=result\_rep[0];

shared\_memory\_1[1]=result\_xor[1];

shared\_memory\_1[33]=result\_rep[1];

shared\_memory\_1[2]=result\_xor[2];

shared\_memory\_1[34]=result\_rep[2];

shared\_memory\_1[3]=result\_xor[3];

shared\_memory\_1[35]=result\_rep[3];

shared\_memory\_1[4]=result\_xor[4];

shared\_memory\_1[36]=result\_rep[4];

shared\_memory\_1[5]=result\_xor[5];

shared\_memory\_1[37]=result\_rep[5];

shared\_memory\_1[6]=result\_xor[6];

shared\_memory\_1[38]=result\_rep[6];

shared\_memory\_1[7]=result\_xor[7];

shared\_memory\_1[39]=result\_rep[7];

shared\_memory\_1[8]=result\_xor[8];

shared\_memory\_1[40]=result\_rep[8];

shared\_memory\_1[9]=result\_xor[9];

shared\_memory\_1[41]=result\_rep[9];

shared\_memory\_1[10]=result\_xor[10];

shared\_memory\_1[42]=result\_rep[10];

shared\_memory\_1[11]=result\_xor[11];

shared\_memory\_1[43]=result\_rep[11];

shared\_memory\_1[12]=result\_xor[12];

shared\_memory\_1[44]=result\_rep[12];

shared\_memory\_1[13]=result\_xor[13];

shared\_memory\_1[45]=result\_rep[13];

shared\_memory\_1[14]=result\_xor[14];

shared\_memory\_1[46]=result\_rep[14];

shared\_memory\_1[15]=result\_xor[15];

shared\_memory\_1[47]=result\_rep[15];

shared\_memory\_1[16]=result\_xor[16];

shared\_memory\_1[48]=result\_rep[16];

shared\_memory\_1[17]=result\_xor[17];

shared\_memory\_1[49]=result\_rep[17];

shared\_memory\_1[18]=result\_xor[18];

shared\_memory\_1[50]=result\_rep[18];

shared\_memory\_1[19]=result\_xor[19];

shared\_memory\_1[51]=result\_rep[19];

shared\_memory\_1[20]=result\_xor[20];

shared\_memory\_1[52]=result\_rep[20];

shared\_memory\_1[21]=result\_xor[21];

shared\_memory\_1[53]=result\_rep[21];

shared\_memory\_1[22]=result\_xor[22];

shared\_memory\_1[54]=result\_rep[22];

shared\_memory\_1[23]=result\_xor[23];

shared\_memory\_1[55]=result\_rep[23];

shared\_memory\_1[24]=result\_xor[24];

shared\_memory\_1[56]=result\_rep[24];

shared\_memory\_1[25]=result\_xor[25];

shared\_memory\_1[57]=result\_rep[25];

shared\_memory\_1[26]=result\_xor[26];

shared\_memory\_1[58]=result\_rep[26];

shared\_memory\_1[27]=result\_xor[27];

shared\_memory\_1[59]=result\_rep[27];

shared\_memory\_1[28]=result\_xor[28];

shared\_memory\_1[60]=result\_rep[28];

shared\_memory\_1[29]=result\_xor[29];

shared\_memory\_1[61]=result\_rep[29];

shared\_memory\_1[30]=result\_xor[30];

shared\_memory\_1[62]=result\_rep[30];

shared\_memory\_1[31]=result\_xor[31];

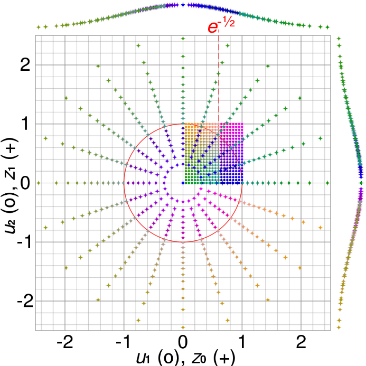
shared\_memory\_1[63]=result\_rep[31];

**break**;

**Simulated AWGN Channel**

1. Box-Muller Transform

透過下方公式則可將uniform distribution轉換成gaussian distribution。





圖x、Box-Muller Transform示意圖

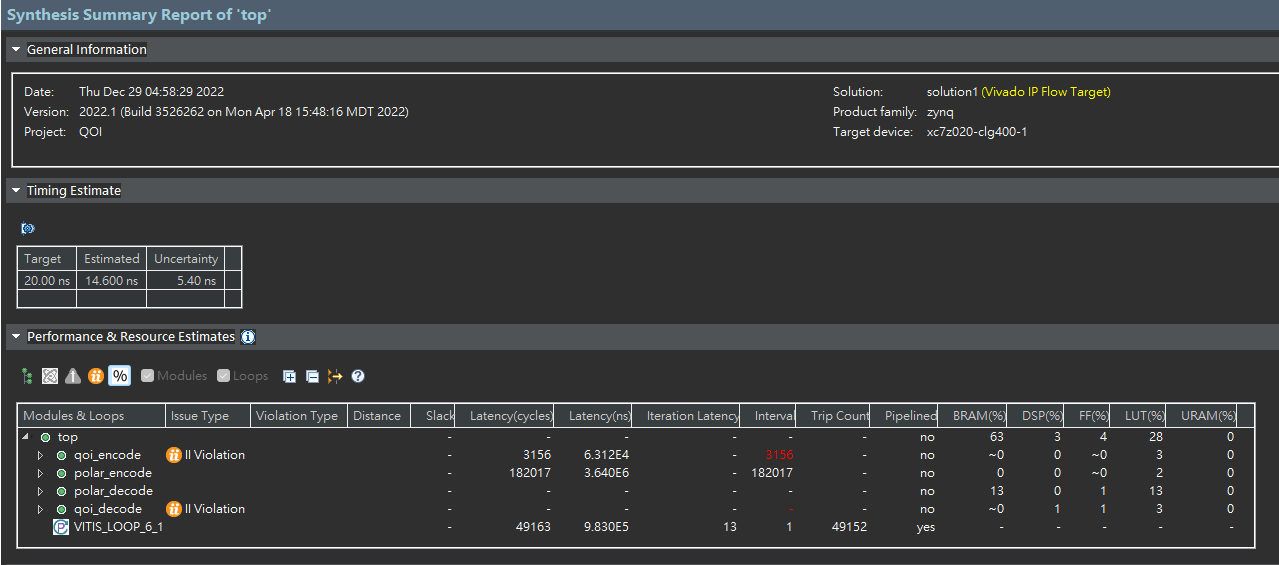
1. Hardware

我們透過look up table對cosine和log 兩函數去做近似，並透過c本身的rand函數產生uniform distribution並利用查找的方式計算出gaussian noise，以模擬傳輸中AWGN的情況。

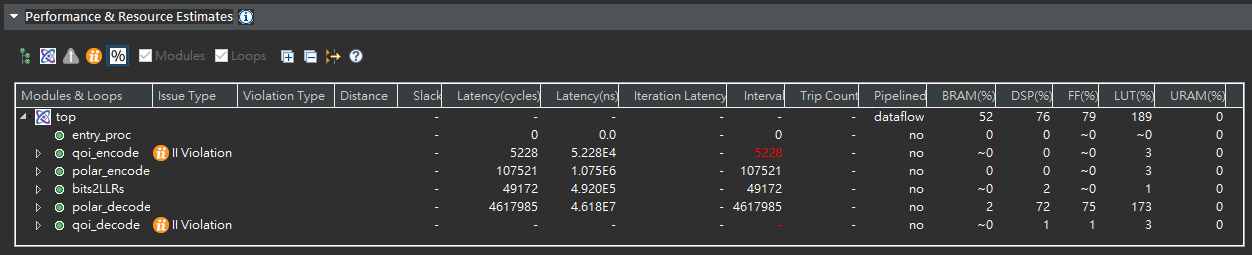
**Result**

12/29 cycle=20ns

此design最終有上PYNQ板子上驗證功能性，詳細application與功能描述於後面的DEMO中。



01/15 透過unroll和partition去加速整體運算，並將design優化到cycle=10ns

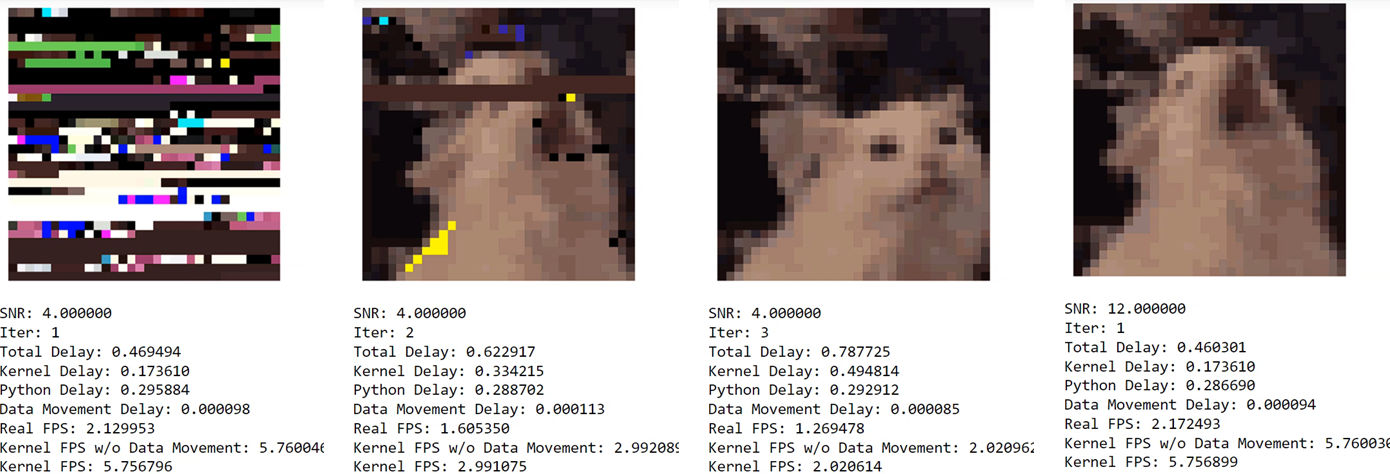


最後這個版本整體時長相較於前面節省許多，我們也嘗試透過fix-point analysis去減少整體資源用量，但當我們使用ap\_int之類的type去轉換時合成出來的design反而資源用量更多，我們認為可能是實際上PYNQ上面的routing以及相關的資源比較適合傳統的data type，若強制使用ap\_int反而會需要更多的LUT去完成較不規則的運算，故最終我們僅將12/29號的版本運行於PYNQ上。

**Demo (On Board)**

我們的實驗情境為透過傳輸多張GIF圖片的動畫，並於接收端觀察實際上收到的訊息，我們的架構本身Decoding Iteration數目和Channel的SNR都是reconfigurable的，故我們以下展示四種不同設定的實驗以驗證我們整體系統應用的正確性與效能。

1. 在low SNR且decoding較少iteration數目的情況下，可以看到我們的架構傳出來的pixel是有明顯錯誤的。
2. 在low SNR且decoding中等iteration數目的情況下，可以看到我們的架構傳出來的pixel是有明顯改善可以看到原始影片的大致內容。
3. 在low SNR且decoding較多iteration數目的情況下，可以看到我們的架構可以有效還原影片的品質。
4. 在high SNR且decoding較少iteration數目的情況下，可以看到我們的架構亦可以有效傳輸。

****