B. CÓDIGO FUENTE

B.1 Algoritmos

B.1.1 Función de obtención de los resultados

```
1  %
2  %                        BER_calc(encoderDVB, 'LLR')
3  %
4  % Esta funcion se encarga de llamar a los distintos algoritmos y de
5  % guardar los resultados que se van obteniendo. Recibe como argumentos
6  % el codificador (generado mediante encoderDVB=fec.ldpcencode(H)) y
7  % el algroritmo a usar.
8  %
9  % El algoritmo se pasa mediante una cadena de caracteres que puede ser:
10 %
11 % LLR: Version logaritmica del algoritmo Sumprod
12 % LLR_trunc: Version logaritmica del algoritmo Sumprod con la
13 %          tangente truncada.
14 % PW: Aproximacion lineal a trozos de la tanh del LLR
15 % LUT: Aproximacion constante a trozos de la tanh del LLR
16 % MS: Simplificacion de la regla de la tanh
17 % FI: Aproximacion del enfoque de gallager
18 %
19 % Durante la ejecucion se va escribiendo en el prompt de matlab
20 % '.-.' por cada palabra erronea y '.*.' por cada una correcta,
21 % esto facilita el observar el funcionamiento del programa.
22 %
23 % La funcion no devulve ningun valor. Los resultados son escritos
24 % en el fichero archivo_ALG_temp.mat, donde ALG es el algoritmo
25 % elegido. Durante la ejecucion se van escribiendo varios archivos
26 % de texto:
27 %
28 % SNR_Pe_ALG_temp.txt: En la primera columna se van escribiendo
29 % las SNRs y en la segunda las Pe. Abajo se presenta un resumen
30 % del tamano de paso, la ultima SNR calculada, el numero bits
31 % erroneos y el numero de bits totales de la ultima iteracion.
32
33 % Pal_erroneas_ALG.txt: Se van escribiendo para cada palabra
```

```
% decodificada el numero de errores encontrados y las palabras
% correctas e incorrectas
%
% Buclesup_ALG.txt. En este archivo se van escribiendo las Pe
% inferiores al minimo establecido.
%
% EJEMPLO DE USO:
%
%      BER_calc(encoderDVB, 'LLR')
%


function BER_calc(encoderDVB, ident)
format long

% Parametros internos de la funcion

Max_pal=10000;       % Maximo de palabras analizadas por cada punto
Min_err=1000;        % Minimo numero de errores detectados
Min_pal=50;          % Minimo numero de palabras erroneas detectadas
Ec=1;                % Energia de bit normalizada

% Parametros de la SNR
SNR_max=10;          % SNR_max inicial para la busqueda dicotomica.
SNR_p=0.4;           % Punto de comienzo del vector de SNRs
PASO=0.25;           % Tamano de paso de entre SNRs
i=1;                 % Numero de punto


% Creamos e inicializamos las variables que vamos a usar
r=zeros(1,encoderDVB.NumInfoBits);
c2=zeros(1, encoderDVB.NumInfoBits);
p_1=zeros(1, encoderDVB.NumInfoBits);
A=encoderDVB.ParityCheckMatrix;
seguir=1;
ult_error=0;
SNR_v=[];
Pe=[];



% Obtenemos la informacion necesaria de la matriz y la guardamos en un fichero
reordenat(A);


% Dependiendo del algoritmo selecionado usaremos una funcion u otra
```

```
80  switch ident
81    case 'LLR',
82      funcion_h=@LLR_dec;
83    case 'LLR_trunc',
84      funcion_h=@LLR_trunc_dec;
85    case 'PW',
86      funcion_h=@LLR_PW_dec;
87    case 'LUT',
88      funcion_h=@LLR_LUT_dec;
89    case 'MS',
90      funcion_h=@Minsum_dec;
91    case 'FI',
92      funcion_h=@Fi_func_dec;
93    otherwise,
94      error('No se ha introducido un identificador valido (LLR, LLR_trunc, PW, LUT,
          MS, FI) para el algoritmo ');
95  end
96
97
98  % Miramos si hay datos guardados de una ejecucion anterior (Modifica los
      parametros de
99  % la SNR)
100
101 archivo=exist(['archivo_',ident,'_temp.mat'],'file');
102 if(archivo)
103     load(['archivo_',ident,'_temp.mat']);
104 end
105
106
107
108 while 1                             % Bucle infinito
109
110     % Volvemos a inicializar las variables
111     errores=0;
112     total=0;
113     palabras=0;
114     pal_erroneas=0;
115     seguir=1;
116
117     while (seguir==1)
118         % Generamos un vector de informacion aleatorio, lo codificamos y lo
              enviamos por
119         % un canal gaussiano variando la SNR en cada punto
120
121         R=encoderDVB.NumInfoBits/encoderDVB.BlockLength;
122         infoDVB=((rand(1,encoderDVB.NumInfoBits)-0.5)>0);
```

```
123         codewordDVB=encode(encoderDVB, infoDVB);
124         [r, c2, p_1, Lc] = gausschbpsk(codewordDVB, Ec, SNR_p, R);
125
126         % Decodificamos la palabra usando el algoritmo seleccionado
127         [cest, sol] = funcion_h(A, r, Lc);
128
129
130         % Vemos si se ha decodificado correctamente y si no, contabilizamos los
                errores
131         % y actualizamos las variables para el siguiente punto.
132
133         palabras=palabras+1;
134         if(isequal(cest, codewordDVB))                  % Palabra correcta
135             fprintf(1,'.*.');
136         else                                            % Palabra incorrecta
137             errores=length(find(cest~=codewordDVB))+errores;
138             pal_erroneas=pal_erroneas+1;
139             fprintf(1,'\n.-.');
140         end
141
142         % Comprobamos si ya podemos pasar al siguiente punto
143
144         if(pal_erroneas>=Min_pal || palabras==Max_pal || errores>=Min_err)
145             if(pal_erroneas>=Min_pal)
146                 seguir=0;
147             end
148             if(errores<Min_err && palabras<Max_pal)
149                 seguir=1;
150             end
151             if (palabras==Max_pal)
152                 seguir=0;
153             end
154         end
155
156         % Calculamos el numero de bits enviados hasta el momento y guardamos la
157         % informacion provisional en el archivo "pal_erroneas_ALG.txt.
158
159         total=total+encoderDVB.BlockLength;
160         [fi4] = fopen(['pal_erroneas_',ident,'.txt'],'w');
161         fprintf(fi4, 'Palabras erroneas = %d; Palabras = %d Errores= %d\n',
                pal_erroneas, palabras, errores);
162         fclose(fi4);
163
164     end
165
166     % Vemos si el error es menor que el target (para la busqueda dicotomica) y
```

```
167     % actualizamos las variables para la proxima iteracion
168
169     ult_error=errores/total;
170     if(ult_error<5e-7)
171         [fi10] = fopen(['buclesup_',ident,'.txt'],'a');
172         fprintf(fi10, 'Error detectado= %12.10f, para SNR %12.10f\n', ult_error,
                SNR_p);
173         fclose(fi10);
174         SNR_max=SNR_p;
175         PASO=PASO/2;
176         SNR_p=SNR_v(i-1)+PASO;
177         i=i-1;
178
179     else
180         SNR_v(i)=SNR_p;
181         Pe(i)=errores/total;
182         if(SNR_p+PASO==SNR_max)
183             PASO=PASO/2;
184             SNR_p=SNR_p+PASO;
185         else
186             SNR_p=SNR_p+PASO;
187         end
188     end
189
190     % Guardamos los resultados
191     i=i+1;
192     v=clock;
193     s = datestr(v);
194     save(['archivo_',ident,'_temp.mat'], 'SNR_v', 'Pe', 'SNR_p', 'PASO', 'i', '
            SNR_max')
195     save(['archivo_',ident,'_temp.mat'], 'SNR_v', 'Pe', 'SNR_p', 'PASO', 'i', '
            SNR_max')
196     [fi3] = fopen(['avance_',ident,'.txt'],'a');
197     fprintf(fi3, '\nPunto %d %s\n', i, s);
198     st=fclose(fi3);
199     fprintf(1, '\nPunto %d\n', i);
200     [fi] = fopen(['SNR_Pe_',ident,'_temp.txt'],'w');
201     fprintf(fi, '%12.10f %12.10f\n', [SNR_v;Pe]);
202     st = fclose(fi);
203     [fi] = fopen(['SNR_Pe_',ident,'_temp.txt'],'a');
204     fprintf(fi, '\n
            **********************************************************\n');
205     fprintf(fi, ' Iteracion: %d Ultimo SNR: %12.10f PASO: %12.10f Errores: %d Total
            : %d Ult_Error= %12.10f', i-1, SNR_v(i-1), PASO, errores, total, ult_error)
            ;
```

```
206     fprintf(fi, '\n
            **************************************************************\n');
207     st = fclose(fi);
208
209 end
210 end
```

Listado B.1: Función para calcular la BER: "BER_calc.m"

```
1  %                                   REORDENA
2  % Esta funcion se encarga de calcular los vectores con la matriz ordenada
3  % si no se encuentra el archivo de configuracion "mat_config". Si el
4  % archivo
5  % existe, comprueba si corresponde a la matriz de chequeo de paridad que
6  % se esta usando. Si el archivo no existe, lo crea.
7  %
8  % EJEMPLO DE USO:
9  %
10 %             reordena(H);
11 %
12
13 function reordena(H)
14 igual=0;
15
16 % Comprueba si existe el archivo, si existe lo carga y compara la matriz
17 % de chequeo de paridad almacenada con la que se le pasa a la funcion.
18
19 archivo=exist('mat_config.mat','file');
20 if(archivo)
21     load('mat_config.mat');
22 end
23
24 if(exist('H_','var'))
25     igual=isequal(H, H_);
26 end
27
28 % Comprueba que existan las variables. Si existen las almacena y si no,
29 % las calcula.
30 if(exist('valores_', 'var') && exist('valores2_','var') && exist('posicionf_','
       var') && exist('posicionf2_','var') && exist('posicion_','var') && exist('
       posicion2_','var') && exist('H_','var') && igual==1)
31     fprintf(1, 'Se han encontrado los datos de la matriz\n')
32     valores=valores_;
33     valores2=valores2_;
34     posicion=posicion_;
35     posicion2=posicion2_;
36     posicionf=posicionf_;
```

```
37      posicionf2=posicionf2_;
38      save mat_config_temp.mat valores posicion posicionf valores2 posicion2
            posicionf2;
39  else
40      fprintf(1, 'No se han encontrado "mat_config.mat" sobre la matriz H.
            Generando...\n')
41      s=size(H);
42      [irow, jrow]=find(H==1);
43      [valores, posicion, posicionf]=agrupa_vec_c(irow', s(1));
44      [valores2, posicion2, posicionf2]=agrupa_vec_c(jrow', s(2));
45      if(archivo==0)
46          valores_=valores;
47          valores2_=valores2;
48          posicion_=posicion;
49          posicion2_=posicion2;
50          posicionf_=posicionf;
51          posicionf2_=posicionf2;
52          H_=H;
53          %Guarda una version del archivo para ella y otra para poder ser
54          %leido por los decodificadores (sin las _)
55          save mat_config.mat valores_ posicion_ posicionf_ valores2_ posicion2_
                posicionf2_ H_;
56          save mat_config_temp.mat valores posicion posicionf valores2 posicion2
                posicionf2;
57          fprintf(1,'Se ha creado el archivo "mat_config.mat" en el direc. de
                trabajo\n');
58      else
59          movefile('./mat_config.mat','~mat_config.mat')
60          valores_=valores;
61          valores2_=valores2;
62          posicion_=posicion;
63          posicion2_=posicion2;
64          posicionf_=posicionf;
65          posicionf2_=posicionf2;
66          H_=H;
67          %Guarda una version del archivo para ella y otra para poder ser leido por
                los
68          %decodificadores (sin las _)
69          save mat_config.mat valores_ posicion_ posicionf_ valores2_ posicion2_
                posicionf2_ H_;
70          save mat_config_temp.mat valores posicion posicionf valores2 posicion2
                posicionf2;
71          %El archivo antiguo se renombra en lugar de eliminarse
72          fprintf(1,'El archivo "mat_config.mat" existe pero no corresponde a H, se
                ha renombrado a "~mat_config"\n');
73      end
```

```
74      fprintf(1,'\n');
75  end
76  end
```

Listado B.2: Función para obtener los datos de la matriz: "reordena.m"

B.1.2 **Algoritmo Sumprod**

```
%
%Algoritmo sum product decoder. Toma como parametros de entrada
%la probabilidad de 1 estimada de la palabra recibida y la matriz
%de chequeo de paridad.
%Indica si se encontro la solucion en "sol" (sol=1) y la devuelve
%en "cest".
%
%A lo largo de la ejecucion se mostrara informacion sobre la
% ejecucion y una "barra de progreso" que indica que el algoritmo
%esta funcionando.
%
%Su uso es el siguiente:
%
% [cest, sol] = sumprod_dec(A, p_1);
%
%Se recomienda usar ";" al final de la llamada para no visualizar en
%resultado "cest" en pantalla ya que puede ser muy grande.
%


function [cest, sol]=sumprod_dec(A,p_1)

%Iniciamos el temporizador
c1=clock;

%Obtenemos los indices de los elementos distintos de 0 de la matriz A
[irow, jrow]=sparseorganizer(A);

%Creamos las variables que vamos a utilizar
L=50;
sol=0;              %Vale 0 si no se encuentra solucion y 1 si se encuentra
p_0=1-p_1;
s=size(A);          %Dimensiones de la matriz de chequeo

%Vectores usados para calcular los datos
Qmn_1=zeros(length(irow),1);
Qmn_0=zeros(length(irow),1);
Rmn_1=zeros(length(irow),1);
Rmn_0=zeros(length(irow),1);
deltaQmn=zeros(length(irow),1);
deltaRmn=zeros(length(irow),1);

%Almacenan los elementos de la matriz por filas y columnas guardando donde
%empiezan y acaban.
```

```
45 posicion=zeros(1,s(1));
46 posicionf=zeros(1,s(1));
47 posicion2=zeros(1,s(2));
48 posicionf2=zeros(1,s(2));
49 qn_0=zeros(1,s(2));
50 valores=zeros(1,length(irow));
51 valores2=zeros(1,length(irow));
52
53 %Numero de iteraciones
54
55 %Inicializamos para la primera iteracion
56 Qmn_1=p_1(jrow);
57 Qmn_0=p_0(jrow);
58
59 %Indexamos la matriz por filas y columnas (si no se ha hecho antes)
60 [valores, valores2, posicion, posicion2, posicionf, posicionf2] = reordena(A);
61
62
63 %Iteramos
64 for iteration=1:L
65 %*****************************************************************%
66     %************************************************%
67     %                  Paso horizontal               %
68     %************************************************%
69
70     %Calculamos deltaQmn
71     deltaQmn=Qmn_0-Qmn_1;
72
73     [deltaRmn]=hor_step(valores, posicion, posicionf, deltaQmn);
74
75     %Calculamos Rmn(0) y Rmn(1)
76
77     Rmn_1=((1-deltaRmn)/2);
78     Rmn_0=((1+deltaRmn)/2);
79
80     fprintf(1,'.');
81
82     %************************************************%
83     %                  Paso Vertical                 %
84     %************************************************%
85     %Calculamos los nuevos Qmn
86
87     [Qmn_0, Qmn_1, qn_0, qn_1]=ver_step(valores2, posicion2, posicionf2, Rmn_0,
           Rmn_1,p_0, p_1, jrow);
88
89     fprintf(1,'.');
```

```
90  %*********************************************************************%
91
92      %************************************************%
93      %             Comprobacion del resultado         %
94      %************************************************%
95
96      %Estimamos el vector recibido
97      cest=qn_1>=0.5;
98
99      %Comprobamos si se corresponde con algun vector correcto
100
101     if mod(A*cest',2)==0
102         sol=1;
103         break
104     end
105 end
106
107 fprintf(1,'\n');
108
109 %Paramos el temporizador
110 c2=clock;
111 time=c2-c1;
112
113 %Imprimimos los resultados del tiempo empleado y las iteraciones
114 if(sol==1)
115     fprintf(1,'Se ha encontrado solucion en %f minutos %f segundos con %d
            iteraciones.\n', time(5), time(6),iteration);
116 else
117     fprintf(1,'No se ha encontrado una posible solucion con %d iteraciones.\n',
            iteration);
118 end
119 end
```

Listado B.3: Código fuente del algoritmo Sum-Prod: "sumprod_dec.m"

B.1.3 **Algoritmo LLR**

```
%
%                                   LLR decoder
%
% Algoritmo log likelihood decoder. Toma como parametros de entrada el
% vector de senal recibido y la matriz de chequeo de paridad y la
% fiabilidad
% del canal. Indica si se encontro la solucion en "sol" (sol=1) y la
% devuelve en "cest".
%
%
% EJEMPLO DE USO:
%
%             [cest, sol] = loglikelihood_dec(A, r, Lc);
%


function [cest, sol] = LLR_dec(A, r, Lc)

%Iniciamos el temporizador (Se usa para calcular tiempos de ejecucion)
c1=clock;

%Obtenemos los indices de los elementos distintos de 0 de la matriz A

[irow, jrow]=find(A==1);
%Creamos las variables que vamos a utilizar
niterations=50;              %Numero de iteraciones
s=size(A);
sol=0;                       %Vale 0 si no se encuentra solucion y 1 si se
    encuentra


%Vectores usados para calcular los datos
NUmn=zeros(1, length(irow));
NUmn_t=zeros(1, length(irow));
lambdan=zeros(1,s(2));

%Inicializamos para la primera iteracion
lambdan=r*Lc;

%Cargamos los datos de la matriz generados por reordena.m
load mat_config_temp.mat

%Iteramos
for l=1:niterations
```

```
44
45 %************************************************************************%
46
47                 %************************************************%
48                 %                 Check Node Update              %
49                 %************************************************%
50
51
52     [NUmn]=check_node_LLR(valores, posicion, posicionf, lambdan, NUmn, jrow');
53
54
55                 %************************************************%
56                 %                  Bit Node Update               %
57                 %************************************************%
58
59     [lambdan]=bit_node(NUmn, posicion2, posicionf2, r, Lc);
60
61
62 %************************************************************************%
63
64                 %************************************************%
65                 %            Comprobacion del resultado          %
66                 %************************************************%
67
68     %Estimamos el vector recibido
69     cest=lambdan>0;
70
71     %Comprobamos si se corresponde con algun vector correcto
72     if mod(A*cest',2)==0
73         if not(isequal(cest, zeros(1, length(cest))))
74             sol=1;
75             break;
76         end
77     end
78
79 end
80
81 %Paramos el temporizador
82 c2=clock;
83 tiempo=c2-c1;
84
85
86 %Imprimimos los resultados del tiempo empleado y las iteraciones
87 if(sol==1)
88     [fi] = fopen('avance_LLR.txt','a');
```

```
89      fprintf(fi, 'Se ha encontrado solucion en %f minutos %f segundos con %d
            iteraciones. \n', tiempo(5), tiempo(6),l);
90      st=fclose(fi);
91  else
92      [fi] = fopen('avance_LLR.txt','a');
93      fprintf(fi, 'No se ha encontrado una posible solucion con %d iteraciones.\n',
            l);
94      st=fclose(fi);
95  end
96
97  end
```

Listado B.4: Código fuente del algoritmo LLR: "LLR_dec.m"

```
1  /*************************************************************************/
2  /*          Actualizacion de los nodos de chequeo del algoritmo LLR      */
3  /*************************************************************************/
4
5  #include "mex.h"
6  #include "stdio.h"
7  #include "math.h"
8
9  /* Llamada a la funcion en matlab:
10
11    [vectorsal]=check_node_LLR(vector_entrada, posin, posfin, lambdan, NUmn, jrow)
12
13 */
14
15 /* En esta funcion usamos las funciones hiperbolicas ideales              */
16
17
18 void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
19 {
20
21
22   /* Creamos vectores a todos los parametros de entrada y salida,
23      ademas de otras variables.                                          */
24
25   double *val_ent, *posIn, *posFin, *lambdan, *NUmn_ant, *NUmn, *jrow, ac;
26   int i, j, k;
27   int tam, tam_pos;
28
29   /* Asociamos lo vectores a los parametros de entrada y
30      salida de la funcion    */
31
32   val_ent = mxGetPr(prhs[0]);
33   tam = mxGetN(prhs[0]);
```

```
34   posIn = mxGetPr(prhs[1]);
35   posFin = mxGetPr(prhs[2]);
36   lambdan = mxGetPr(prhs[3]);
37   NUmn_ant = mxGetPr(prhs[4]);
38   jrow = mxGetPr(prhs[5]);
39   tam_pos=mxGetN(prhs[1]);
40   plhs[0] = mxCreateDoubleMatrix(1,tam, mxREAL);
41   NUmn= mxGetPr(plhs[0]);
42
43   /* Recorremos la matriz */
44
45   for (i=0; i<tam_pos; i++)
46     {
47
48       for(k=(int)posIn[i]; k<=(int)posFin[i]; k++)
49         {
50           ac=1;
51
52           for (j=(int)posIn[i]; j<=(int)posFin[i]; j++)
53             {
54               /* Para todos los elementos menos el indicado (eliminamos el valor
55                  enviado en la iteracion previa) vamos acumulando el producto. */
56               if(j!=k)
57                 {
58
59                   ac=ac*(tanh(-(lambdan[(int)jrow[(int)val_ent[j-1]-1]-1]-NUmn_ant
                         [(int)val_ent[j-1]-1])/2));
60                 }
61             }
62           /* Almacenamos el valor acumulado */
63           NUmn[(int)val_ent[k-1]-1]=-2*atanh(ac);
64
65         }
66     }
67   return;
68 }
```

Listado B.5: Código fuente del la actualización de los nodos de chequeo LLR: "check_node_LLR.c"

```
1  /*************************************************************************/
2  /*                        Actualizacion de los nodos de bit              */
3  /*************************************************************************/
4  #include "mex.h"
5  #include "stdio.h"
6  #include "math.h"
7  /* Llamada a la funcion en matlab:
8
```

```
9     [lambdan]=bit_node(NUmn, posin, posfin, r, Lc)
10
11  */
12
13  void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
14  {
15    /* Creamos vectores a todos los parametros de entrada y salida, ademas de otras
16       variables tipo double. */
17    double *posIn, *posFin, *lambdan, *NUmn, *r, suma;
18    int i,j, Lc;
19    int tam_pos;
20
21    /* Asociamos lo vectores a los parametros de entrada y salida de la funcion */
22    NUmn = mxGetPr(prhs[0]);
23    posIn = mxGetPr(prhs[1]);
24    posFin = mxGetPr(prhs[2]);
25    tam_pos=mxGetN(prhs[1]);
26    r= mxGetPr(prhs[3]);
27    Lc=(int)(mxGetScalar(prhs[4]));
28    plhs[0] = mxCreateDoubleMatrix(1,tam_pos, mxREAL);
29    lambdan= mxGetPr(plhs[0]);
30
31    /* Vamos sumando los elementos de las columnas (La infromacion de todos los
          nodos de
32       chequeo para cada nodo de bit) */
33    for (i=0; i<tam_pos; i++)
34      {
35        suma=0;
36        for(j=(int)posIn[i]; j<=(int)posFin[i]; j++)
37          {
38            suma=suma+NUmn[j-1];
39          }
40        /* Guardamos la suma agregandole el valor correspondiente de r
41           (se se suma la probabilidad del nodo de bit)          */
42        lambdan[i]=suma+r[i]*Lc;
43      }
44    return;
45  }
```

Listado B.6: Código fuente del la actualización de los nodos de bit: "bit_node.c"

B.1.4 **Algoritmo Minsum**

```
%
%                                   Minsum decoder
%
% Algoritmo minimum log likelihood decoder. Toma como parametros de
% entrada el vector de senal recibido, la matriz de chequeo de paridad
% y la fiabilidad del canal. Indica si se encontro la solucion en "sol"
% (sol=1) y la devuelve en "cest".
%
% EJEMPLO DE USO:
%
%             [cest, sol] = Minsum_dec(A, r);
%



function [cest, sol] = Minsum_dec(A, r, Lc)

%Iniciamos el temporizador
c1=clock;

%Obtenemos los indices de los elementos distintos de 0 de la matriz A
[irow, jrow]=find(A==1);

%Creamos las variables que vamos a utilizar

niterations=50;       %Numero de iteraciones
s=size(A);
sol=0;                %Vale 0 si no se encuentra solucion y 1 si se encuentra

%Vectores usados para calcular los datos

lambdan=zeros(1,s(2));
NUmn=zeros(1, length(irow));


%Inicializamos para la primera iteracion
lambdan=r*Lc;

%Cargamos los datos de la matriz generados por reordena.m
load mat_config_temp.mat

%Iteramos
for l=1:niterations

```

```
%*************************************************************************%
                %*************************************************%
                %                 Check Node Update               %
                %*************************************************%

    [NUmn]=check_node_MS(valores, posicion, posicionf, lambdan, NUmn, jrow');


                %*************************************************%
                %                  Bit Node Update                %
                %*************************************************%

    [lambdan]=bit_node(NUmn, posicion2, posicionf2, r, Lc);



%*************************************************************************%

                %*************************************************%
                %              Comprobacion del resultado         %
                %*************************************************%

    %Estimamos el vector recibido
    cest=lambdan>0;

    %Comprobamos si se corresponde con algun vector correcto

    if mod(A*cest',2)==0
        if not(isequal(cest, zeros(1, length(cest))))
            sol=1;
            break;
        end
    end
end


%Paramos el temporizador
c2=clock;
tiempo=c2-c1;


%Imprimimos los resultados del tiempo empleado y las iteraciones
if(sol==1)
    [fi] = fopen('avance_MS.txt','a');
    fprintf(fi, 'Se ha encontrado solucion en %f minutos %f segundos con %d
        iteraciones. \n', tiempo(5), tiempo(6),l);
```

```
90      st=fclose(fi);
91  else
92      [fi] = fopen('avance_MS.txt','a');
93      fprintf(fi, 'No se ha encontrado una posible solucion con %d iteraciones.\n',
            l);
94      st=fclose(fi);
95  end
96  end
```

Listado B.7: Código fuente del algoritmo Minsum: "Minsum_dec.m"

```
1   /*********************************************************************/
2   /*          Actualizacion de los nodos de chequeo del algoritmo MS     */
3   /*********************************************************************/
4
5   #include "mex.h"
6   #include "stdio.h"
7   #include "math.h"
8
9   /* Llamada a la funcion en matlab:
10
11    [vectorsal]=check_node(vector_entrada, posin, posfin, lambdan, NUmn, jrow)
12
13  */
14
15  /* En esta funcion usamos una simplificacion de la regla de la tanh       */
16
17  void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
18  {
19    /* Creamos vectores a todos los parametros de entrada y salida, ademas de otras
20       variables.                                                            */
21
22    double *val_ent,*posIn, *posFin, *lambdan,*NUmn_ant,*NUmn,*jrow, minimo,
          minimo_p;
23    int i,j,k, contk, signo_p, signo, valor;
24    int tam,tam_pos, inicio, fin;
25
26    /* Asociamos lo vectores a los parametros de entrada y salida de la funcion */
27    val_ent = mxGetPr(prhs[0]);
28    tam = mxGetN(prhs[0]);
29    posIn = mxGetPr(prhs[1]);
30    posFin = mxGetPr(prhs[2]);
31    lambdan = mxGetPr(prhs[3]);
32    NUmn_ant = mxGetPr(prhs[4]);
33    jrow = mxGetPr(prhs[5]);
34    tam_pos=mxGetN(prhs[1]);
35    plhs[0] = mxCreateDoubleMatrix(1,tam, mxREAL);
```

```
36  NUmn= mxGetPr(plhs[0]);
37
38  /* Recorremos la matriz */
39  for (i=0; i<tam_pos; i++)
40    {
41      for(k=(int)posIn[i]; k<=(int)posFin[i]; k++)
42        {
43          signo=1;
44          minimo=-1;
45          minimo_p=0;
46
47          /* Vamos sumando los elementos de las filas */
48          for (j=(int)posIn[i]; j<=(int)posFin[i]; j++)
49            {
50              /* Para todos los elementos menos el indicado (eliminamos la
51                 contribucion del mensaje enviado en la iteracion anterior)
52                 calculamos el minimo de la funcion y los signos           */
53
54              if(j!=k)
55                {
56                  minimo_p=fabs(-1*(lambdan[(int)jrow[(int)val_ent[j-1]-1]-1]-
                        NUmn_ant[(int)val_ent[j-1]-1]));
57                  signo_p=(int)(-1*(lambdan[(int)jrow[(int)val_ent[j-1]-1]-1]-
                        NUmn_ant[(int)val_ent[j-1]-1])/minimo_p);
58                  signo=signo_p*signo;
59                  if(minimo_p<minimo || minimo==-1)
60                    {
61                      minimo=minimo_p;
62                    }
63                }
64            }
65          /* Almacenamos el valor                                          */
66          NUmn[(int)val_ent[k-1]-1]=-1*minimo*signo;
67        }
68    }
69
70  return;
71  }
```

Listado B.8: Código fuente del la actualización de los nodos de chequeo MS truncada: "check_node_MS.c"

B.1.5 **Algoritmo LLR con tanh truncada**

```
%                          Algoritmo LLR con tanh truncada
%
% Este algoritmo funciona como el LLR pero en lugar de usar la tanh
% ideal se usa una modificacion que trunca su argumento a un maximo.
% En nuestro caso a |x|=7.
%
% EJEMPLO DE USO:
%
%             [cest, sol] = LLR_trunc_dec(A, r, Lc);
%


function [cest, sol] = LLR_trunc_dec(A, r, Lc)

%Iniciamos el temporizador

c1=clock;

%Obtenemos los indices de los elementos distintos de 0 de la matriz A

[irow, jrow]=find(A==1);

%Creamos las variables que vamos a utilizar

niterations=50;       %Numero de iteraciones
s=size(A);
sol=0;                %Vale 0 si no se encuentra solucion y 1 si se encuentra

%Vectores usados para calcular los datos

NUmn=zeros(1, length(irow));
NUmn_t=zeros(1, length(irow));
lambdan=zeros(1,s(2));

%Inicializamos para la primera iteracion
lambdan=r*Lc;

%Cargamos los datos de la matriz generados por reordena.m
load mat_config_temp.mat

%Iteramos
for l=1:niterations

%*************************************************************************%
```

```
45
46              %************************************************%
47              %                  Check Node Update             %
48              %************************************************%
49
50      [NUmn]=check_node_LLRtrunc(valores, posicion, posicionf, lambdan, NUmn, jrow
            ');
51
52              %************************************************%
53              %                  Bit Node Update               %
54              %************************************************%
55
56      [lambdan]=bit_node(NUmn, posicion2, posicionf2, r, Lc);
57
58  %*************************************************************************%
59
60
61              %************************************************%
62              %           Comprobacion del resultado           %
63              %************************************************%
64
65      %Estimamos el vector recibido
66      cest=lambdan>0;
67
68      %Comprobamos si se corresponde con algun vector correcto
69      if mod(A*cest',2)==0
70          if not(isequal(cest, zeros(1, length(cest))))
71              sol=1;
72              break;
73          end
74      end
75
76  end
77
78  %Paramos el temporizador
79  c2=clock;
80  tiempo=c2-c1;
81  %Imprimimos los resultados del tiempo empleado y las iteraciones
82  if(sol==1)
83      [fi] = fopen('avance_LLR_trunc.txt','a');
84      fprintf(fi, 'Se ha encontrado solucion en %f minutos %f segundos con %d
            iteraciones. \n', tiempo(5), tiempo(6),l);
85      st=fclose(fi);
86  else
87      [fi] = fopen('avance_LLR_trunc.txt','a');
```

```
88      fprintf(fi, 'No se ha encontrado una posible solucion con %d iteraciones.\n',
            l);
89      st=fclose(fi);
90  end
91
92  end
```

Listado B.9: Código fuente del algoritmo LLR con tanh truncada: "LLR_trunc_dec.m"

```
1   /*********************************************************************************/
2   /* Actualizacion de los nodos de chequeo del algoritmo LLR con tanh truncada */
3   /*********************************************************************************/
4   #include "mex.h"
5   #include "stdio.h"
6   #include "math.h"
7
8   /* Llamada a la funcion en matlab:
9
10    [vectorsal]=check_node_LLRtrunc(vector_entrada, posin, posfin, lambdan, NUmn,
          jrow)
11
12  */
13
14  /* En esta funcion usamos una version de la tanh con el argumento truncado a |x
        |=7 */
15
16  /* Declaramos las nuevas funciones tanh_trunc y atanh_trunc                */
17  double tanh_trunc(double ind);
18  double atanh_trunc(double ac);
19
20  void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
21  {
22    /* Creamos vectores a todos los parametros de entrada y salida,
23       ademas de otras variables.                                          */
24    double *val_ent,*posIn, *posFin, *lambdan,*NUmn_ant,*NUmn,*jrow,ac;
25    int i,j,k, contk, signo_p, signo, valor;
26    int tam,tam_pos, inicio, fin;
27
28    /* Asociamos lo vectores a los parametros de entrada y salida de la funcion */
29    val_ent = mxGetPr(prhs[0]);
30    tam = mxGetN(prhs[0]);
31    posIn = mxGetPr(prhs[1]);
32    posFin = mxGetPr(prhs[2]);
33    lambdan = mxGetPr(prhs[3]);
34    NUmn_ant = mxGetPr(prhs[4]);
35    jrow = mxGetPr(prhs[5]);
36    tam_pos=mxGetN(prhs[1]);
```

```
37    plhs[0] = mxCreateDoubleMatrix(1,tam, mxREAL);
38    NUmn= mxGetPr(plhs[0]);
39
40    /* Recorremos la matriz                                              */
41    for (i=0; i<tam_pos; i++)
42      {
43
44        for(k=(int)posIn[i]; k<=(int)posFin[i]; k++)
45          {
46            ac=1;
47
48            for (j=(int)posIn[i]; j<=(int)posFin[i]; j++)
49              {
50                /* Para todos los elementos menos el indicado (eliminamos el valor
51                   enviado en la iteracion previa) vamos acumulando el producto. */
52                if(j!=k)
53                  {
54
55                    ac=ac*(tanh_trunc(-(lambdan[(int)jrow[(int)val_ent[j-1]-1]-1]-
                          NUmn_ant[(int)val_ent[j-1]-1])/2));
56                  }
57              }
58            /* Almacenamos el valor                                        */
59            NUmn[(int)val_ent[k-1]-1]=-2*atanh(ac);
60          }
61      }
62
63    return;
64  }
65
66  double tanh_trunc(double ind)
67  {
68    double tang_t;
69    double signo;
70    double modulo;
71    modulo=fabs(ind);
72    signo=modulo/ind;
73
74    /* Si el argumento en valor absoluto supera un cierto valor, lo truncamos */
75    if(modulo>7)
76      {
77        tang_t=tanh(7)*signo;
78      }
79    else
80      {
81        tang_t=tanh(ind);
```

```
82      }
83    return tang_t;
84  }
85
86
87  double atanh_trunc(double ac)
88  {
89    double atang_t;
90    double signo;
91    double modulo;
92    modulo=fabs(ac);
93    signo=modulo/ac;
94    /* Si el argumento en valor absoluto supera un cierto valor, lo truncamos */
95    if(modulo>7)
96      {
97        atang_t=atanh(modulo)*signo;
98      }
99    else
100     {
101       atang_t=atanh(ac);
102     }
103   return atang_t;
104 }
```

Listado B.10: Código fuente del la actualización de los nodos de chequeo con tanh truncada: "check_node_LLRtrunc.c"

B.1.6 **Algoritmo LLR con aproximación lineal de la tanh**

```
%
%                Algoritmo LLR con aproximacion lineal de la tanh
%
% Este algoritmo funciona como el LLR pero en lugar de usar la tanh ideal
% se usa una aproximacion lineal a trozos.
%
% EJEMPLO DE USO:
%
%             [cest, sol] = LLR_PW_dec(A, r, Lc);
%

function [cest, sol] = LLR_PW_dec(A, r, Lc)

%Iniciamos el temporizador
c1=clock;

%Obtenemos los indices de los elementos distintos de 0 de la matriz A

[irow, jrow]=find(A==1);

%Creamos las variables que vamos a utilizar
niterations=50;     %Numero de iteraciones
s=size(A);
sol=0;              %Vale 0 si no se encuentra solucion y 1 si se encuentra

%Vectores usados para calcular los datos
NUmn=zeros(1, length(irow));
NUmn_t=zeros(1, length(irow));
lambdan=zeros(1,s(2));

%Inicializamos para la primera iteracion
lambdan=r*Lc;

%Cargamos los datos de la matriz generados por reordena.m
load mat_config_temp.mat

%Iteramos
for l=1:niterations
%*************************************************************************%

                  %***********************************************%
                  %                  Check Node Update            %
                  %***********************************************%

```

```
45      [NUmn]=check_node_PW(valores, posicion, posicionf, lambdan, NUmn, jrow');
46
47
48                  %************************************************%
49                  %                 Bit Node Update                %
50                  %************************************************%
51
52      [lambdan]=bit_node(NUmn, posicion2, posicionf2, r, Lc);
53
54
55  %*****************************************************************************%
56
57                  %************************************************%
58                  %           Comprobacion del resultado           %
59                  %************************************************%
60
61      %Estimamos el vector recibido
62      cest=lambdan>0;
63
64      %Comprobamos si se corresponde con algun vector correcto
65      if mod(A*cest',2)==0
66          if not(isequal(cest, zeros(1, length(cest))))
67              sol=1;
68              break;
69          end
70      end
71
72  end
73
74  %Paramos el temporizador
75  c2=clock;
76  tiempo=c2-c1;
77
78  %Imprimimos los resultados del tiempo empleado y las iteraciones
79  if(sol==1)
80      [fi] = fopen('avance_LLR_PW.txt','a');
81      fprintf(fi, 'Se ha encontrado solucion en %f minutos %f segundos con %d
            iteraciones. \n', tiempo(5), tiempo(6),l);
82      st=fclose(fi);
83  else
84      [fi] = fopen('avance_LLR_PW.txt','a');
85      fprintf(fi, 'No se ha encontrado una posible solucion con %d iteraciones.\n',
             l);
86      st=fclose(fi);
87  end
88  end
```

Listado B.11: Código fuente del algoritmo LLR con aprox. lineal de la tanh: "LLR_PW_dec.m"

```
/*************************************************************************/
/*Actualizacion de los nodos de chequeo del algoritmo LLR con aprox lineal de la
    tanh*/
/*************************************************************************/

#include "mex.h"
#include "stdio.h"
#include "math.h"

/* Llamada a la funcion en matlab:

   [vectorsal]=check_node_PW(vector_entrada, posin, posfin, lambdan, NUmn, jrow)

 */

/* En esta funcion usamos una aproximacion lineal a trozos de la tanh          */

/* Declaramos las nuevas funciones tanh_trunc y atanh_trunc                    */
double tanh_aprox(double ind);
double atanh_aprox(double ac);
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    /* Creamos vectores a todos los parametros de entrada y salida,
       ademas de otras variables.                                              */
    double *val_ent,*posIn, *posFin, *lambdan,*NUmn_ant,*NUmn,*jrow,ac, ind;
    int i,j,k, contk, signo_p, signo, valor;
    int tam,tam_pos;

    /* Asociamos lo vectores a los parametros de entrada y salida de la funcion
        */
    val_ent = mxGetPr(prhs[0]);
    tam = mxGetN(prhs[0]);
    posIn = mxGetPr(prhs[1]);
    posFin = mxGetPr(prhs[2]);
    lambdan = mxGetPr(prhs[3]);
    NUmn_ant = mxGetPr(prhs[4]);
    jrow = mxGetPr(prhs[5]);
    tam_pos=mxGetN(prhs[1]);
    plhs[0] = mxCreateDoubleMatrix(1,tam, mxREAL);
    NUmn= mxGetPr(plhs[0]);
    /* Recorremos la matriz                                                   */
    for (i=0; i<tam_pos; i++)
    {
```

```
42          for(k=(int)posIn[i]; k<=(int)posFin[i]; k++)
43          {
44              ac=1;
45              for (j=(int)posIn[i]; j<=(int)posFin[i]; j++)
46              {
47                  /* Para todos los elementos menos el indicado (eliminamos el valor
48                     enviado en la iteracion previa) vamos acumulando el producto. */
49                  if(j!=k)
50                  {
51                      ind=-(lambdan[(int)jrow[(int)val_ent[j-1]-1]-1]-NUmn_ant[(int)
                            val_ent[j-1]-1])/2;
52                      ac=ac*tanh_aprox(ind);
53                  }
54              }
55              /* Almacenamos el valor
                                                                    */
56              NUmn[(int)val_ent[k-1]-1]=-2*atanh_aprox(ac);
57          }
58      }
59      return;
60  }
61
62  double tanh_aprox(double ind)
63  {
64      double tang_aprox;
65      if(ind<=-7)
66      {
67      tang_aprox=-0.9998;
68      }
69      if(-7<ind && ind<=-3)
70      {
71          tang_aprox=0.0012*ind-0.9914;
72      }
73      if(-3<ind && ind<=-1.6)
74      {
75          tang_aprox=0.0524*ind -0.8378;
76      }
77      if(-1.6<ind && ind<=-0.8)
78      {
79          tang_aprox=0.322*ind -0.4064;
80      }
81      if(-0.8<ind && ind<=0.8)
82      {
83          tang_aprox=0.83*ind;
84      }
85      if(0.8<ind && ind<=1.6)
```

```
86      {
87          tang_aprox=0.322*ind +0.4064;
88      }
89      if(1.6<ind && ind<=3)
90      {
91          tang_aprox=0.0524*ind +0.8378;
92      }
93      if(3<ind && ind<7)
94      {
95          tang_aprox=0.0012*ind +0.9914;
96      }
97      if(ind>=7)
98      {
99      tang_aprox=0.9998;
100     }
101     return tang_aprox;
102 }
103
104 double atanh_aprox(double ac)
105 {
106     double atan_aprox;
107     if(ac<=-0.999998)
108     {
109     atan_aprox=-7.165;
110     }
111     if (-0.999998<ac && ac<=-0.9951)
112     {
113         atan_aprox=(ac+0.9914)/0.0012;
114     }
115     if (-0.9951<ac && ac<=-0.9217)
116     {
117         atan_aprox=(ac+0.8378)/0.0524;
118     }
119     if(-0.9217<ac && ac<=-0.6640)
120     {
121         atan_aprox=(ac+0.4064)/0.322;
122     }
123     if(-0.6640<ac && ac<=0.6640)
124     {
125         atan_aprox=ac/0.83;
126     }
127     if(0.6640<ac && ac<=0.9217)
128     {
129         atan_aprox=(ac-0.4064)/0.322;
130     }
131     if(0.9217<ac && ac<=0.9951)
```

```
132     {
133         atan_aprox=(ac-0.8378)/0.0524;
134     }
135     if(ac>0.9951 && ac<=0.999998)
136     {
137         atan_aprox=(ac-0.9914)/0.0012;
138     }
139       if(ac>0.999998)
140     {
141     atan_aprox=7.165;
142     }
143
144     return atan_aprox;
145 }
```

Listado B.12: Código fuente del la actualización de los nodos de chequeo con aprox. lineal de la tanh: "check_node_PW.c"

B.1.7 Algoritmo LLR con aproximación constante de la tanh

```
%
%                Algoritmo LLR con aproximacion constante de la tanh
%
% Este algoritmo funciona como el LLR pero en lugar de usar la tanh ideal
% se usa una aproximacion constante a trozos.
%
% EJEMPLO DE USO:
%
%             [cest, sol] = LLR_PW_dec(A, r, Lc);
%

function [cest, sol] = LLR_LUT_dec(A, r, Lc)

%Iniciamos el temporizador

c1=clock;

%Obtenemos los indices de los elementos distintos de 0 de la matriz A

[irow, jrow]=find(A==1);

%Creamos las variables que vamos a utilizar

niterations=50;      %Numero de iteraciones
s=size(A);
sol=0;               %Vale 0 si no se encuentra solucion y 1 si se encuentra

%Vectores usados para calcular los datos

NUmn=zeros(1, length(irow));
NUmn_t=zeros(1, length(irow));
lambdan=zeros(1,s(2));


%Inicializamos para la primera iteracion

lambdan=r*Lc;

%Cargamos los datos de la matriz generados por reordena.m

load mat_config_temp.mat

%Iteramos
for l=1:niterations
```

```
45
46 %*************************************************************************%
47
48                 %************************************************%
49                 %                  Check Node Update             %
50                 %************************************************%
51
52     [NUmn]=check_node_LUT(valores, posicion, posicionf, lambdan, NUmn, jrow');
53
54                 %************************************************%
55                 %                  Bit Node Update               %
56                 %************************************************%
57
58     [lambdan]=bit_node(NUmn, posicion2, posicionf2, r, Lc);
59
60
61
62 %*************************************************************************%
63
64                 %************************************************%
65                 %           Comprobacion del resultado           %
66                 %************************************************%
67     %Estimamos el vector recibido
68     cest=lambdan>0;
69
70     %Comprobamos si se corresponde con algun vector correcto
71    if mod(A*cest',2)==0
72         if not(isequal(cest, zeros(1, length(cest))))
73             sol=1;
74             break;
75         end
76     end
77
78 end
79 %fprintf(1, '\n');
80 %Paramos el temporizador
81 c2=clock;
82 tiempo=c2-c1;
83
84 %Imprimimos los resultados del tiempo empleado y las iteraciones
85 if(sol==1)
86     [fi] = fopen('avance_LLR_LUT.txt','a');
87     fprintf(fi, 'Se ha encontrado solucion en %f minutos %f segundos con %d
           iteraciones. \n', tiempo(5), tiempo(6),l);
88     st=fclose(fi);
89 else
```

```
90      [fi] = fopen('avance_LLR_LUT.txt','a');
91      fprintf(fi, 'No se ha encontrado una posible solucion con %d iteraciones.\n',
            l);
92      st=fclose(fi);
93  end
94
95  end
```

Listado B.13: Código fuente del algoritmo LLR con aprox. constante de la tanh: "LLR_LUT_dec.m"

```
1   /*********************************************************************************
        */
2   /*   Actualizacion de los nodos de chequeo del algoritmo LLR con aprox constante
        de la tanh */
3   /*********************************************************************************
        */
4
5   #include "mex.h"
6   #include "stdio.h"
7   #include "math.h"
8
9   /* Llamada a la funcion en matlab:
10
11     [vectorsal]=check_node_LUT(vector_entrada, posin, posfin, lambdan, NUmn, jrow)
12
13   */
14
15  /* En esta funcion usamos una aproximacion constante a trozos de la tanh c      */
16
17  /* Declaramos las nuevas funciones tanh_trunc y atanh_trunc                      */
18  double tanh_LUT(double ind);
19  double atanh_LUT(double ac);
20
21  void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
22  {
23      /* Creamos vectores a todos los parametros de entrada y salida,
24         ademas de otras variables.                                                 */
25      double f, f2, *val_ent,*posIn, *posFin, *lambdan,*NUmn_ant,*NUmn,*jrow,ac,
            ind;
26      int i,j,k, contk, signo_p, signo, valor, w;
27      int tam,tam_pos;
28
29      /* Asociamos lo vectores a los parametros de entrada y salida de la funcion
            */
30      val_ent = mxGetPr(prhs[0]);
31      tam = mxGetN(prhs[0]);
32      posIn = mxGetPr(prhs[1]);
```

```
33      posFin = mxGetPr(prhs[2]);
34      lambdan = mxGetPr(prhs[3]);
35      NUmn_ant = mxGetPr(prhs[4]);
36      jrow = mxGetPr(prhs[5]);
37      tam_pos=mxGetN(prhs[1]);
38      plhs[0] = mxCreateDoubleMatrix(1,tam, mxREAL);
39      NUmn= mxGetPr(plhs[0]);
40
41
42      /* Recorremos la matriz                                                    */
43      for (i=0; i<tam_pos; i++)
44      {
45
46          for(k=(int)posIn[i]; k<=(int)posFin[i]; k++)
47          {
48              ac=1;
49
50              for (j=(int)posIn[i]; j<=(int)posFin[i]; j++)
51              {
52                  /* Para todos los elementos menos el indicado (eliminamos el valor
53                     enviado en la iteracion previa) vamos acumulando el producto. */
54                  if(j!=k)
55                  {
56                      ind=-(lambdan[(int)jrow[(int)val_ent[j-1]-1]-1]-NUmn_ant[(int)
                            val_ent[j-1]-1])/2;
57
58                      ac=ac*tanh_LUT(ind);
59                  }
60              }
61              /* Almacenamos el valor
                                                                        */
62              NUmn[(int)val_ent[k-1]-1]=-2*atanh_LUT(ac);
63          }
64      }
65
66      return;
67  }
68
69
70  double tanh_LUT(double ind)
71  {
72      double tang_aprox;
73      if(ind<=-7)
74      {
75          tang_aprox=-0.99991;
76      }
```

```
77      if(-7<ind && ind<=-3)
78      {
79          tang_aprox=-0.99991;
80      }
81      if(-3<ind && ind<=-1.6)
82      {
83          tang_aprox=-0.9801;
84      }
85      if(-1.6<ind && ind<=-0.8)
86      {
87          tang_aprox=-0.8337;
88      }
89      if(-0.8<ind && ind<=0)
90      {
91          tang_aprox=-0.3799;
92      }
93      if(ind>0 && ind<=0.8)
94      {
95          tang_aprox=0.3799;
96      }
97      if(0.8<ind && ind<=1.6)
98      {
99          tang_aprox=0.8337;
100     }
101     if(1.6<ind && ind<=3)
102     {
103         tang_aprox=0.9801;
104     }
105     if(3<ind && ind<7)
106     {
107         tang_aprox=0.99991;
108     }
109     if(ind>=7)
110     {
111         tang_aprox=0.99991;
112     }
113     return tang_aprox;
114 }
115
116
117 double atanh_LUT(double ac)
118 {
119     double atan_aprox;
120     if(ac<=-0.999998)
121     {
122         atan_aprox=-3.3516;
```

```
123     }
124     if (-0.999998<ac && ac<=-0.9951)
125     {
126         atan_aprox=-3.3516;
127     }
128     if (-0.9951<ac && ac<=-0.9217)
129     {
130         atan_aprox=-1.9259;
131     }
132     if(-0.9217<ac && ac<=-0.6640)
133     {
134         atan_aprox=-1.0791;
135     }
136     if(-0.6640<ac && ac<=0)
137     {
138         atan_aprox=-0.3451;
139     }
140
141
142     if(0<ac && ac<=0.6640)
143     {
144         atan_aprox=0.3451;
145     }
146     if(0.6640<ac && ac<=0.9217)
147     {
148         atan_aprox=1.0791;
149     }
150     if(0.9217<ac && ac<=0.9951)
151     {
152         atan_aprox=1.9259;
153     }
154     if(0.9951<ac && ac<=0.999998)
155     {
156         atan_aprox=3.3516;
157     }
158     if(ac>0.999998)
159     {
160        atan_aprox=3.3516;
161     }
162
163     return atan_aprox;
164 }
```

Listado B.14: Código fuente del la actualización de los nodos de chequeo con aprox. constante de la tanh: "check_node_LUT.c"

B.1.8 **Algoritmo basado en el enfoque de Gallager con función fi simplificada**

```
%             Aproximacion del algoritmo basado en el enfoque de Galleger
%
% En este algoritmo se usa una simplificacion de la funcion de Gallager
% para facilitar la implementacion en hardware del algoritmo
%
% EJEMPLO DE USO:
%
%             [cest, sol] = Fi_func_dec(A, r, Lc);
%

function [cest, sol] = LLR_trunc_dec(A, r, Lc)

%Iniciamos el temporizador

c1=clock;

%Obtenemos los indices de los elementos distintos de 0 de la matriz A

[irow, jrow]=find(A==1);

%Creamos las variables que vamos a utilizar

niterations=50;      %Numero de iteraciones
s=size(A);
sol=0;               %Vale 0 si no se encuentra solucion y 1 si se encuentra

%Vectores usados para calcular los datos
NUmn=zeros(1, length(irow));
NUmn_t=zeros(1, length(irow));
lambdan=zeros(1,s(2));

%Inicializamos para la primera iteracion
lambdan=r*Lc;

%Cargamos los datos de la matriz generados por reordena.m
load mat_config_temp.mat

%Iteramos
for l=1:niterations

%*************************************************************************%

              %*********************************************%
              %                  Check Node Update          %
```

```
45              %************************************************%
46
47      [NUmn]=check_node_FI(valores, posicion, posicionf, lambdan, NUmn, jrow');
48
49              %************************************************%
50              %                  Bit Node Update               %
51              %************************************************%
52
53      [lambdan]=bit_node(NUmn, posicion2, posicionf2, r, Lc);
54
55  %*************************************************************************%
56
57              %************************************************%
58              %           Comprobacion del resultado           %
59              %************************************************%
60
61      %Estimamos el vector recibido
62      cest=lambdan>0;
63
64      %Comprobamos si se corresponde con algun vector correcto
65      if mod(A*cest',2)==0
66          if not(isequal(cest, zeros(1, length(cest))))
67              sol=1;
68              break;
69          end
70      end
71
72  end
73
74
75  %Paramos el temporizador
76  c2=clock;
77  tiempo=c2-c1;
78
79
80  %Imprimimos los resultados del tiempo empleado y las iteraciones
81  if(sol==1)
82      [fi] = fopen('avance_LLR_trunc.txt','a');
83      fprintf(fi, 'Se ha encontrado solucion en %f minutos %f segundos con %d
            iteraciones. \n', tiempo(5), tiempo(6),l);
84      st=fclose(fi);
85  else
86      [fi] = fopen('avance_LLR_trunc.txt','a');
87      fprintf(fi, 'No se ha encontrado una posible solucion con %d iteraciones.\n',
             l);
88      st=fclose(fi);
```

```
89 end
90
91 end
```

Listado B.15: Código fuente del algoritmo basado en el enfoque de Gallager con función fi simplificada: "Fi_func_dec.m"

```
1  /*************************************************************************/
2  /* Actualizacion de los nodos de chequeo del enfoque de Gallager con una funcion */
3  /* simplificada                                                          */
4  /*************************************************************************/
5
6  #include "mex.h"
7  #include "stdio.h"
8  #include "math.h"
9
10 /* Llamada a la funcion en matlab:
11
12    [vectorsal]=check_node_FI(vector_entrada, posin, posfin, lambdan, NUmn, jrow)
13
14  */
15 /* En esta funcion usamos una ecuacion simplificada de la funcion de Gallager */
16
17 /* Definimos la nueva funcion de la ecuacion de Gallager                 */
18 double funcion_fi(double r);
19
20 void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
21 {
22     /* Creamos vectores a todos los parametros de entrada y salida,
23        ademas de otras variables.                                        */
24     double f, f2, *val_ent,*posIn, *posFin, *lambdan,*NUmn_ant,*NUmn,*jrow,ac,
           ind;
25     int i,j,k, contk, signo_p, signo, valor, w;
26     int tam,tam_pos;
27
28     /* Asociamos lo vectores a los parametros de entrada y salida de la funcion
           */
29     val_ent = mxGetPr(prhs[0]);
30     tam = mxGetN(prhs[0]);
31     posIn = mxGetPr(prhs[1]);
32     posFin = mxGetPr(prhs[2]);
33     lambdan = mxGetPr(prhs[3]);
34     NUmn_ant = mxGetPr(prhs[4]);
35     jrow = mxGetPr(prhs[5]);
36     tam_pos=mxGetN(prhs[1]);
37     plhs[0] = mxCreateDoubleMatrix(1,tam, mxREAL);
38     NUmn= mxGetPr(plhs[0]);
```

```
39
40
41      /* Recorremos la matriz                                          */
42      for (i=0; i<tam_pos; i++)
43      {
44
45          for(k=(int)posIn[i]; k<=(int)posFin[i]; k++)
46          {
47              ac=1;
48
49              for (j=(int)posIn[i]; j<=(int)posFin[i]; j++)
50              {
51                  /* Para todos los elementos menos el indicado (eliminamos el valor
52                     enviado en la iteracion previa) vamos acumulando el producto. */
53                  if(j!=k)
54                  {
55                      ind=-(lambdan[(int)jrow[(int)val_ent[j-1]-1]-1]-NUmn_ant[(int)
                            val_ent[j-1]-1])/2;
56
57                      ac=ac*funcion_fi(ind);
58                  }
59              }
60              /* Almacenamos el valor
                                                                        */
61              NUmn[(int)val_ent[k-1]-1]=-2*funcion_fi(ac);
62          }
63      }
64
65      return;
66  }
67
68  double funcion_fi(double r)
69  {
70      double decimal, entero, fi, r2;
71      r2=fabs(r)/log(2);
72      entero=floor(r2);
73      decimal=r-entero;
74  /*  Ecuacion exacta (Elegir solo una de las dos)                      */
75  /*  fi=log((1+pow(2,(-1*r2)))/(1-pow(2,(-1*r2))));                   */
76  /*  Ecuacion Simplificada (Elegir solo una de las dos)                */
77      fi=log((1+(1-decimal/2)*pow(2,-1*entero))/(1-(1-decimal/2)*pow(2,-1*entero)))
            ;
78      return fi;
79  }
```

Listado B.16: Código fuente del la actualización de los nodos de chequeo con modificación de la ecuación de Gallager: "check_node_FI.c"