# Diseño de un Permutador para un Decodificador Turbo 3GPP LTE de Tasa Variable

Rodríguez Aguiñaga Adrian, Sánchez Adame Moisés, Calvillo Téllez Andrés

Centro de Investigación y Desarrollo de Tecnología Digital, CITEDI - IPN, Tijuana B. C.
TEL: 01(664)6231344,
correo-e: arodriguez@citedi.mx, msanchez@citedi.mx; calvillo@citedi.mx
*Paper received on 01/10/12, Accepted on 25/10/12.*

**Abstract.** Se presenta el análisis y modelación en Matlab de un turbo codificador con permutaciones variables. En la evaluación de la estimación se consideran las especificaciones técnicas de 3GPP LTE, para sistemas de comunicación de última milla sometidas a un ambiente ruidoso caracterizado por AWGN. La estructura del turbo codificación presento un mejor desempeño para una tasa de error de $1x10^{-7}$, mejorando la relación BER vs Eb/$N_0$, para comunicaciones inalámbricas con estándares UMTS.

**Keywords:** permutadores dou-binarios, turbo códigos, 3GPP, Tasa de error.

## 1 Introducción

La turbo codificación presenta múltiples trabajos en la comunidad internacional de las ciencias de la comunicación y continúa siendo un tema de investigación actual, en 1993, Claude Berrou, Alain Glavieux y Punya Thitimajshima, presentaron los Turbo códigos. Ellos centraban su interés en la codificación para corrección de errores, demostrando un excelente rendimiento y resultados que concordaban con la teoría desarrollada por el estadounidense Claude Shannon. Con ello nace una nueva era en el campo de la codificación y más específicamente, para los códigos decodificados iterativamente. Hoy día la demanda de servicios de telecomunicaciones se centra en tecnologías de cuarta generación (4G) donde las tasas máximas previstas son de 100 Mbps en enlace descendente y 50 Mbps en ascendente (con un ancho de banda en ambos sentidos de 20MHz).

Las comunicaciones inalámbricas se desarrollan constantemente, provocando la evolución de dispositivos móviles. La propuesta de este trabajo es crear un esquema de permutado basado en la arquitectura de la 3GGP, que resulte totalmente genérico, con esto se pretende que este modelo resulte aplicable a cualquier otro modelo de turbo codificación con el cual se quiera trabajar, con tan solo modificar las características y/o técnicas implementadas en la construcción de codificador o el decodificador y a su vez que el modelo permita la interacción con los bloques de bits y control se la energía simulada.

*Mireya García-Vázquez, Grigori Sidorov (Eds.).*
*Advances in Computing Science and Control.*
*Research in Computing Science 59, 2012, pp. 231-240.*

En la Sección dos se describe el desarrollo del algoritmo permutador, las problemáticas inmediatas, consideraciones, soluciones para la adaptación, el desarrollo e implementaciones del algoritmo, mostrando en la sección tres los resultados y finalmente en la sección cuatro las conclusiones

## 2 Desarrollo del Algoritmo Permutador

Los algoritmos de permutación, definen el gran parte el desempeño del turbo codificador, por lo cual tomamos como base los estándares para áreas locales y metropolitanas [1, 2], y en base a ellos se construyó el algoritmo que representara el funcionamiento del permutador conforme a los parámetros muestreados en el apéndice A.

- *N*: tamaño del bloque. El codificador/decodificar es alimentado por $k$ bits ($k = 2 \cdot N\ bits$).

- *P0, P1, P2* y *P3*: parámetros del algoritmo permutador o estados de transición según sea el tamaño de *N*, *Pi*, i=1, 2, 3…

Esta arquitectura de algoritmo (apéndice B) de doble paso permite construir y determinar el tamaño del bloque, por lo que debe de ser capaz de trabajar con bloques de datos de distintos tamaños, leer los bits de forma ordenada o reordenada según sea su caso, debido que cada paso emplea una técnica de asignación diferente.

De igual manera los valores de P, como se observa en la en el apéndice a, dependen de N, por lo que un ciclo recorre todo el rango de valores para N, desde 0 hasta N-1 y también a su vez conforme se ejecuta el algoritmo, los bloques entrantes a la cadena se pueden expresar como sub ciclos o procesos adyacentes, lo cual expone la obtener un grado de ejecución simultanea para los procesos, lo cual hace factible la implementación de la técnica de trabajo divide y vencerás, con lo que N, cambiara a n, siendo $n < N$ , $n \subset N$ y $n = \frac{N}{Pe}$, se asigna $Pe$ como el grado o nivel de bloques para efectuar el proceso en sub bloques.

### 2.1 Problemáticas Inmediatas

Normalmente los valores de $Pe$, son expresados en potencias de 2, sin embargo estos valores resultan no ser eficientes cuando presentan denominaciones elevadas, además de resultar en casos en los cuales los valores de Pe, podrían no ser divisibles a N, estos casos se tienes que definir y se atacaran con una técnica distinta [2], para toda Pi, donde $Pe$ puede tomar estos valores por los niveles de conveniencia definidos para evitar la complejidad [3], [4].

```
Para j= rango desde 0 hasta N-1
Subdivisión en procesos:
Proceso 1: desde 0 hasta n-1.
```

```
Proceso 2: desde n hasta 2·n-1.
Proceso n: desde (Pe-1)·n hasta N-1.
(k-1)·n hasta k·n-1.
```

Esta arquitectura presenta problemáticas para los casos de Pe = 8 y N= 36,108 y 180 y son tratados por la función fix, que se encarga de eliminar el fraccionario del resultante de la operación.

$$n = fix\left(\frac{n}{Pe}\right) \tag{1}$$

Otro punto a tener en cuenta es que la creación de rejillas o sub bloques creará solapamiento realizado por el decodificador. Con un solapamiento suficiente de muestras la decodificación será realizada óptimamente y a una alta tasa de bits [5] .Cada sub bloque $n$ es dividido en rejillas, y cada rejillas debe de estar compuesta por un número óptimo de muestras, para evitar conflictos de direccionamiento de memoria, ya que se disponen N bloques y con $Pe$ procesos, a cuales se necesitará almacenar los resultados en memoria y los datos serán almacenados y referenciados para n posiciones, es decir desde 0 hasta N-1, como se muestra en la Fig. 1.

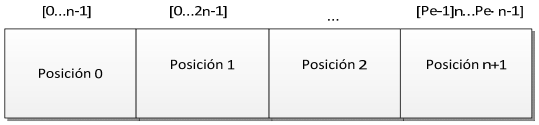


Fig. 1. Acceso a memoria evitando traslapes en la información de cada bloque.

Así la posición 0 de la memoria corresponderá con el índice 0, generado por el permutador y la posición n-1 corresponderá al índice N-1, así cuando el permutador genera secuencias de índices ordenados, cada uno de los procesos alternos también generara índices.

$$[(k-1)\cdot n, \ldots, k\cdot n-1] \tag{2}$$

Las entradas y salidas “suaves” deben de leer los datos entregados por el permutador en forma sincrónica, aun cuando haya existido algún error o colisión, para evitar la implementación de sincronizadores muy complejos [6 - 8].

## 2.2 Consideraciones y Soluciones para Adaptación del Algoritmo

Se tomaran en cuentas las consideraciones propuestas para resolver errores o colisiones, que son las siguientes:

1) Detectar los casos específicos de mayor ocurrencia error= 38, 108,180.

2) La proporción de recursiones con errores frente a las correctas, cuando se dan valores de *N* y *Pe* que producen un error es muy alta, se normaliza en todos los casos.
3) Cuando ocurre colisión, son únicamente dos los productores que colisionan es decir, una petición puede ser procesada al instante mientras la otra ha de esperar al siguiente ciclo para ser procesada.

Cuando la tercera consideración sea efectiva se detiene el permutador para meter el error a un ciclo nulo, el cual pude ayudar al sistema sincronizar, mediante la activación de un control de flujo de bits determinado por un diagrama de estados el cual auxiliara a mantener la sincronía y evitar errores de entrada y salida de datos, para el codificador o el decodificador, como se muestra en la Fig. 2.

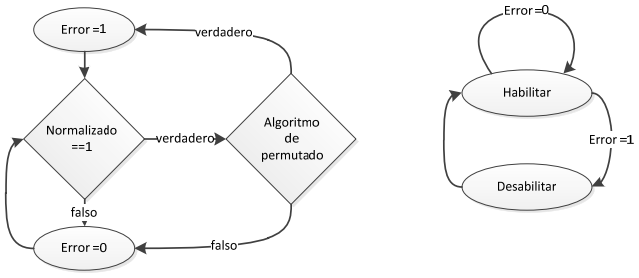


Fig. 2. Momentos en cual el permutador permitirá sincronización para evitar errores.

## 2.3 Desarrollo del Algoritmo

Los bloques son implementados en base a los algoritmos MAP, estos algoritmos realizan recorridos hacia adelante y hacia atrás, por lo cual, otra de las características a tomar en cuenta es que el permutador debe de operar estos bloques en la misma dirección que los decodificadores ver Fig.3. Para ello se hace uso del algoritmo LIFO (Last-Input, First Output), en el algoritmo permutador se generan únicamente índices, no direcciones y referencias por lo cual resulta importante implementar el sistema LIFO, para disponer de la lista direcciones cuando se requieran.

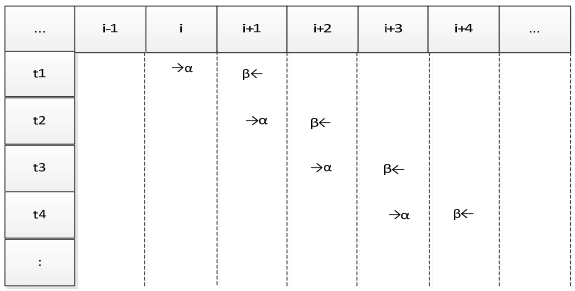


Fig. 3. Sub bloques para un algoritmo MAP, para el permutado.

Si observamos en la ilustración 3, $i+1$ y $i+2$ , en $t2$ alfa y beta ocurren al mismo tiempo, esta situación introduce la necesidad de crear un arreglo bidimensional o una pila doble para evitar conflictos y proveer direcciones así como referencias en conjunto con el generador de direcciones donde ya han sido reordenadas para conservar el orden obedeciendo el conmutado del algoritmo permutador.

## 2.4 Implementación del Algoritmo

Es posible manipular el algoritmo para lograr implementaciones de menor complejidad, tal es el caso de la división o inclusión de parámetros como sub procesos, estos permiten simplificar el orden del algoritmo y su ejecución, por lo que nos conviene establecer puntos de referencia en el algoritmo donde esto resulte valido (Tabla 2),a través del empleo de pocos recursos para calcular del módulo donde únicamente es necesario un registro como pila, un sustractor y un comparador de magnitud, esto permite mejorare la métrica aplicada a los cálculos en el modulo-N, esto significa que bastara con los dos bits menos significativos (BMS), para establecer su módulo.

Existe un incremento de ciclos en los bloques mayores, pero esto no necesariamente indica mayor complejidad dado que al ser un proceso de sumas recursivas y sub procesos mantiene su simplicidad, a diferencia de utilizar quizá menos ciclos de una arquitectura más compleja, por lo que el algoritmo queda descrito como se muestra a continuación:

Tabla 1. Algoritmo

```
for j = 0 … N-1
      if j = 0  { pila = 0}
          else  { pila = pila + P0}
end if
      if pila >= N { pila = pila – N}
end if
aux = 2BMS( binario( j ))
if aux = 00 { i = pila + 1}
         else if aux = 01
         { i = pila + mod(1+N/2+P1, N)}
```

Comportamiento

| | j | P0·j |
|---|---|---|
| Proceso 1 | 0 | 0 |
| | : | : |
| | n-1 | n-1·P0 |
| Proceso 2 | n | n·P0 |
| | : | : |
| | 2·n-1 | (2·n-1)·P0 |

```
else if aux = 10
        { i = pila + mod(1+P2, N)}
else if aux = 11
       { i = pila + mod(1+N/2+P3, N)}
end if
if i >= N
       { i = i – N}
end if
```

| | | |
|---|---|---|
| Proceso Pe | ( Pe-1)·n | ((Pe-1)·n)·P0 |
| | : | : |
| | n·Pe-1 | (n·Pe-1)·P0 |

Con esta implementación se consigue el mismo comportamiento así como la reducción de recursos empleados para ello.

Debido a que no todos los casos iniciales son cero, esto porque *n* no es en todos los casos múltiplo de N, el algoritmo implementa un análisis de los dos bits menos significativos del tamaño de sub-bloque *n* que son la compensación entre los casos del proceso *k* y *k+1*, así que en lugar de iniciar en cero para cada proceso se inicializa con una $variable = P0 \cdot k \cdot n$.

Para asegurar que no se realice ninguna recursión extra en la normalización del registro pila, el nuevo parámetro será normalizado antes de ser almacenado temporalmente, entonces la $variable = mod\,(P0 \cdot k \cdot n, N)$.

## 3 Resultados

Los experimentos se realizaron con bloques de datos desde 24 hasta 2400 bits (Tabla 2), con un código 1/3, para las simulaciones se empleó matlab, aplicando algoritmos de turbo codificadores 3GPP, trabajando en conjunto con las mejoras descritas en la sección 2 de este artículo, relacionadas al permutador de datos. Se establece un rango de pasos mayor a 0.2 dB e inferior a 2.4 dB para la relación BER vs Eb/N0, en la caracterización mostrada en las gráficas.

Para las cadenas de bloques mayores a mil bits, se observa un desempeño superior contra las de menor dimensión, como podemos observar en la ilustración 4 y 5, esto debido a que se observa como conforme aumenta el tamaño del bloque, así mismo como el grado información provista por los procesos es mayor y por tanto el rendimiento del permutador es mayor.

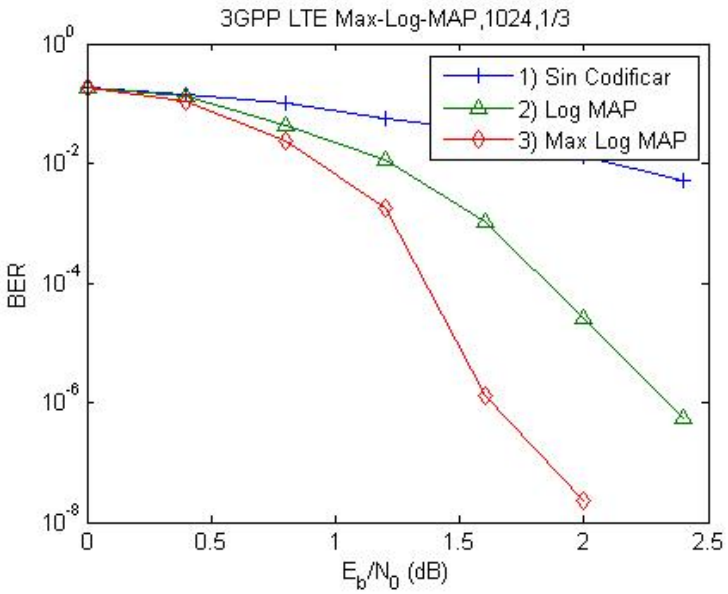


Fig. 4. Comportamiento de la tasa de error de bits, con bloques de permutación superiores a mil bits.

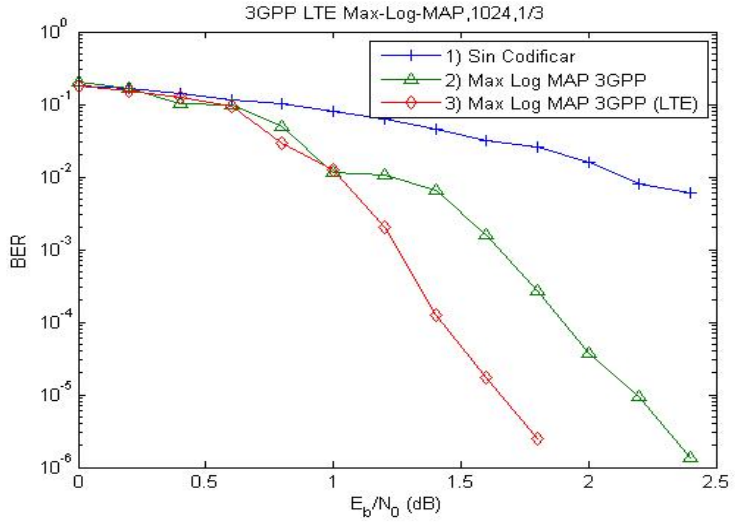


Fig. 5. Comportamiento de la tasa de error de bits, con bloques de permutación menores a mil bits.

Sin embargo debido a la necesidad de entregar los datos contemporáneamente y que en los bloques de bits de menor tamaño se encuentran los casos de error de N, resulta no ser divisible entero de un byte, el desempeño sigue siendo bajo.

## 4 Conclusiones

La implementación de las técnicas de permutado descritas en este artículo de logran tasas de error de bits, que superan la tasa de un error por cada millón de bits transmitidos, en base a las simulaciones realizadas con la sincronización de procesos de hasta $9.53x10^{-7}$, logrando esto por la retroalimentación de ambos CRS en forma sincrónica y con entradas controladas, en el cual se observa que el sacrifico de energía es de 0.2 dB, con un esquema de modulación BPSK y a través de un canal AWGN.

También se observa que a partir de la cuarta iteración los parámetros de BER, no sufren mayores ganancias, con lo cual se elimina tiempo de cómputo que ya no aportara mayores ganancias, al desempeño.

Al hacer estas adaptaciones a los permutadores, obtenemos una mejora en la tasa de bits, a diferencia de los modelos basados en cadenas de pseudo aleatorias de bits en un permutador. Además de disminuir el tiempo de procesamiento, definiendo el rango de iteraciones útiles para un proceso de codificación, y no alcanzar procesamientos en los cuales la información de ganancia ya no aporta datos relevantes para las decisiones suaves y duras del sistema de turbo codificación.

## 5 Referencias

1. IEEE 802.16-2004. IEEE Standard for Local and Metropolitan area Networks. Part 16: Air Interface for Fixed Broadband Wireless Access Systems.
2. IEEE 802.16e2005. IEEE Standard for Local and Metropolitan area Networks. Part 16: Air Interface for Fixed Broadband Wireless Access Systems.
3. S. Yoon, Y. Bar-Ness, "A Parallel MAP Algorithm for Low Latency Turbo Decoding",IEEE Communications Letters, Vol. 6, Nº 7, July 2002.
4. F. Speziali, J. Zory, "Scalable and Area Efficient Concurrent Permutador for HighThroughput Turbo-Decoders", IEEE Computer Society.
5. R. Dobkin, M. Peleg, R. Ginosar, “Parallel Permutador Design and VLSI Architecturefor Low-Latency MAP Turbo Decoders”, IEEE Transactions on VLSI Systems, Vol. 13, Nº 4, April 2005.
6. Corneliussen, A, Silpakar, P. Hardware-Accelerated NIOS-II Implementation of a Turbo Decoder. Computer and Electrical Engineering, 2009. ICCEE '09. Second International Conference on Date of Conference: 28-30 Dec. 2009
7. Han, J. Erdogan, A. Arslan, T. Power and Area Efficient Turbo Decoder Implementation for Mobile Wireless Systems. Signal Processing Systems Design and Implementation, 2005. IEEE Workshop on Digital Object Identifier. 2005, Page(s): 705 – 709.
8. Han, J.H.; Erdogan, A.T.; Arslan, T. Implementation of an efficient two-step SOVA turbo decoder for wireless communication systems. Global Telecommunications Conference, 2005. GLOBECOM '05. IEEE Volume: 4 2005, Page(s): 5 pp. - 2433

Tabla 2. Valores de P0, P1, P2 y P3 según N.

| N | P0 | P1 | P2 | P3 |
|---|---|---|---|---|
| 24 | 5 | 0 | 0 | 0 |
| 36 | 11 | 18 | 0 | 18 |
| 48 | 13 | 24 | 0 | 24 |
| 72 | 11 | 6 | 0 | 6 |
| 96 | 7 | 48 | 24 | 72 |
| 108 | 11 | 54 | 56 | 2 |
| 120 | 13 | 60 | 0 | 60 |
| 144 | 17 | 74 | 72 | 2 |
| 180 | 11 | 90 | 0 | 90 |
| 192 | 11 | 96 | 48 | 144 |
| 216 | 13 | 108 | 0 | 108 |
| 240 | 13 | 120 | 60 | 180 |
| 480 | 13 | 240 | 120 | 360 |
| 960 | 13 | 480 | 240 | 720 |
| 1440 | 17 | 720 | 360 | 540 |
| 1920 | 17 | 960 | 480 | 1440 |
| 2400 | 17 | 1200 | 600 | 1800 |

Tabla 3. Algoritmo binario de reordenamiento de datos para sistemas 3GPP.

Paso 1: alternar los pares de bits.

Asignar *u0 = [(A0, B0), (A1, B1), (A2, B2), (A3, B3),... (AN-1, BN-1)]* , como entrada del codificador 1.
Para i = 0 hasta *N*-1
Si ( i mod 2 == 1 ) intercambiar los pares *(Ai,Bi)* • *(Bi,Ai)*

Con lo cual obtendremos una secuencia similar a esta:
*u1 = [(A0,B0), (B1,A1), (A2,B2), (B3,A3), ... (BN-1,AN-1)] = [u1(0), u1(1), u1(2), u1(3), ... , u1(N-1)]*

Paso 2: P(j), la función *P(j), proporciona las direcciones en la secuencia* u1, que deberán ser asignada en direcciones "*j*" ,de la secuencia permutada • *u2(i) = u1(P(j))*

Para *j* = 0 hasta N-1
Conmutar *j* mod 4:
Caso 0: $P(j) = (P0 \cdot j + 1) \bmod N$
caso 1: $P(j) = (P0 \cdot j + 1 + N/2 + P1) \bmod N$
caso 2: $P(j) = (P0 \cdot j + 1 + P2) \bmod N$

caso 3: $P(j) = (P0 \cdot j + 1 + N/2 + P3) \bmod N$
la secuencia obtenida por el paso 2 es la siguiente:
u2 = [u1(P(0)), u1(P(1)), u1(P(2)), u1(P(3)), ... ,
u1(P(N-1))] =
= [(BP(0), AP(0)), (AP(1), BP(1)), (BP(2), AP(2)),
(AP(3), BP(3)), ... , (AP(N-1), BP(N-1))].
u2 es la secuencia de entrada para el codificador 2.