Simulación de errores cuánticos en el ambiente Scilab

Fonseca de Oliveira, André
Buksman, Efrain

Documento de Trabajo No. 4
Facultad de Ingeniería
Universidad ORT Uruguay
Julio, 2007
ISSN 1688-3217

Documento de Trabajo

UNIVERSIDAD ORT
Uruguay

ISSN 1688-3217

Simulación de errores cuánticos en el ambiente Scilab

Ing. André Fonseca de Oliveira
Dr. Efrain Buksman

Documento de Trabajo Nº 4
Facultad de Ingeniería
Universidad ORT Uruguay

Julio 2007

Simulación de errores cuánticos en el ambiente Scilab

André Fonseca de Oliveira y Efrain Buksman

***Abstract*—Este artculo presenta la implementación de un ambiente de simulacin de circuitos cunticos bajo el entorno abierto de computación numérica Scilab. Se detalla la propuesta de descripción de estados cuánticos y las rutinas implementadas para facilitar la construcción de circuitos y simulación de errores. A modo de ejemplo se describe la implementación de un código corrector de 3 qubits.**

I. Introducción

La computación cuántica surgió de los trabajos de Feynman, Bennett y Brassard entre otros [1] y se basa en la utilizacin de las propiedades de la mecánica cuántica, como la superposición y el entrelazado, en algoritmos computacionales. Está basada en un modelo teórico diferente al de la computación clásica (máquina de Turing), así como el computador clásico maneja bits con compuertas lógicas como AND y OR, el computador cuántico utiliza qubits. Las compuertas cuánticas operan sobre estos qubits, debiendo ser transformaciones unitarias y por tanto reversibles (Hadamard, Cnot, X , etc.). Si bien la computación cuántica está en su fase inicial (actualmente se trabaja con máquinas de un máximo de siete qubits), se espera que un computador escalable surja en un futuro cercano. Se cree que estos computadores cuánticos funcionando en su plenitud conseguirán resolver problemas en forma exponencialmente más rápida que sus pares clásicos (por ejemplo la defactorización de números primos).

El programa scilab [1] [2] es una plataforma de código abierto (*open source*) para la utilización y programación de algoritmos numéricos. Su facilidad de utilización permite al usuario crear bibliotecas de rutinas (toolboxes) para diversas áreas de conocimiento.

Este artículo describe un toolbox de simulación de compuertas cuánticas con la intención de ser utilizado como simulador genérico de algoritmos cuánticos. La sección II se describe las rutinas existentes en el toolbox, estando presentes algunos casos simples de circuitos cuánticos en la sección III. Como ejemplo principal se mostrará, en la sección IV, su uso en un sistema para el código corrector de tres qubits, precursor del codigo de Shor [3] de nueve qubits. Finalmente en la sección V, se sugieren futuras mejoras y aplicaciones para el toobox desarrollado.

A. Fonseca de Oliveira es catedrático de Electrónica Analógica y Control Automático del Departamento de Electrónica y Telecomunicaciones, Facultad de Ingeniería Bernard Wand-Polaken, Universidad ORT Uruguay (email: fonseca@ort.edu.uy).

E. Buksman es catedrático de Física en la Facultad de Ingeniería Bernard Wand-Polaken, Universidad ORT Uruguay (email: buksman@ort.edu.uy).

[1] La última versión, para diversos sistemas operativos, puede ser descargada de la página del consorcio *"www.scilab.org"*.

II. Descripción del toolbox

Esta sección detalla las funciones existentes en el toolbox desarrollado. La mayoría de las funciones tiene su nomenclatura basada en [4]. En II-A se detalla las formas de ingresar e interpretar un estado cuántico en sus dos representaciones: binaria y canónica. En II-B se describe las tranformaciones cuánticas implementadas, mientras que en II-C se detalla la las funciones que posibilitan la construcción de circuitos cuánticos. En II-D están las funciones auxiliares implementadas para el correcto funcionamiento del toolbox.

A. Estados cuánticos

Existen dos formas básicas de representar un estado cuántico de n qubits como combinación lineal de los estados base del espacio de Gilbert de dimensión 2^n:

- Utilizando una descripción binaria para los vectores base del espacio. En el caso $n = 2$ se tiene que

$$|q_{est} >= \alpha_0|00 > +\alpha_1|01 > +\alpha_2|10 > +\alpha_3|11 >,$$

$$\alpha_0^2 + \alpha_1^2 + \alpha_2^2 + \alpha_3^2 = 1.$$

- Mediante la combinación de vectores ortonormales de longitud 2^n (descripción canónica). Para el caso $n = 2$ se tiene

$$q_{est} = \left[\begin{array}{cccc} \alpha_0 & \alpha_1 & \alpha_2 & \alpha_3 \end{array} \right]^T.$$

El toolbox propuesto utiliza la segunda notación para los cómputos internos, pero permite al usuario utilizar la primera notación para ingresar estados cuánticos y visualizar los resultados obtenidos.

El ingreso un estado cuántico en la segunda notación es inmediato, siendo solamente necesario escribir un vector *columna* de dimensiones 1×2^n con norma *unitaria*.

A continuación vamos a describir la forma de trabajar con estados cuánticos utilizando la notación en binario. En el toolbox, un estado cuántico en la primer notación está representado mediante una lista de objetos. Cada objeto, a su vez, es una lista de dos objetos ms:

- el primero es el coeficiente o peso del vector,
- el segundo es el vector base, en binario, expresado mediante un vector fila.

En resumen, el estado

$$|q_{est} >= \frac{1}{\sqrt{2}}|00 > +\frac{1}{\sqrt{2}}|11 >$$

se construye como

```
q_est_lst = list(); // lista vacia

elem = list(1/sqrt(2), [0 0]);

// Agrega el elemento en la ultima posicin
q_est_lst($+1) = elem;

elem = list(1/sqrt(2), [1 1]);

// Agrega el elemento en la ultima posicion
q_est_lst($+1) = elem;
```

Este mismo estado puede ser representado utilizando la descripción canónica como

```
q_est_2n = sqrt(2) ([1 0 0 0]' + [0 0 0 1]');
```

Con la finalidad de facilitar la conversión entre las descripciones para los casos de dimensionalidad alta (n grande) fueron implementadas 3 funciones especiales en el *toolbox*:

- qb_lst2n.sci: Convierte de en notación binaria a notación canónica.

```
q_est_lst = list(); // lista vacia
elem = list(1/sqrt(2), [0 0]);
q_est_lst($+1) = elem;
elem = list(1/sqrt(2), [1 1]);
q_est_lst($+1) = elem;

q_est_2n = qb_lst2n(q_est_lst);
```

- qb_2nlst.sci: Convierte de notación canónica a notación binaria.

```
q_est_2n = sqrt(2) ([1 0 0 0]' + [0 0 0 1]');

q_est_lst = qb_2nlst(q_est_2n);
```

- qb_lstprint.sci: Imprime en pantalla el estado cuántico dado en forma binaria.

```
qb_lstprint(q_est_lst);

coef = 0.707107          base_vec =
!0  0  !

coef = 0.707107          base_vec =
!1  1  !
```

B. Transformaciones

En el toolbox están implementadas las principales tranformaciones unitarias.

- qb_trans_I.sci: *qb_trans_I(n)* crea la matriz M de transformacin identidad de n qubits.

```
MI = qb_trans_I(2)

MI  =

  1.  0.  0.  0.
  0.  1.  0.  0.
  0.  0.  1.  0.
  0.  0.  0.  1.
```

- qb_trans_X.sci: *qb_trans_X(n)* crea la matriz M de negación de n qubits.

```
MX = qb_trans_X(1)

MX  =

  0.  1.
  1.  0.
```

- qb_trans_Z.sci: *qb_trans_Z(n)* crea la matriz M de cambio de fase n qubits.

```
MZ = qb_trans_Z(1)

MZ  =

  1.    0.
  0.  - 1.
```

- qb_trans_Y.sci: *qb_trans_Y(n)* crea la matriz M de transformacin Y (composición iXZ) de n qubits.

```
MY = qb_trans_Y(1)

MY  =

  0  - i
  i    0
```

- qb_trans_Za.sci: *qb_trans_Za(n, α)* crea la matriz M de cambio de fase con un ángulo $\alpha\pi$ de n qubits.

```
MZa = qb_trans_Za(1, 1)

MZa  =

  1.   0
  0  - 1.
```

- qb_trans_Rk.sci: *qb_trans_Rk(n, k)* crea la matriz M definida por *qb_trans_Za(n, 2^(1-k))*.
- qb_trans_WalshHadamard.sci: *qb_trans_WalshHadamard(n)* crea la matriz M de transformación de Hadamard de n qubits.

```
MH = qb_trans_WalshHadamard(1)

MH  =

  0.7071068    0.7071068
  0.7071068  - 0.7071068
```

- qb_trans_Swap.sci: *qb_trans_Swap()* crea la matriz M de transformación *'Swap'* (intercambio entre los valores de 2 qubits).

```
MS = qb_trans_Swap

MS  =

  1.  0.  0.  0.
  0.  0.  1.  0.
  0.  1.  0.  0.
  0.  0.  0.  1.
```

- qb_trans_lambdaU.sci: *qb_trans_lambdaU(M)* crea la matriz Mc para la transformación M controlada por el primer *qubit*.

```
MX = qb_trans_X(1)

MX  =

   0.   1.
   1.   0.

M_C12 = qb_trans_lambdaU(MX)

M_C12  =

  1.  0.  0.  0.
  0.  1.  0.  0.
  0.  0.  0.  1.
  0.  0.  1.  0.
```

C. Circuitos cuánticos

Con la finalidad de la composicón de sistemas más complejos fueron implementadas las siguientes funciones:

- qb_prod_tensorial.sci: *qb_prod_tensorial(M1, M2)* crea la matrix M que representa el producto tensorial de las matrices (o vectores) $M1$ y $M2$. Se utiliza para composición de sistemas más complejos como producto tensorial de sistemas más simples. La figura 1 ilustra un sistema para estados de 4 qubits en el cuál se utiliza una compuerta $CNOT$ en donde el primer qubit es el de control ($C12$). Los otros qubits pasan a través de un subsistema identidad.

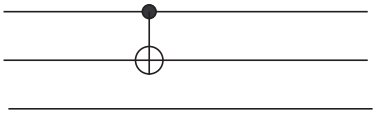

Fig. 1. Sistema con C21 e identidad en el qubit 3.

Este sistema se implementa de la siguiente forma:

```
MX = qb_trans_X(1)

MX  =

  0.  1.
  1.  0.

M_C12 = qb_trans_lambdaU(MX)

M_C12  =

  1.  0.  0.  0.
  0.  1.  0.  0.
  0.  0.  0.  1.
  0.  0.  1.  0.

M3 = qb_trans_I(1)

M3  =

  1.  0.
  0.  1.

M = qb_prod_tensorial(M_C12, M3)

M  =

  1.  0.  0.  0.  0.  0.  0.  0.
  0.  1.  0.  0.  0.  0.  0.  0.
  0.  0.  1.  0.  0.  0.  0.  0.
  0.  0.  0.  1.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  1.  0.
  0.  0.  0.  0.  0.  0.  0.  1.
  0.  0.  0.  0.  1.  0.  0.  0.
  0.  0.  0.  0.  0.  1.  0.  0.
```

- qb_cambia_filas.sci: *qb_cambia_filas(M, i, j)* realiza la matriz MC que representa un sistema igual al representado por la matrix M con el intercambio de los qubits i y j (no es lo mismo que aplicar la transformación *Swap*). El sistema de la figura 2 representa un sistema para estados de 3 qubits en donde se aplica la compuesta *Hadamard* en el tercer qubit mediante el control del primero.

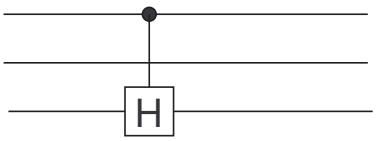

Fig. 2. Sistema Hadarmard en el tercer qubit controlado por el primero.

La implementación se realizaría de la siguiente forma:

```
M1 = qb_trans_WalshHadamard(1)

M1  =

  0.7071068    0.7071068
  0.7071068  - 0.7071068

M12 = qb_trans_lambdaU(M1)

M12  =

  1.  0.  0.           0.
  0.  1.  0.           0.
  0.  0.  0.7071068    0.7071068
  0.  0.  0.7071068  - 0.7071068

M3 = qb_trans_I(1)

M3  =

  1.  0.
  0.  1.

M = qb_prod_tensorial(M12, M3)
```

```
M  =

  1.   0.   0.   0.   0.    0.      0.      0.
  0.   1.   0.   0.   0.    0.      0.      0.
  0.   0.   1.   0.   0.    0.      0.      0.
  0.   0.   0.   1.   0.    0.      0.      0.
  0.   0.   0.   0.   0.7   0.      0.7     0.
  0.   0.   0.   0.   0.    0.7     0.      0.7
  0.   0.   0.   0.   0.7   0.    - 0.7     0.
  0.   0.   0.   0.   0.    0.7     0.    - 0.7

M = qb_cambia_filas(M, 2, 3)

M  =

  1.   0.   0.   0.   0.      0.     0.      0.
  0.   1.   0.   0.   0.      0.     0.      0.
  0.   0.   1.   0.   0.      0.     0.      0.
  0.   0.   0.   1.   0.      0.     0.      0.
  0.   0.   0.   0.   0.7     0.7    0.      0.
  0.   0.   0.   0.   0.7   - 0.7    0.      0.
  0.   0.   0.   0.   0.      0.     0.7     0.7
  0.   0.   0.   0.   0.      0.     0.7   - 0.7
```

- qb_medida.sci: *qb_medida(qb_est_in, ind_qb, Mbmatrix)* realiza la medida del qubit *"ind_qb"* del estado *qb_est_in*. La matrix *Mbmatrix* es la matriz que contiene en sus filas todos los estados binarios posibles para la cantidad de qubits del estado *qb_est_in*, siendo su utilización opcional (simplemente por desempeño del código). Esta rutina devuelve un estado *qb_est_sal* con la misma cantidad de $qubits$, pero con la medida en el qubit *"ind_qb"* realizada (al azar, con las probabilidades según el estado original).

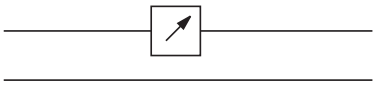

Fig. 3. Sistema con medida en el primer qubit.

La figura 3 representa un sistema en el cuál se mide el primer qubit. El código para implementar este sistema es el detallado a continuación:

```
q_est = list();
a=list(0.5, [0 0]);
q_est($+1)=a;
a=list(0.5, [0 1]);
q_est($+1)=a;
a=list(0.5, [1 0]);
q_est($+1)=a;
a=list(0.5, [1 1]);
q_est($+1)=a;

qb_lstprint(q_est)
```

```
coef = 0.500000          base_vec =
!0  0  !

coef = 0.500000          base_vec =
!0  1  !

coef = 0.500000          base_vec =
!1  0  !

coef = 0.500000          base_vec =
!1  1  !

q_2n = qb_lst2n(q_est)

q_2n  =

  0.5
  0.5
  0.5
  0.5

[q_2n_out, q_med] = qb_medida(q_2n, 1)

q_med  =

  0.
q_2n_out  =

  0.7071068
  0.7071068
  0.
  0.

q_est_out = qb_2nlst(q_2n_out);

qb_lstprint(q_est_out)

coef = 0.707107          base_vec =
!0  0  !

coef = 0.707107          base_vec =
!0  1  !
```

- qb_binfunc.sci: *qb_binfunc(Mfunc)* crea la matriz Mf que realiza las funciones lógicas indicada en la matriz Mf. La matriz debe tener 2^n filas indicando que son funciones booleana de n bits, y m columnas que indican la cantidad de funciones. Para un mejor detalle referirse a la figura 4.

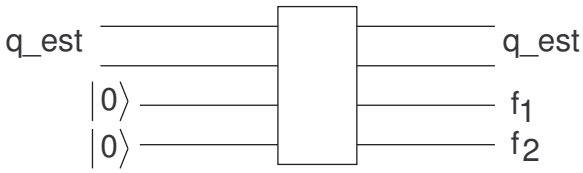

Fig. 4. Sistema con funciones lógicas $a \,\&\, b$ y $\bar{a} \,\&\, b$.

```
// Estado 0.5|00>+0.5|01>+0.5|10>+0.5|11>

q_est = [0.5 0.5 0.5 0.5]'

q_bin = qb_2nlst(q_est);
qb_lstprint(q_bin);

 coef = 0.5   base_vec =

!0  0  !

 coef = 0.5   base_vec =

!0  1  !

 coef = 0.5   base_vec =

!1  0  !

 coef = 0.5   base_vec =

!1  1  !

// Creamos la descripcin de ambas funciones
// NOT(a) AND b ..... a AND b

Mat_fun = [0 0; 1 0; 0 0; 0 1]

Mat_fun  =

  0.  0.
  1.  0.
  0.  0.
  0.  1.

Mf = qb_binfunc(Mat_fun);

// El nuevo sistema va a tener
// 2 qubits extras de entrada

q_in = qb_prod_tensorial(q_est, [1 0 0 0]');

q_sal = Mf*q_in;

q_bin = qb_2nlst(q_sal);
qb_lstprint(q_bin);

 coef = 0.5   base_vec =

!0  0  0  0  !

 coef = 0.5   base_vec =

!0  1  1  0  !

 coef = 0.5   base_vec =

!1  0  0  0  !

 coef = 0.5   base_vec =

!1  1  0  1  !

// Las funciones se observan en los
// 2 ultimos qubits!
```

D. Funciones auxiliares

La principal función auxiliar implementada devuelve un vector de 2^n filas y n columnas, en donde las filas forman una sucesión de todos los binarios de 0 a 2^{n-1}.

- qb_bmatrix.sci: *qb_bmatrix(n)* devuelve los números binarios de 0 a2^{n-1} en las filas de la matriz M resultado.

```
Mb = qb_bmatrix(3)

Mb  =

  0.  0.  0.
  0.  0.  1.
  0.  1.  0.
  0.  1.  1.
  1.  0.  0.
  1.  0.  1.
  1.  1.  0.
  1.  1.  1.
```

III. EJEMPLOS DE USO

A. Ejemplo 1

El primer ejemplo a ser implementado es ilustrado en la figura 5.

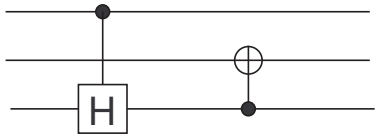

Fig. 5. Sistema del ejemplo 1.

El código para implementar el sistema es:

```
// Primera transformacion

Mtemp = qb_trans_WalshHadamard(1);
// Hadamard 2 fila controlado
Mtemp = qb_trans_lambdaU(Mtemp);
// Tercer qubit
M1 = qb_prod_tensorial(Mtemp, qb_trans_I(1));
M1 = qb_cambia_filas(M1, 2, 3);

// Segunda transformacion

Mtemp = qb_trans_X(1);
// C12
Mtemp = qb_trans_lambdaU(Mtemp);
// Tercer qubit
```

```
M2 = qb_prod_tensorial(Mtemp, qb_trans_I(1));
// C32
M2 = qb_cambia_filas(M2, 1, 3);

// Sistema total

M = M2*M1;
```

A continuación aplicamos el sistema de transformaciones al estado $|100>$.

```
q_est_bin_in = list(list(1, [1 0 0]));

q_est_2n_in = qb_lst2n(q_est_bin_in);

q_est_2n_sal = M*q_est_2n_in;

q_est_bin_sal = qb_2nlst(q_est_2n_sal);

qb_lstprint(q_est_bin_sal);

coef = 0.707107          base_vec =
!1  0  0  !

coef = 0.707107          base_vec =
!1  1  1  !
```

B. Ejemplo 2

Para este segundo ejemplo se construye el sistema de la figura 6.

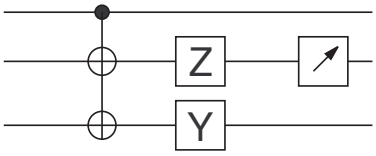

Fig. 6. Sistema del ejemplo 2.

La implementación se realiza como a continuación.

```
// Primera transformacion

Mtemp = qb_trans_X(2);
// C1_23
M1 = qb_trans_lambdaU(Mtemp);

// Segunda transformacin

Mtemp = qb_trans_I(1);
Mtemp2 = qb_trans_Z(1);
Mtemp = qb_prod_tensorial(Mtemp, Mtemp2);
Mtemp2 = qb_trans_Y(1);
M2 = qb_prod_tensorial(Mtemp, Mtemp2);

// Sistema total

M = M2*M1;
// Al resultado de este sistema
// se mide el segundo qubit
```

Si se tiene como entrada el estado $|101>$

```
q_bin_in = list(list(1, [1 0 1]));

q_est_2n_in = qb_lst2n(q_bin_in);

q_est_2n_sal = M*q_est_2n_in;

[q_med, q_bit] = qb_medida(q_est_2n_sal,2)

 q_bit  =

    1.
 q_med  =

    0
    0
    0
    0
    0
    0
    0
  - i

q_bin_sal = qb_2nlst(q_med);

qb_lstprint(q_bin_sal);

 coef = -%i   base_vec =
!1  1  1  !
```

IV. Código de tres qubits

A. Código correctores para canales con ruido

Otra de la áreas de la computación cuántica en la que se ha avanzado es el uso de canales cuánticos para la comunicación segura, a lo que se le suele llamar criptografía cuántica [5]. Este campo no necesita en principio de un computador cuántico, sino que utiliza solamente tecnología de manipulación de un solo fotón a través de fibra óptica y ya se encuentra en su fase comercial.

Lamentablemente los qubits cuánticos son sumamente frágiles y los canales de comunicación cuánticos presentan ruido, o sea, sufren de interferencia con el ambiente llamada comúnmente de decoherencia. Esta decoherencia introduce errores y debe ser controlada si queremos mandar mensajes en forma eficiente. En varios artículos recientes [6][7][8] se ha considerado la corrección de errores como la forma más efectiva de controlar estos efectos indeseados.

La interacción con el ambiente se entiende como una evolución unitaria del conjunto qubit más entorno, ocasionando una proyección del qubit sobre el subespacio de medida. Sin embargo es posible demostrar que un error cualquiera en un solo qubit que incluya decoherencia se puede expresar como una combinación de los cuatro operadores:

$$E = e_0 I + e_1 X + e_2 Z + e_3 XZ$$

donde I es la identidad, X es el operador flip (negación)

$$X(a|0> + b|1>) = a|1> + b|0>$$

y Z el cambio de fase

$$Z(a|0> + b|1>) = a|0> - b|1> .$$

Estos operadores conforman una base para cualquier transformación [5]. Esto permite que sea posible limitar el estudio a aplicaciones de solamente los operadores X y Z.

B. Corrector de tres qubits

El código de tres qubits 7 fue diseado con la finalidad de la detección y corrección de errores de negación producidos por el canal [9]. Es un código de distancia mínima tres y para esto utiliza dos qubits auxiliares. En el sistema originalmente propuesto se genera y se mide el síndrome del error que permite decidir, utilizando lógica clásica, cual de los tres qubits tiene que ser corregido.

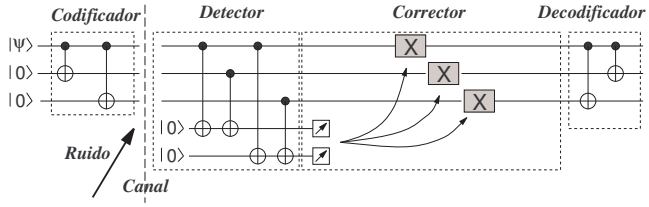

Fig. 7. Código de 3 qubits. Sistema original.

En este ejemplo se propone otro tipo de corrección que utiliza solamente compuertas cuánticas y no utiliza en ninguna medición. Este código además de ser automático, consigue corregir errores de dos qubits del tipo Cnot como se muestra a continuación.

La figura 8 ilustra al proceso de codificación, envio (sobre un canal) y recepción (detección, corrección y decodificación) del sistema propuesto.

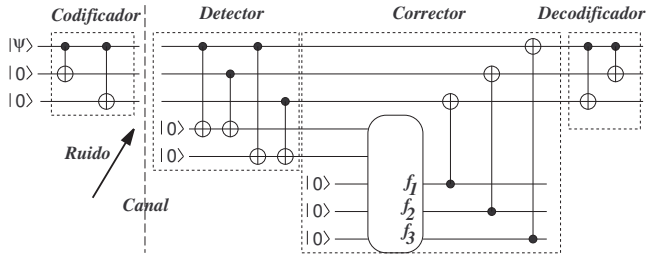

Fig. 8. Código de 3 qubits. Sistema propuesto.

A continuación se explica el desarrollo del ejemplo de implementación y utilización del código propuesto utilizando el toolbox desarrollado.

C. Qubit de entrada

Para la demostración del uso del codificador se aprovechar la propiedad de paralelismo de la computación cuántica y se ingresará al sistema el qubit

$$|+> = \frac{\sqrt{2}}{2}(|0> + |1>) .$$

Para construir el estado cuántico q_est = $|+>$ se ingresa el siguiente código en scilab:

```
// qubit de entrada

qbit_in = [1 1]';
qbit_in = qbit_in/norm(qbit_in);

qb_lst = qb_2nlst(qbit_in);
qb_lstprint(qb_lst);

coef = 0.7071068   base_vec =

 0

coef = 0.7071068   base_vec =

 1
```

D. Codificador

Con la finalidad de poder recuperar posibles errores debidos al envio de la información por un canal real, se codifica la señal con información redundante. En este ejemplo se utilizará el, ya mencionado, codificador de tres qubits (figura 9) :

$$\begin{aligned} |0> &\rightarrow |000> \\ |1> &\rightarrow |111> \end{aligned}$$

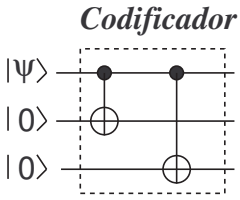

Fig. 9. Codificador.

Como primer paso se debe ampliar el espacio del qubit de entrada a un espacio complejo de dimensón tres (tres qubits). Esto se hace mediante el producto tensorial del qubit de entrada con el estado $|00>$.

```
// Agregamos 2 qubits

// Estado 1|00>
q_temp = [1 0 0 0]';

// Composicion con la entrada
// Estado de 3 qubits
```

```
q_in = qb_prod_tensorial(qbit_in, q_temp);

qb_lst = qb_2nlst(q_in);
qb_lstprint(qb_lst);

coef = 0.7071068   base_vec =

!0  0  0  !

coef = 0.7071068   base_vec =

!1  0  0  !
```

A continuación armamos la matriz del sistema codificador y evaluamos el estado final codificado.

```
// Sistema

M1 = qb_trans_X(1);
M1 = qb_trans_lambdaU(M1);
Mtemp = qb_trans_I(1);
M1 = qb_prod_tensorial(M1, Mtemp);

M2 = M1;
M2 = qb_cambia_filas(M1, 2, 3);

Mcod = M2*M1;

// Codifica

q_cod = Mcod * q_in;

qb_lst = qb_2nlst(q_cod);
qb_lstprint(qb_lst);

coef = 0.7071068   base_vec =

!0  0  0  !

coef = 0.7071068   base_vec =

!1  1  1  !
```

En este ejemplo se ve claramente el paralelismo: el estado final indica al mismo tiempo la codificación de un estado $|0>$ y un estado $|1>$.

E. Canal

Esta parte simula posibles errores y/o desviaciones introducidas en el canal de comunicación. Hay que armar un sistema que transforme un estado de tres qubits (salida del codificador). En este ejemplo se simulará un error mediante una transformación $C12$ (figura 10). Considerando el qubit de entrada ($|+>$) esto resultará en un estado cuántico en el cuál habrá error el en segundo bit para el estado base $|111>$, pero no para el $|000>$. Luego con un sólo ejemplo es posible simular la transmisión con error y sin error (una vez más el paralelismo cuántico);

```
// Canal con C12
```

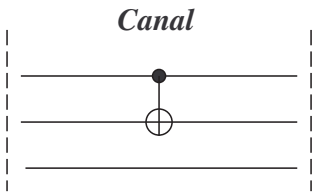

Fig. 10. Canal.

```
Mcanal = qb_trans_X(1);
Mcanal = qb_trans_lambdaU(Mcanal);
Mtemp = qb_trans_I(1);
Mcanal = qb_prod_tensorial(Mcanal, Mtemp);

q_canal = Mcanal * q_cod;

qb_lst = qb_2nlst(q_canal);
qb_lstprint(qb_lst);

coef = 0.7071068   base_vec =

!0  0  0  !

coef = 0.7071068   base_vec =

!1  0  1  !
```

F. Detector

Esta parte del sistema detecta si hay algún qubit erróneo en la recepción (erroes de negación). Para esto se amplia la dimensión del estado cuántico a cinco mediante el agregado de dos qubits. La figura 11 ilustra el sistema propuesto.

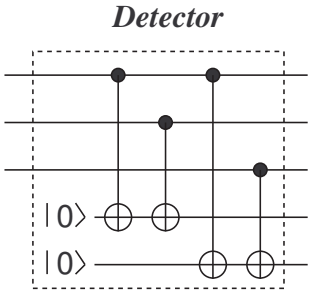

Fig. 11. Detector.

```
// Transformacion de 3 qubits a 5 qubits

q_corr1 = qb_prod_tensorial(q_canal, [1 0 0 0]');

qb_lst = qb_2nlst(q_corr1);
qb_lstprint(qb_lst);
```

```
coef = 0.7071068   base_vec =

!0  0  0  0  0  !

coef = 0.7071068   base_vec =

!1  0  1  0  0  !

// Deteccion

M1 = qb_trans_X(1);
M1 = qb_trans_lambdaU(M1);
Mtemp = qb_trans_I(3);
M1 = qb_prod_tensorial(M1, Mtemp);
M1 = qb_cambia_filas(M1, 2, 4);

M2 = M1;
M2 = qb_cambia_filas(M2, 1, 2);

M3 = M1;
M3 = qb_cambia_filas(M3, 4, 5);

M4 = M3;
M4 = qb_cambia_filas(M4, 1, 3);

MCorr1 = M4*M3*M2*M1;

q_corr2 = MCorr1 * q_corr1;

qb_lst = qb_2nlst(q_corr2);
qb_lstprint(qb_lst);

coef = 0.7071068   base_vec =

!0  0  0  0  0  !

coef = 0.7071068   base_vec =

!1  0  1  1  0  !
```

La informacón contenida en los nuevos qubits son el síndrome del error En este caso la acción correctiva deberá:

- En el caso de tener $|00>$ se considera que no hay error.
- En el caso de tener $|01>$ se considera que hay en el tercero qubit.
- En el caso de tener $|10>$ se considera que hay en el segundo qubit.
- En el caso de tener $|11>$ se considera que hay en el primer qubit.

Para esto se deberá realizar la lógica necesaria en el sistema corrector, explicado a continuación.

G. Corrector

Este sistema realiza tres funciones booleanas (en su forma cuántica) para la determinación de cuál qubit será corregido (mediante inversión). La figura 12 ilustra este proceso, en donde f_1, f_2 y f_3 representan las salidas para las funciones $\bar{a}b$, $a\bar{b}$ y ab, respectivamente.

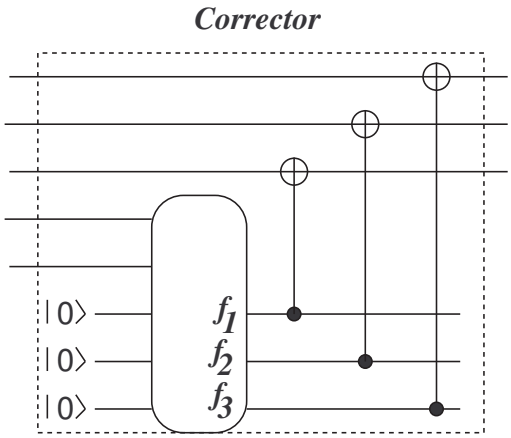

Fig. 12. Corrector.

Para la utilización del sistema es necesario ampliar nuevamente el espacio de trabajo (3 qubits más, uno por función boolenana).

```
// Transformacion de 5 qubits a 8 qubits
// para las funciones binarias

q_temp = [1 0 0 0 0 0 0 0]';
q_corr3 = qb_prod_tensorial(q_corr2, q_temp);

qb_lst = qb_2nlst(q_corr3);
qb_lstprint(qb_lst);

coef = 0.7071068   base_vec =

!0  0  0  0  0  0  0  0  !

coef = 0.7071068   base_vec =

!1  0  1  1  0  0  0  0  !


b_x = qb_bmatrix(2)

 b_x  =

  0.  0.
  0.  1.
  1.  0.
  1.  1.

fcorr = (1-b_x(:,1)).*b_x(:,2);
fcorr = [fcorr,(b_x(:,1).*(1-b_x(:,2)))];
fcorr = [fcorr,(b_x(:,1).*b_x(:,2))]
```

```
fcorr  =

  0.  0.  0.
  1.  0.  0.
  0.  1.  0.
  0.  0.  1.

Mf = qb_binfunc(fcorr);

// Sistema - parte2

M1 = qb_trans_I(3);
MCorr2 = qb_prod_tensorial(M1, Mf);

q_corr4 = MCorr2 * q_corr3;

qb_lst = qb_2nlst(q_corr4);
qb_lstprint(qb_lst);

coef = 0.7071068   base_vec =

!0  0  0  0  0  0  0  0  !

coef = 0.7071068   base_vec =

!1  0  1  1  0  0  1  0  !

// Correctores

M1 = qb_trans_X(1);
M1 = qb_trans_lambdaU(M1);
Mtemp = qb_trans_I(6);
Mtemp = qb_prod_tensorial(M1,Mtemp);
M1 = qb_cambia_filas(Mtemp, 2, 3);
M1 = qb_cambia_filas(M1, 1, 6);

M2 = Mtemp;
M2 = qb_cambia_filas(M2, 1, 7);

M3 = Mtemp;
M3 = qb_cambia_filas(M3, 1, 8);
M3 = qb_cambia_filas(M3, 1, 2);

MCorr3 = M3*M2*M1;

q_corr5 = MCorr3 * q_corr4;

qb_lst = qb_2nlst(q_corr5);
qb_lstprint(qb_lst);

coef = 0.7071068   base_vec =

!0  0  0  0  0  0  0  0  !

coef = 0.7071068   base_vec =

!1  1  1  1  0  0  1  0  !
```

Se puede observar que los tres primeros qubits representan el código original y ya están corregidos.

H. Decodificador

El sistema decodificador lleva los qubits segundo y tercero a cero. En este ejemplo esto no ha sido implementado, ya que si se observa el resultado de la salida del corrector en los primeros tres qubits se tiene el mensaje (codificado) original. La información es tranportada solamente en el primer qubit.

Si para algún caso particular es necesaria, su implementación es inmediata y similar a los bloques anteriores.

V. Conclusiones y futuros trabajos

Si bien los errores que provienen de la decoherencia (ruido) de un canal son corregibles, las compuertas que se aplican sobre los qubits pueden producir errores por si mismas. Luego una forma de solucionar el problema sería codificar cada compuerta en un proceso de cadena o concatenación [8]. Este procedimiento puede ocacionar un aumento del error en lugar de disminuirlo. En principio sería interesante estudiar cuantitativamente el efecto producido por un error no corregido por estar este en el último eslabón de la concatenación chequeando asi la corrección tolerante a fallos. En un proximo trabajo se podria hacer un estudio comparativo del efecto estadistico de cada tipo de error (Z,Z,etc) por ejemplo graficando la distancia definida de alguna manera entre el resultado obtenido y el esperado [5].

References

[1] Richard Feynman, "Simulating physics with computers," *Journal of Theoretical Physics*, vol. 21, pp. 467, 1982.

[2] Claude Gomez, *Engineering and Scientific Computing with Scilab*, Springer, 1999.

[3] P. W. Shor, "Scheme for reducing decoherence in quantum computer memory," *Physical Review A*, vol. 52, pp. 2493–6, 1995.

[4] Jesus Garcia López de Lacalle, "Introducción a la algorítmica y criptografía cuánticas," Apuntes del curso de doctorado dictado en la Universidad ORT Uruguay, 2006.

[5] Michael A. Nielsen and Isaac L. Chuang, *Quantum Computation and Quantum Information*, Cambridge University Press, Cambridge, 2000.

[6] D. Gottesman, "Fault-tolerant quantum computation with local gates," *Journal of Modern Optics*, vol. 47, pp. 333–345, 2000.

[7] J. Preskill, "Reliable quatum computers," *Proc. R. Soc. Lond. A*, pp. 385–410, 1998.

[8] P. W. Shor, "Fault-tolerant quantum computation," *Proc. 35th Annual Sysposium on Fundamentals of Computer Science*, pp. 56–65, 1996.

[9] A. M. Steane, "Quatum computing and error correction," http://arxiv.org/abs/quant-ph/0304016, 2000.