QuantumLab: simulador de código abierto para computación cuántica

Laura Gatti , André Fonseca de Oliveira , Efrain Buksman , Jesús García-López

***Resumen**: En este trabajo, se presenta un simulador para computación cuántica desarrollado en el ambiente de cálculo numérico Scilab. Está construido como una colección de bibliotecas de funciones ordenadas de forma temática con el objetivo de permitir una ampliación del mismo de manera sencilla. Se da una breve descripción de las rutinas implementadas y un ejemplo de uso.*

***Palabras clave**: Quantum computation, Numerical simulation*

***Abstract**: In this article we introduce a quantum computer simulator implemented in the numerical computation environment Scilab. It is developed as a collection of libraries arranged thematically, in order to simplify future extensions. A brief description of implemented routines is given as well as an example of use.*

***Keywords:** Quantum Computation, Numerical simulation.*

1. Introducción

El área de computación cuántica surgió de los trabajos pioneros de Feynman y Deutsch [1][2], entre otros, ante la idea de utilizar la evolución de sistemas cuánticos como una herramienta de cálculo en sí misma. Se basa en la utilización de las propiedades de la mecánica cuántica, como la superposición y el entrelazamiento para la realización de algoritmos computacionales.

Hay importantes diferencias en la manipulación de la información cuántica respecto a la clásica. Ejemplos de estas diferencias son el teorema de no clonación para estados cuánticos y el hecho de que todas las operaciones en un sistema cerrado deben ser reversibles haciendo imposible la realización de una compuerta AND clásica.

Este artículo describe el ambiente QuantumLab3, un simulador de código abierto construido en el ambiente *Scilab* [3] con la intención de ser utilizado como simulador genérico de algoritmos cuánticos para la enseñanza e investigación. *Scilab* es una plataforma de cálculo numérico que permite la creación de bibliotecas de rutinas (toolboxes) para diversas áreas de conocimiento. Esta versión de *QuantumLab* incluye una ampliación en la cantidad de operaciones, y mejoras numéricas respecto a las anteriores [4].

En la sección 2, se describe la estructura del simulador, mientras los distintos formatos para la representación de estados cuánticos son introducidos en la sección 3. Las diferentes bibliotecas de funciones son detalladas en la sección 4. En la sección 5, se ilustra un ejemplo de aplicación del simulador en el análisis de la propagación de errores en un código cuántico. Finalmente, en la sección 6, son mencionadas las líneas para trabajos futuros y las conclusiones del uso del simulador.

2. Estructura del simulador

El simulador *QuantumLab³* está desarrollado como un conjunto de bibliotecas de funciones ordenadas de forma temática en diferentes carpetas, con el objetivo de mantener un orden estructural en el simulador y así permitir una ampliación del mismo de manera sencilla.

En la versión actual, hay cinco áreas como es indicado en la Fig. 1.

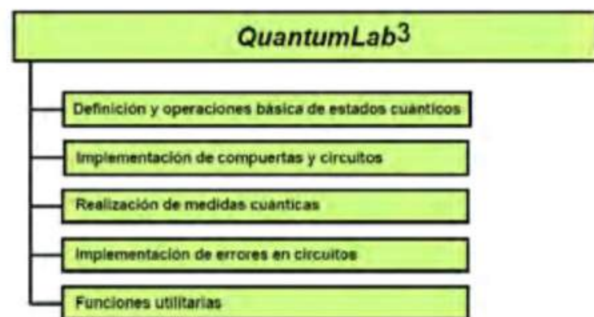


Figura 1. Estructura de carpetas de funciones del simulador *QuantumLab³*.

3. Representación de estados cuánticos

Con la intención de tener flexibilidad en la elección entre el desempeño en los cálculos y el estudio de la propagación de errores, se ha propuesto distintos formatos para la representación de estados, o ensambles de estados cuánticos. En la elección, se ha considerado su estructura, o sea, el tipo de objeto y sus dimensiones, de forma de imposibilitar el confundir un formato con otro. De aquí que el simulador reconoce cuatro tipos de formatos válidos para representar estados que se resumen a continuación:

F_I: Formato vectorial para puros (ensamble de estados): es una matriz que contiene m estados de n qubits, representados mediante una matriz de dimensiones $2^n \times m$. Cada columna de la matriz representa un estado cuántico. Todos los estados del ensamble deben ser de la misma dimensión.

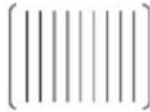


Figura 2. Estado tipo I.

F_II: Formato para mezcla vectorial (único estado): es una lista que contiene únicamente dos elementos. El primero es una matriz que representa un ensamble de puros como en el formato anterior, y el segundo un vector fila con la misma cantidad de elementos que las columnas de la matriz, representado las probabilidades de cada estado puro. Es un estado mezcla válido para el simulador.

Figura 3. Estado tipo II.

F_III: Formato para mezcla matricial (único estado): es una lista que contiene las matrices de densidad de todos los estados cuánticos componentes (sean estos puros o mezclas) y un vector fila conteniendo las probabilidades de cada uno de los estados cuánticos anteriores. Todas las matrices de densidad deben tener la misma dimensión.

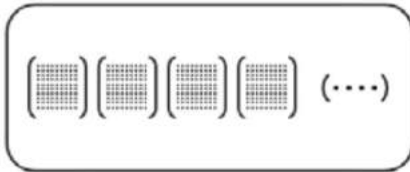


Figura 4. Estado tipo III.

F_IV: Formato matricial (ensamble de estados): consta de una lista que contiene tantas matrices de densidad como estados cuánticos se quieran representar. Todas deben de tener la misma dimensión.

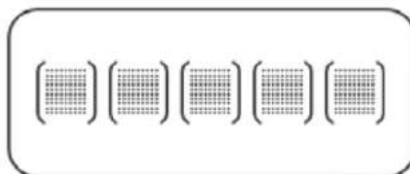


Figura 5. Estado tipo IV

En el simulador, se denomina *estado* a un objeto que cumpla todas las condiciones para ser clasificado en uno de los formatos representados en las figuras de 2 a 5 (un sólo estado cuántico o un ensamble de estados cuánticos).

4. Funciones del toolbox

4.1. Funciones para estados cuánticos

Son un conjunto de funciones que facilitan la creación y la manipulación de estados cuánticos. La figura 6 ilustra el conjunto de funciones definidas. Todas utilizan el prefijo *qb_st*.

Figura 6. Funciones para estados cuánticos.

- *state = qb_st_state(st, nqb, coef)*. Crea estados de tipos vectoriales en base a los elementos de la base computacional de la dimensión que se elija. Se pueden crear tanto un conjunto de estados puros como un estado mezcla en el formato tipo *II*.
- *state = qb_st_rand(tipo, dim, cant, orden)*. Crea estados vectoriales randómicos del tipo de formato que se elija, de la dimensión y compuesto por la cantidad de elementos que se quiera.
- *[typ, cant] = qb_st_chktype(st)*. Analiza la estructura de un objeto y determina si el mismo es un estado válido. En el caso de serlo, devuelve como primer parámetro un número que identifica el tipo de formato y la cantidad de estados.
- *bool = qb_st_chkpurestate(st [, tipoest])*. Verifica si un estado es puro (o todos en el caso de ensambles).
- *st-out = qb_st_chkmezcla(st [, tipoest])*. Verifica si un estado mezcla está correctamente definido. En caso contrario, lo corrige.
- *dim = qb_st_dim(st [, tipoest])*. Devuelve la cantidad de qubits que componen al estado cuántico.
- *[res, pos] = qb_st_chknorm(st [, tipoest])*. Verifica si el estado cumpla la condición de normalización. En el caso de ensambles, todas deben cumplirla y, de no ser así, devuelve la posición de los que no cumplen.
- *[res, pos]=qb_st_normalize(st [, tipoest])* Verifica que los estados estén normalizados. En caso contrario, los normaliza.
- *st_out = qb_st_conver(st_in, type out [, tipoest])* Permite, si es posible, convertir un estado a otro formato del tipo que se elija.
- *[sts out,v fases] = qb_st_globalextract(sts in [, tipoest])*. Para estados con descripción vectorial extrae la fase global.
- *st_o = qb_st_extractst(st, ind [, tipoest])*. Para ensambles representados en los formatos *I* y *IV*, posibilita extraer los estados indicados.

- *st_o = qb_st_elimst(st, ind [, tipoest])*. Para ensambles representados en los formatos *I* y *IV*, elimina los estados indicados.
- *st_o = qb_st_ordenlexico(st [, tipoest])*. Ordena ensambles de estado de forma a posibilitar la comparación.
- *res = qb_st_equal(st 1, st 2, orden [, tipoest_1, tipoest_2])*. Compara dos estados de forma a determinar si tienen los mismos elementos.
- *res = qb_st_equaldens(st_1, st_2, orden [, tipoest_1, tipoest_2])*. Determina si dos estados representan la misma matriz de densidad.
- *st_o = qb_st_agregast(st, st nuevo, [, tipoest_1, tipoest_2])*. Agrega estados a un conjunto de estados original. Solo implementado para estados de los tipos *I* y *IV*.
- *qb_st_lstprint(st, [tipoest])*. Imprime en pantalla un estado cuántico en formato binario. Solo implementado para estados de los tipos *I* y *II*.

4.2. Funciones para circuitos cuánticos

Son funciones que permiten obtener compuertas clásicas utilizadas para la creación de circuitos cuánticos más complejos. Utilizan el prefijo *qb_circ* como ilustrado en la figura 7.

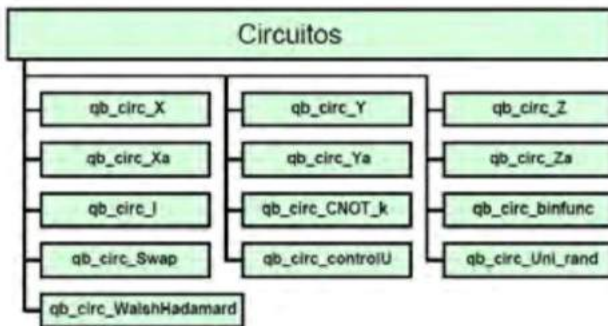


Figura 7. Funciones para circuitos cuánticos.

- *M = qb_circ_X(cant_qubits)*. Crea el operador definido por aplicar la matriz de Pauli σ_X en la cantidad indicada de qubits. Las funciones *qb_circ_Y(cant_qubits)* y *qb_circ_Z(cant_qubits)* realizan los mismo para los operadores σ_Y y σ_Z, respectivamente.
- *M = qb_circ_Xa(cant qubits, par)*. Generaliza la función anterior permitiendo la creación de rotaciones continuas en el eje indicado. Las funciones *qb_circ_Ya(cant_qubits, par))* y *qb_circ_Za(cant_qubits, par)* funcionan en forma análoga.
- *M = qb_circ_I(cant_qubits)*. Crea un operador identidad con la cantidad de qubits indicada.
- *M = qb_circ_CNOT_k(k)*. Crea una compuerta del tipo *CNOT* generalizada, o sea, con rotación continua desde la identidad hasta la *CNOT*.
- *M = qb_circ_binfunc(Mfunc)*. Crea un operador que crea el operador que realiza las funciones binarias indicadas por la matriz *Mfunc*.
- *M = qb_circ_Swap()*. Crea un operador para intercambiar dos qubits cuánticos.
- *MU = qb_circ_controlU(M [, cant_control])*. Crea una compuerta con una transformación unitaria *M* controlada por la cantidad de qubits.
- *M = qb_circ_Uni_rand(cant_qb)*. Crea un operador unitario al azar.
- *M = qb_circ_WalshHadamard(cant qubits)*. Crea un operador *Hadamard* para varios qubits.

4.3. Funciones para medidas cuánticas

En la figura 8, se ilustran las rutinas para la implementación de medidas cuánticas. Utilizan el prefijo *qb_mt*.

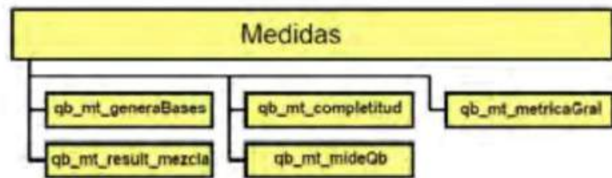


Figura 8. Funciones para medidas cuánticas.

- *[mat_med] = qb_mt_generaBases(vect_dire)*. Dado un vector *vect_dire* (una dirección dada en un espacio de Hilbert) esta función construye una base ortonormal del espacio vectorial complejo en que contenga a este vector normalizado. Devuelve una lista conteniendo los proyectores en cada uno de los subespacios asociados a los vectores que conforman la base.
- *res = qb_mt_completitud(matriz _med, dim _st)*. Controla que un conjunto de operadores de medida cumpla la relación de completitud.
- *[prob, result] = qb_mt_metricaGral(st _in, matriz_ med, ind_med, [,typ, cant])*. Dado un conjunto de medidas cualquiera que cumpla la relación de completitud esta función permite utilizar estos operadores para obtener la probabilidad asociada a cada uno de ellos, o algunos, y el estado colapsado que se obtendría una vez realizada la medida. Un aspecto a destacar es que la creación de los operadores de medida depende del usuario, permitiendo que se implemente cualquier tipo de medida y no únicamente las medidas proyectivas. Por ejemplo, dados los operadores que describen una medida POVM ésta puede simularse utilizando esta rutina.
- *st_out = qb_mt_result_mezcla(st_in, matriz_med, vect)*. Realiza una medida y devuelve como resultado un estado mezcla en el formato *II* o *III*, según sea la entrada. El estado de entrada a este sistema debe de contener un único estado para que efectivamente la salida pueda escribirse como una mezcla los formatos mencionados.
- *[result, st_out] = qb_mt_mideQb(st_in, in_qb, dire, vect)*. Esta función simula la medida de uno de los qubits que conforman al estado, permitiendo elegir

la base en la cual se realiza la medida. A diferencia de las funciones anteriores, en ésta el resultado que se obtiene es el estado colapsado, es decir, se realiza un sorteo aleatorio y se elige uno de entre los dos posibles resultados.

4.4. Funciones para errores cuánticos

En esta biblioteca, se encuentran implementadas rutinas que posibilitan la generación de errores de dos tipos. Los primeros son errores para sistemas cerrados, siendo generado mediante giros ocasionados por una transformación unitaria. La segunda clase de errores modelan la interacción con el entorno. Las rutinas existentes son indicadas en la figura 9 y utilizan el prefijo *qb_cerror*.

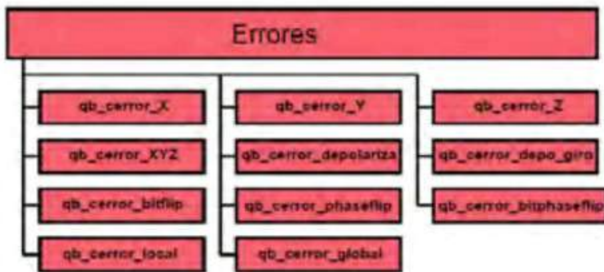


Figura 9. Funciones para errores cuánticos.

- *MError = qb_cerror_X(dim, ind_qb, param)*. Crea una compuerta que ocasiona un error con un giro arbitrario respecto al eje *X* en cada uno de los qubits del sistema, en forma independiente, indicados por *ind_qb*. Los demás qubits no son afectados. Las funciones *qb_cerror_Y* y *qb_cerror_Z* operan en forma análoga.
- *MError = qb_cerror_XYZ(dim, ind_qb, pars, cetype)*. Más general que las anteriores. Crea un compuerta para un estado de *dim* qubits, que genera errores continuos con rotaciones según combinaciones de los operadores *I*, *X*, *Y* , *Z* en *n* qubits al mismo tiempo (indicados en *ind_qb*).
- *[Comp, st_o] = qb_cerror_depolariza(st, ind_qb, param [, tipo_est])*. Crea los operadores para la simulación de un canal de despolarización. Se indican los qubits que será afectados y la probabilidad de mantener el estado. Si el parámetro *st* indica la dimensión del espacio total retorna los operadores. En caso de indicar un estado cuántico, retorna el resultado de la operación.
- *[Comp,st_o] = qb_cerror_bitflip(st, ind_qb, param [, tipo_est])*. Genera los operadores para un error del tipo *BitFlip*. Al igual que en la rutina anterior, si se pasa un estado cuántico se devuelve el estado resultante. Las rutinas *qb_cerror_phaseflip* y *qb_cerror_bitphaseflip* son análogas, utilizando los respectivos operadores.
- *[Comp, st_o] = qb_cerror_depo giro(st, ind qb, pa- ram [, tipo_est])*. Esta función pretende generalizar un tipo de error cualquiera sobre un qubit, combinando el error de despolarización con un error unitario cualquiera sobre los qubits indicados por *ind_ qb*.
- *[Comp, st _o] = qb_cerror_local(st, ind_qb, param [, tipo_est])*. Simula errores de compuerta. Dado un estado de entrada y los qubits a los cuales se quiere afectar, se hace interactuar a cada uno con otro qubit del entono a través de una transformación unitaria aleatoria en $\mathcal{H}^2$. Luego mediante el uso de la traza parcial se descartan todos los qubits correspondientes al entorno y se obtiene un estado de la misma dimensión que el anterior, pero afectado por el ruido.
- *[Comp, st_o] = qb_cerror_global(st, ind_qb, param [, tipo_est])*. Simula una interacción genérica con el ambiente. Dado un estado de entrada, se lo hace interactuar con el entorno que es modelado con tantos qubits como se quiera a través de una transformación unitaria cualquiera del espacio producto sistema/ambiente. Luego con una traza parcial se descarta al ambiente y se obtiene un nuevo estado.

4.5. Funciones utilitarias

En esta biblioteca, existen rutinas implementadas para facilitar las realizaciones de operaciones entre operadores, entre estados, y entre operadores y estados. Como existen diferentes formas de representar un estado, o un ensamble de estados, las operaciones que se quieran realizar dependen del formato en que se encuentren. Luego resulta útil tener rutinas que hagan estas operaciones transparentes. Otra clase de rutinas existente son las rutinas que son sirven de base para la realización de varias de las rutinas de otras bibliotecas. Estas rutinas llevan el prefijo *qb_ut* y se ilustran en la figura 10.

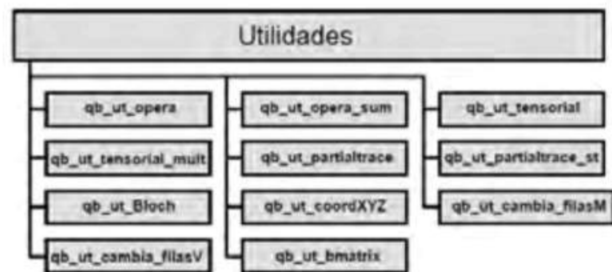


Figura 10. Funciones utilitarias.

- *st_out = qb_ut_opera(st_in, C [, tipo_est])*. Como hay distintos formatos para los estados, esta función se encarga de hacer transparente al usuario la aplicación de un operador *C* a un estado, o ensamble de estados.
- *st_out = qb_ut_opera_sum(st_in, M_list [, tipo_est])*. Similar a la anterior, pero para realizar operaciones cuánticas con operadores de Krauss indicados en la lista *M_list* (Operator-Sum Representation).
- *st = qb_ut_tensorial(comp_1, comp_2 [, tipo_est_1, tipo_est_2)*. Partiendo de dos estados componentes, devuelve el sistema compuesto de ambos estados.
- *st = qb_ut_tensorial_mult(comp_1, comp_2,*

comp_3, ..., comp_n). Similar al anterior, pero realiza el producto tensorial de varios elementos a la vez.

- *dPT = qb_ut_partialtrace(dM, ind)*. Realiza el operador traza parcial para matrices.
- *st_out = qb_ut_partialtrace_st(st_in, ind [, tipo_est])*. Realiza el operador traza parcial para estados.
- *qb_ut_Bloch(st, param [, tipo_est])*. Dibuja la esfera de Bloch con la representación de ensambles de estados de 1 qubit.
- *coordXYZ = qb_ut_coordXYZ(sts [, tipo_est])*. Calcula las coordenadas cartesianas asociadas a cada estado de 1 qubit. Es necesaria para la función qb_ut_Bloch.
- *NC = qb_ut_cambia_filasM(C, i, j)*. Crea una descripción matricial de un operador, o de una matriz de densidad, con los qubits i y j intercambiados.
- *st_out = qb_ut_cambia filasV(st_in, q1, q2)*. Análoga a la anterior, pero para estados con descripción matricial.
- *Mb = qb_ut_bmatrix(n)*. Rutina auxiliar para la creación de todos los números binarios, ordenados por fila, hasta 2n.

5. Ejemplo

Desde los inicios de la computación cuántica, se ha realizado un gran esfuerzo para que los circuitos cuánticos puedan funcionar correctamente aun en presencia de errores [5][6][7][8]. Una idea es codificar los estados cuánticos en sistemas con una mayor dimensión de forma de crear redundancia que permita la detección y corrección de errores, o sea, la utilización de correctores [9][10][11]. En este ejemplo, se analizará la capacidad de corrección de un código cuántico utilizando simulación.

5.1. Código de Shor

El código cuántico de 9 qubits propuesto por Peter Shor [12] es un código capaz de corregir cualquier tipo de error en un solo qubit. La figura 11 ilustra la implementación del código con el circuito codificador (a la izquierda del canal) y el corrector/decodificador (a la derecha).

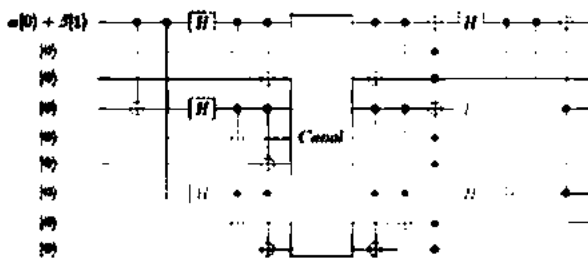


Figura 11. Circuito para el código de Shor.

Las palabras de código son:

$$|0\rangle \rightarrow |0_L\rangle \equiv \frac{(|000\rangle + |111\rangle)(|000\rangle + |111\rangle)(|000\rangle + |111\rangle)}{\sqrt{8}} \quad (1)$$

$$|1\rangle \rightarrow |1_L\rangle \equiv \frac{(|000\rangle - |111\rangle)(|000\rangle - |111\rangle)(|000\rangle - |111\rangle)}{\sqrt{8}} \quad (2)$$

5.2. Experiencia

Para mostrar un caso de uso típico del simulador, se propone estudiar qué ocurre cuando más de un qubit se ve afectado por alguna clase de ruido, y cómo evoluciona el estado al ser reintroducido múltiple veces al canal.

Para esto, se toma un estado aleatorio de un qubit, se lo codifica y se lo somete a un ruido unitario cualquiera en los qubits 1 y 4 del estado codificado. Se decodifica el estado y se toma traza parcial para obtener el estado a la salida del canal. Con este estado resultante, se repite la experiencia 200 veces, es decir, se lo codifica nuevamente, se lo afecta por el mismo error, se lo decodifica y se toma traza parcial.

A continuación, se muestra el funcionamiento del simulador en la construcción de las compuertas de codificación y en la evolución de los estados a través del canal.

5.2.1 *Compuertas de codificación y decodificación*

Para crear la matriz de codificación se implementa la rutina *qb_shor_compEncod* utilizando las rutinas del simulador como sigue:

Algoritmo 1 *Compuertas de codificación y decodificación*

```
function M = qb_shor_compEncod()
// Se crea una compuerta CNOT
CN = qb_circ_CNOT_k(1);
//Se realiza el producto tensorial entre el
// CNOT y la Identidad de dimension 7.
 C0 = qb_ut_tensorial(CN,qb_circ_I(7));
//Se establece que es a los qubits 4 y 7 se les aplica
//un NOT, con el qubit 1 como control. Por esto se
// intercambia el orden de los qubits 2 por 4 y 2 por 7.
C1a = qb_ut_cambia_filasM(C0,2,4);
C1a = sparse(C1a);
C1b = qb_ut cambia_filasM(C0,2,7);
C1b = sparse(C1b);
//Componinendo ambas se obtiene la
// primera etapa de Shor
C1 = C1b*C1a;
// Se aplica compuertas de Hadamard
// en qubits 1,4,7
C2a = qb_circ_WalshHadamard(1);
C2b = sparse(eye(4,4));
C2 = qb_ut_tensorial_mult(C2a,
   C2b,C2a,C2b,C2a,C2b);
//Se crea de manera analoga a la
// primera etapa la correccion en X.
C3a = sparse(CN.*.eye(2,2));
C3b = qb_ut_tensorial mult(C3a,
    C3a,C3a);
C3c = qb_ut_cambia_filasM(C3a, 2,3);
C3c = qb_ut_tensorial_mult(C3c,
    C3c,C3c);
C3 = C3c*C3b;
//Compuerta final
M = sparse(C3*C2*C1);
```

Para crear la matriz de decodificación se tiene una rutina similar llamada *qb shor compDecod*.

5.2.2 *Simulación del canal*

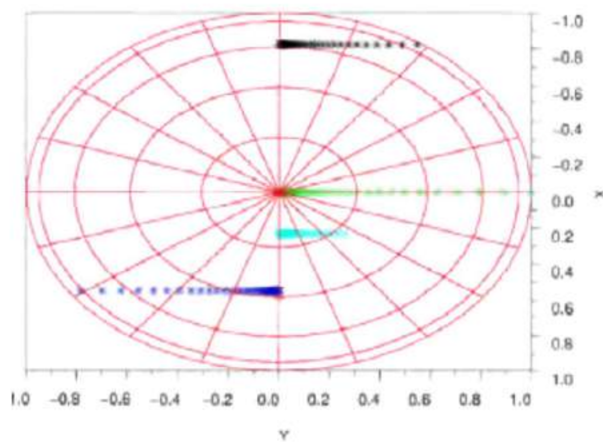
Para la simulación de la acumulación del error en la evolución del estado se ejecuta el siguiente archivo:

Algoritmo 2 *Simulación del canal*

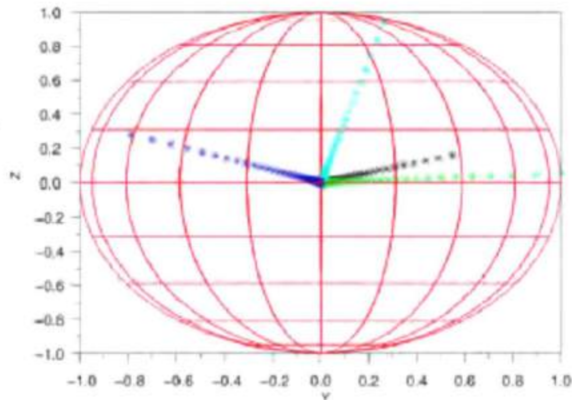
```
// Codigo para la simulacion de la evolucion
// de errores utilizando el codigo de Shor

//Se crea un estado de un qubit
// puro aleatorio.
st_in = qb_st_rand (1, 1, 1);
// Se crean las 8 ancillas auxiliares
//para codificar el estado
st_id8 = qb_st_state(0,8);
//Se convierte el estado al tipo 4
st_ev = qb_st_conver(st_in,4);
//Se crea la matriz que simula un error
//unitario cualquiera que afecta los
//qubits 1 y 4 de la palabra codificada.
M_er = qb_cerror_XYZ(9,[1,4],rand(1,4));
//Se itera 200 veces el procedimiento
for i = 1:200
  //Se toma el qubit del que se quiere
  //simular su pasaje por el canal
  st_prog = qb_st_extractst(st_ev,i);
  //Codifico al estado
  st_aux = qb_ut_tensorial(st_prog,st_id8);
  st_cod = qb_ut_opera(st_aux,M_encod);
  //Se lo afecta por el error
  st_canal = qb_ut_opera(st_cod,M_er);
  //Se decodifica el estado que llega
  //al otro lado del canal
  st_dec = qb_ut_opera(st_canal,M_decod);
  //Tomando traza parcial se obtine el
  //subsistema que corresponde al qubit
  //que se simula su paso por el canal.
  st_par=qb_ut_partialtrace_st( st_dec,1);
  //Se agrega el estado al conjunto
  //de resultados para volver a
  //pasarlo por el canal.
  st_ev = qb_st_agregast(st_ev,st_par);
end

//Se grafican los estados
//obtenidos en la evolucion
qb_ut_Bloch(st_ev,[1,6,6]);
```

5.3. Resultados de la experiencia

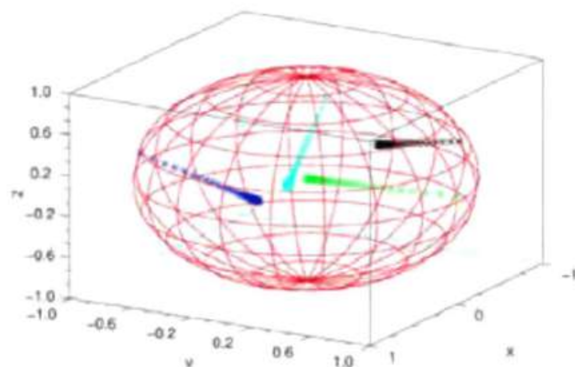
La experiencia se realiza con cuatro estados iniciales aleatorios diferentes. Las gráficas que se obtienen muestran en la esfera de Bloch cómo evolucionan estos estados a medida que son reingresados al canal.



(a) Vista de la esfera de Bloch desde el eje Z.



(b) Vista de la esfera de Bloch desde el eje X.



(c) Vista en perspectiva.

Figura 12. Diferentes vistas en la esfera de Bloch de la evolución de los estados considerados.

Lo que se encuentra es que frente a un error como el propuesto, donde los qubits que se ven afectados son fijos, el código de Shor puede preservar cierta parte de información del estado. La coordenada "X" del estado (coordenada de Bloch) no se ve afectada por el ruido. Esto, en principio, podría dar una idea de cómo codificar la información cuántica de tal manera que ésta sea más robusta frente al ruido.

Más allá del propio experimento, la intención en este artículo es mostrar la potencia del simulador y sus posibilidades a la hora de manipular estados y simular distintos escenarios posibles.

6. Conclusiones y trabajos futuros

En este trabajo, se ha presentado un simulador en el ambiente *Scilab*, diseñado con especial énfasis para que las funcionalidades básicas no dependan del tipo de representación elegida para un estado cuántico.

Es importante resaltar que una amplia cantidad de rutinas fueron desarrolladas con parámetros opcionales que evitan el cálculo reiterado, de modo de incrementar la eficiencia del cómputo.

La forma modular propuesta permite la ampliación del simulador manteniendo la estructura básica del mismo, en donde nuevas funciones se pueden agregar a las bibliotecas existentes o, en ausencia de una biblioteca adecuada, se puede implementar una nueva.

Actualmente, se está trabajando sobre un nuevo simulador que incorporará nuevas bibliotecas de funciones para *Correlaciones cuánticas*, *Isotropía de errores* y otras.

Referencias bibliográficas

[1] R. P. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6-7):467–488, 1982.

[2] D. Deutsch. Quantum theory, the church-turing principle and the universal quantum computer. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 400(1818):97–117, 1985.

[3] G. Gomez, C. Bunks, J.-P. Chancelier, F. Delebecque, M. Goursat, R. Nikoukhah, and S. Steer. *Engineering and Scientific Computing with Scilab*. Birkhauser, 1999.

[4] E. Fonseca de Oliveira, A. L. & Buksman. Simulación de errores cuánticos en el ambiente scilab. Technical report, Universidad ORT Uruguay, 2007.

[5] A. R. Calderbank and Peter W. Shor. Good quantum error-correcting codes exist. *Phys. Rev. A*, 54:1098–1105, Aug 1996.

[6] E. Knill, R. Laflamme, and W. H. Zurek. Resilient quantum computation: error models and thresholds. *Pro. R. Soc. A*, 454(1969):365–384, 1998.

[7] Daniel Gottesman. Theory of fault-tolerant quantum computation. *Phys. Rev. A*, 57:127–137, Jan 1998.

[8] J. Preskill. Reliable quantum computers. *Pro. R. Soc. A*, 454(1969):385–410, 1998.

[9] Raymond Laflamme, Cesar Miquel, Juan Pablo Paz, and Wojciech Hubert Zurek. Perfect quantum error correcting code. *Phys. Rev. Lett.*, 77:198–201, Jul 1996.

[10] D. Gottesman. *Stabilizer codes and quantum error correction*. PhD thesis, California Institute of Technology, 1997.

[11] Emanuel Knill, Raymond Laflamme, and Lorenza Viola. Theory of quantum error correction for general noise. *Phys. Rev. Lett.*, 84:2525–2528, Mar 2000.

[12] Peter W. Shor. Scheme for reducing decoherence in quantum computer memory. *Phys. Rev. A*, 52:R2493–R2496, Oct 1995.