

Títol: Llenguatges de Programació Quàntica

Volum: I

Alumne: Eric Marcos Pitarch

Director/Ponent: Lluís Ametller

Departament: FEN

DADES DEL PROJECTE

Títol del Projecte: Llenguatges de Programació Quàntica

Nom de l'estudiant: Eric Marcos Pitarch

Titulació: Enginyeria Informàtica

Crèdits: 37,5

Director/Ponent: Lluís Ametller Congost

Departament: FEN

MEMBRES DEL TRIBUNAL *(nom i signatura)*

President: Gemma Sese Castel

Vocal: Josep Ramón Herrero

Secretari:

QUALIFICACIÓ

Qualificació numèrica:

Qualificació descriptiva:

Data:

Llenguatges de Programació Quàntica

Eric Marcos Pitarch

Juny 2009

Índex

1.Introducció	8
2.Motivació	13
3.Objectius	16
3.1.Objectius personals.....	16
3.2.Objectius tècnics	16
3.3.Objectius tècnics opcionals	17
3.4.Objectius de gestió del projecte.....	17
4.Introducció a la computació quàntica	19
4.1.State Space Postulate	20
4.2.Evolution Postulate	22
4.3.Measurement Postulate.....	24
4.4.Composition of Systems Postulate	25
4.5.Representació gràfica d'un qubit: Esfera de Bloch	29
4.6.Computació reversible	31
4.7.Framework de portes quàntiques	33
5.Algorismes quàntics	35
5.1.Algorisme de Deutsch	38
5.2.Algorisme de Deutsch-Josza.....	39
5.3.Algorisme de Shor	39
6.Computació quàntica i complexitat	43
7.Llenguatges de programació quàntica	45
7.1.Introducció	45
7.2. Structured Quantum Programming.....	47
7.3. Quantum Programming Languages	48
7.4. Toward an architecture for quantum programming	48
7.5.Propostes de llenguatges de programació quàntica	50
7.5.1.qGCL.....	50
7.5.2.QCL.....	51
7.5.3.QML.....	52
7.5.4.QPL i cQPL	52

7.5.5.LanQ	53
8.PyQu.....	55
8.1.Definició	55
8.2.Decisions sobre tecnologies usades	56
8.2.1.Python	56
8.2.2.Llibreries de simulació quàntica	59
8.2.3.Q++.....	59
8.2.4.libquantum.....	60
8.2.5.SWIG	60
8.3.Especificació	60
8.3.1.Creació i gestió de registres quàntics	60
8.3.1.1.Creació d'un registre quàntic:	62
8.3.1.2.Creació de subregistres	62
8.3.1.3.Composició de registres	63
8.3.1.4.Mètodes dels registres (i subregistres).....	64
8.3.2.Operadors	66
8.3.3.Definir subrutines i funcions	72
8.4.Implementació	73
8.4.1.Entendre i estendre Python	73
8.4.1.1.Tipus de dades en Python	74
8.4.1.1.1. <i>Dynamic Typing</i>	74
8.4.1.1.2. <i>L'objecte PyObject i altres estructures de dades</i>	75
8.4.1.2.Passar paràmetres de Python a C i viceversa	79
8.4.1.3.Gestió de la memòria	80
8.4.2.Definir el mòdul.....	81
8.4.3.Simulació de la computació quàntica utilitzant libquantum	82
8.4.3.1.Implementació de subregistres	82
8.4.3.2.Implementació d'operadors unitaris i mesures	84
8.4.4Diferències entre la implemetació i l'especificació.....	87
8.4.5.Distribució del mòdul	88
9.Planificació del projecte	90
9.1.Planificació inicial	90
9.2.Desviació de la planificació inicial	91
10.Gestió del projecte.....	93

11.Futur del projecte	94
12.Conclusions	95
Bibliografia i referències	97

1.Introducció

Se suposa que el Projecte Final de Carrera és la culminació dels estudis superiors. L'última parada d'un viatge de cinc anys. Una empresa on l'estudiant té l'oportunitat i la responsabilitat de demostrar (i de demostrar-se) que està preparat per a carregar amb el títol d'Enginyer, i ell decideix com ho vol fer. Se suposa que, per a tal tasca, l'estudiant ha de posar en comú tots els coneixements i competències adquirits a la universitat; bé, si no tots perquè és impossible, sí una bona mostra d'ells.

Crec que aquest és el sentit del PFC, i, si bé penso que s'ha de tenir present en tot moment, és sobretot a l'inici i al final del projecte on aquesta reflexió té una incidència especial en les decisions que es prenen i en la valoració de la feina feta, respectivament.

Al principi, quan encara no sabia de què volia fer el projecte, em preguntava quina era l'habilitat més important que havia adquirit durant la carrera. La resposta era clara: la capacitat de solucionar problemes, un concepte intencionadament abstracte. Ho considero un tret distintiu en els enginyers, ja siguin d'industrials, de telecomunicacions, d'informàtica... En general, la feina d'un enginyer es pot resumir en: plantejament d'un problema (o necessitat), anàlisi del problema, plantejament i anàlisi de diferents solucions, elecció i implantació de la solució adequada. Sovint es creu que el que s'aprèn en una enginyeria són un conjunt de destreses tècniques (dominar llenguatges de programació, tecnologies, protocols, paradigmes, procediments...), però en realitat és molt més que això. Òbviament, és imprescindible conèixer un ampli ventall de tecnologies per poder aplicar una solució a nivell pràctic, però per sí soles no serveixen per a fer un bon plantejament d'un problema. És més, moltes vegades (i aquest és un

altre fet important que he après durant la carrera) no cal ni conèixer a fons moltes tecnologies, sinó saber que existeixen (inclús intuir que existeixen!).

Això em porta a parlar d'una altra habilitat fonamental que he adquirit aquests anys: la capacitat d'aprendre molt ràpidament. I aquí sí que m'estic referint concretament a tecnologies de la informació, les quals evolucionen extraordinàriament de pressa... i queden desfasades encara més ràpid. Això ens obliga als informàtics a viure en un aprenentatge continu, a adaptar-nos a noves maneres de fer constantment. Crec que els informàtics ens ho hem agafat, resignats, com un *modus vivendi*; dubto que existeixi algun altre ofici que requereixi aquests nivells de reciclatge dels que s'hi dediquen. Personalment he adoptat, de forma inconscient (però sospito que impulsat per la manera de fer a la FIB), una postura davant d'aquesta situació: quedar-me amb la visió global de tot el que aprenc i mantenir-me actualitzat de les novetats tecnològiques, en el sentit de saber que existeixen. Per això deia que no cal dominar totes les tecnologies. Quan tinc un nou projecte intento visualitzar el màxim de solucions possibles diferents i si alguna d'elles implica l'aprenentatge d'una nova tecnologia es converteix en l'excusa perfecte per aplicar-la.

El lector es deu estar preguntant quina relació té tot el que estic dient amb el títol del treball. La relació és que, a l'hora de triar el projecte, volia que tot això quedés reflectit. A part, volia que tingués una component molt important de recerca, més que res per una qüestió de motivació personal. Així és com se'm va acudir fer alguna cosa relacionada amb la computació quàntica (cq). Després de donar-hi moltes voltes vaig decidir fer un estudi sobre les diferents propostes de llenguatges d'alt nivell orientats a la programació quàntica i aportar el meu granet de sorra en forma d'un mòdul per Python que permeti implementar algorismes quàntics en aquest llenguatge.

En aquest punt cal dir que els algorismes quàntics només es poden executar en un computador quàntic, un dispositiu teòric que aprofitaria certes propietats sorprenents del món de les partícules elementals (descrites per la mecànica quàntica) per a realitzar càlculs molt més ràpidament que un computador convencional. Com he dit abans, un computador quàntic és un dispositiu teòric i, tot i que existeixen alguns prototips, es creu que encara falten moltes dècades per a poder construir un computador quàntic realment útil. Per aquest motiu totes les implementacions de llenguatges de programació quàntica es basen en simulacions, la qual cosa vol dir que realment no estem aprofitant cap de les propietats físiques que abans he comentat i no estem obtenint cap avantatge en termes de velocitat de càlcul.

Llavors la pregunta que sorgeix de forma natural és: de què serveix fer un simulador d'un computador quàntic si no ens proporciona cap avantatge en els computadores actuals? Aquesta és una pregunta que ha planat sobre el meu projecte des del principi i per a la qual no tinc una resposta clara. Una de les raons que es pot donar és el de donar suport a la teoria sobre els algorismes quàntics. Quan es veuen programes reals, tot i que basats en simulacions, que produeixen els resultats predits per la teoria, sembla que aquesta es veu reforçada.

Una altra raó és a nivell pedagògic ja que, malgrat comprendre la teoria és imprescindible per entendre el funcionament de la computació quàntica, exemples pràctics, que permetin fer càlculs pas a pas i consultar l'estat de la màquina en qualsevol punt, que permetin jugar i fer proves amb els diferents elements de la cq i que permetin veure els resultats amb nombres concrets, són importantíssims per visualitzar el que està passant i entendre millor els detalls dels algorismes. Jo mateix em puc posar com a exemple d'això; fins que no vaig fer la implementació de l'algorisme de Shor realment no vaig entendre tots els problemes implicats en l'algorisme i com aquest els solventava.

Per últim, la gran utilitat que tenen els llenguatges d'alt nivell de programació quàntica és que són independents del maquinari. Això significa que els mateixos algorismes que puguem programar ara (com per exemple la implementació que he fet de l'algorisme de Shor) funcionaran tal qual quan existeixin computadors quàntics. L'únic que s'haurà de canviar és la capa intermèdia que, en comptes de cridar funcions d'una llibreria de simulació, executarà instruccions de baix nivell d'una computadora quàntica.

A nivell personal (i egoista, perquè no dir-ho), he utilitzat aquest projecte com a excusa per aprendre moltes coses que tenia ganes d'aprendre. Lligat amb el que deia al principi, jo sabia que existia la possibilitat d'estendre el llenguatge Python amb mòduls escrits en C/C++, però no tenia ni idea de com es feia, inclús amb prou feines havia programat en Python abans. En quant a la computació quàntica, vaig fer una assignatura de lliure elecció fa tres anys que em va fer despertar el interès per aquest tema i el PFC es presentava com una oportunitat ideal per aprofundir-hi.

Tot plegat em va fer pensar que era un tema que reunia les característiques que volia per al meu PFC, tant a nivell de continguts com a volum de feina. I haig de dir que estic molt satisfet de tot el que he après i fins on he arribat, i que el projecte segueix obert i sens dubte el continuaré desenvolupant en els propers mesos.

Per acabar aquesta introducció només dir que aquesta memòria es divideix en tres parts. La primera és la base teòrica del projecte, on presento els aspectes fonamentals en que es basa la cq i que són imprescindibles per entendre els algorismes quàntics, una mostra dels quals es presenten a continuació. Per acabar aquesta primera part hi ha un capítol dedicat a la complexitat de la cq i com es relaciona amb les classes de complexitat 'clàssiques'.

La segona part és l'estudi que he fet sobre els llenguatges de programació quàntica; què és el que s'espera d'un llenguatge de programació quàntica, quines aproximacions teòriques s'han plantejat i un anàlisi d'algunes implementacions reals basades en simulacions.

La tercera i última part és pròpiament la del llenguatge dissenyat i implementat per mi. Especificació, decisions, dificultats trobades, explicació del codi, gestió del projecte, etc.

2.Motivació

En aquest apartat vull explicar perquè he triat fer un projecte sobre llenguatges de programació quàntica.

La base teòrica en que se sustenta la computació quàntica, la mecànica quàntica, data dels anys 20 del segle passat, tot i que la computació quàntica com a disciplina independent no es va començar a desenvolupar fins a finals dels 80, principis dels 90, quan certes persones van començar a adonar-se que un hipotètic computador basat en les regles de la mecànica quàntica permetria realitzar càlculs molt més ràpidament que els computadores clàssics basats en el model de màquina de Turing.

Aquests indicis van ser corroborats l'any 1994 de forma contundent, quan Peter Shor va publicar un article on presentava un algorisme capaç de factoritzar un nombre en un temps polinòmic. Això va ser una notícia molt remarcable en el món de la teoria de la computació ja que fins ara no s'ha trobat cap algorisme clàssic que pugui factoritzar un nombre en un temps que no sigui exponencial. A més els sistemes d'encryptació RSA àmpliament utilitzats a tot el món es basen precisament en la dificultat de factoritzar nombres molt grans que són producte de dos nombres primers. Per tant l'algorisme de Shor era capaç de trencar una clau RSA, tot i que no es podia aplicar fins que no es construís una computadora quàntica, clar. Això va provocar que la notícia tingués una dimensió mediàtica molt important i que moltes mirades (i diners) es dirigissin cap aquest camp.

Per a la tranquil·litat del lector només vull apuntar que la criptografia quàntica, una disciplina germana de la computació quàntica, aporta nous sistemes i protocols d'encryptació 100% segurs, i està molt més avançada que la seva germana, fins al punt

que ja hi ha sistemes d'encryptació quàntica comercials i que s'utilitzen amb èxit en el món real.

Un parell d'anys més tard, el 1996, un altre algorisme sorprenent va ser publicat, en aquest cas per Lov Grover. Aquest algorisme permet realitzar una cerca desordenada en un temps $O(\sqrt{N})$. Això significa una millora quadràtica respecte als algorismes de cerca per força bruta que solucionen problemes NP-Complets.

Aquests resultats extraordinaris van provocar certa eufòria i moltes ments es van bolcar en descobrir nous algorismes quàntics. Tot i aquest prometedor inici, sembla que la computació quàntica s'ha quedat estancada o, si més no, avança molt a poc a poc, ja que 15 anys després de l'algorisme de Shor, no ha sortit cap resultat comparable en importància.

Abans de començar el treball mantenia la hipòtesi de que una possible causa de l'avanç lent de la computació quàntica era el llenguatge estrany en què se solen expressar els algorismes quàntics. Com veurem més endavant, normalment els algorismes quàntics s'expressen mitjançant diagrames amb portes quàntiques, que representen operadors que s'apliquen a vectors d'entrada. Crec que és necessari un llenguatge més formal i compacte, que incrementi el nivell d'abstracció, que s'allunyi de la gran varietat de formalismes usats en aquest camp (notació de Dirac, matrius, operadors...) i de la terminologia pròpia dels físics, i que s'apropi a la representació clàssica d'un algorisme des del punt de vista d'un enginyer informàtic.

D'altra banda, fer una extensió per Python també suposa una motivació des del punt de vista tècnic, ja que serà el primer simulador de computació quàntica que es faci en aquest llenguatge. Python és un llenguatge en alça dins de la comunitat informàtica i em va estranyar que no existís cap mòdul de programació quàntica, ja que hi ha moltes

llibries de simulació per a altres llenguatges (la majoria en C) i Python gaudeix d'una comunitat de desenvolupadors enorme.

I per últim, molts dels simuladors que he vist estan mal documentats o són difícils d'aprendre. Vull fer alguna cosa que serveixi per entendre com funciona la computació quàntica, que pugui ser utilitzat com una eina de suport per a l'aprenentatge d'aquest camp des del punt de vista d'un enginyer informàtic.

3.Objectius

A continuació llistaré els objectius que em vaig proposar en començar el projecte. Els he dividit en objectius personals, objectius tècnics, objectius tècnics opcionals i objectius de gestió de projecte.

3.1.Objectius personals

- Aprofundir en els coneixements de computació quàntica, tant des del vessant de la física com des del vessant de la teoria de la computació, que em situïn en un punt de "pre-recerca".
- Tenir un mapa de les diferents aproximacions a llenguatges de programació quàntica que s'han fet en els últims anys. Distingir els pros/contres i limitacions d'aquestes aproximacions.
- Investigar els avenços recents relacionats amb complexitat de la CQ i els seus límits des del punt de vista de la teoria de la computació.
- Investigar les diferents propostes de simulació de CQ en ordinadors actuals.

3.2.Objectius tècnics

- Especificar i dissenyar un llenguatge de programació capaç d'implementar algorismes quàntics. El llenguatge ha de ser "llegible", intuïtiu i fàcil d'aprendre. No ha de trencar amb l'estil de Python.
- Fer un estudi sobre les actuals llibreries de simulació de CQ escrites en /C++ i escollir la més apropiada per a ser utilitzada en aquest projecte.
- Crear un mòdul en C/C++ per estendre el llenguatge Python. Amb aquest mòdul l'interpret de Python podrà executar programes escrits en el llenguatge propi.

- Com a objectiu bàsic concret, l'aplicació, en el moment de la finalització del projecte, ha de ser capaç d'executar amb èxit la implementació de l'algorisme de Shor en el llenguatge propi.
- Crear una bona documentació (a part de la memòria del projecte) del llenguatge creat, però no només una referència del llenguatge (tipus de dades, funcions, operacions, etc) sinó com està enfocat i perquè, què pot fer i què no, com es pot millorar, tutorials i exemples de com utilitzar-lo...

3.3.Objectius tècnics opcionals

- Millores en el rendiment de la simulació (el rendiment no és un objectiu primordial en aquest projecte).
- Permetre la creació de *threads* per a programació en paral·lel.

3.4.Objectius de gestió del projecte

- Cercar altres parts que puguin estar interessades en aquest projecte (tant a nivell d'usuari, com a nivell de desenvolupador), en altres universitats, grups de recerca, individuals... l'objectiu és que el projecte no quedi mort després de la presentació.
- Crear un bon sistema de gestió del projecte que em permeti tenir-ho tot centralitzat i accessible des de la xarxa. Necessitats bàsiques:
 - Subversion: manteniment automàtic de les versions del codi.
 - Accessible a tothom online.
 - Wiki: On anar documentant tot el procés, problemes trobats, solucions, tasques pendents... també links interessants a articles, projectes relacionats, etc.
 - Blog: on aniré informant de l'estat del projecte, idees noves...

Tot això farà que el projecte tingui presència a la xarxa i ha de facilitar a trobar parts interessades. En cas de trobar altres desenvolupadors interessats en participar, un altre objectiu seria gestionar un equip de programadors via Internet (assignació de responsabilitats, repartiment i planificació de feina...).

4.Introducció a la computació quàntica

En aquest capítol introduiré els conceptes bàsics per entendre la computació quàntica però sense donar definicions massa formals, simplement perquè aquest no és l'objectiu del projecte. Hi ha moltíssima bibliografia disponible si el lector està interessat en els detalls. Donaré una visió global i entraré en els detalls que considero claus per poder seguir la resta de la memòria i entendre els algorismes quàntics.

La mecànica quàntica és el marc sobre el que es desenvolupen les teories físiques modernes i no pas, com molta gent es pensa, una teoria física completa de la realitat. En paraules de Michael Nielsen:

"[Quantum mechanics] is, simply, a set of four mathematical postulates. That's all it is – four surprisingly simple postulates which lay the ground rules for our description of the world".

Així doncs descriuré els quatre postulats matemàtics fonamentals de la mecànica quàntica i veurem què se'n deriva de cadascun. Haig d'advertir que les conseqüències físiques que deriven d'aquests postulats moltes vegades semblen impossibles o antinaturals, en el sentit que descriuen fenòmens que no estem acostumats a veure en la nostra realitat. I justament són aquests fenòmens estranys (fins i tot esotèrics) els que doten a la computació quàntica d'una capacitat extraordinària per a realitzar càlculs molt ràpidament. També vull apuntar que, des del punt de vista dels informàtics, "no ens ha d'importar" que signifiquen tots aquests fenòmens a nivell físic. Simplement, els físics ens donen un conjunt de regles matemàtiques a partir de les quals veurem com podem calcular coses i desenvolupar algorismes. Conceptes com "dualitat ona-partícula"

o el gat d'Shrodinger que està "viu i mort alhora" que sempre s'associen a la mecànica quàntica no seran discutits en aquest estudi. També en queden fora discussions sobre possibles implementacions físiques d'un computador quàntic.

Així doncs, ens creurem als físics quan ens diuen que és possible construir una màquina que segueixi aquestes normes i nosaltres ens dedicarem a "jugar" amb ella.

Sense més dilació em dispenso a presentar el primer postulat de la mecànica quàntica.

4.1.State Space Postulate

El primer postulat ens diu que l'estat d'un sistema quàntic ve descrit per un vector \vec{v} unitari en un espai de Hilbert H complex.

Un espai de Hilbert és essencialment un espai vectorial on hi ha definit un *inner product* $\langle v, w \rangle$ (producte escalar) per cada parella de vectors. En el nostre cas, aquest espai vectorial és de nombres complexos, la qual cosa vol dir que un vector d'aquest espai és una tupla de nombres complexos.

Recordem la definició de producte escalar per a vectors complexos:

$$\langle v, w \rangle = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} \cdot \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix} = (v_1^* v_2^* \cdots v_n^*) \cdot \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix} = \sum_{i=1}^n v_i^* w_i$$

Per exemple, l'espai vectorial amb el que estem més familiaritzats és l'espai euclidià de les tres dimensions, on tot vector es pot descriure amb tres nombres reals (a,b,c) que són els que multipliquen els vectors de la base. Anàlogament, en un espai de Hilbert complex de tres dimensions, tot vector es pot descriure amb tres nombres complexos (α, β, γ) .

Per tant, l'estat d'un sistema quàntic és una combinació lineal dels elements de la base d'un espai de Hilbert on els coeficients de les bases són nombres complexos.

No ens ha de preocupar massa la definició d'espai de Hilbert, en realitat és només una generalització d'un espai euclidià. El que sí és important és que els vectors que descriuen estats quàntics han de ser unitaris, això vol dir que han de tenir longitud (norma) 1. Una altra manera de dir-ho és que el producte escalar d'un vector amb sí mateix ha de ser 1.

$$\|v\| = \langle v, v \rangle = 1$$

Perfecte, ara ja sabem com es descriu l'estat d'un sistema quàntic, però... què és un sistema quàntic? Bé, començaré per descriure el sistema quàntic més simple i més endavant veurem com es generen sistemes més complexos.

El sistema quàntic més simple és el qubit, de la mateixa manera que la mínima unitat d'informació és el bit. Un qubit és una idealització matemàtica; físicament pot ser l'espín d'un electró, la polarització d'un fotó, etc, però això no ens ha de preocupar, de la mateixa manera que tampoc ens preocupa si un bit està codificat en una bombeta encesa o apagada o en la tensió elèctrica d'un cable.

L'espai de Hilbert associat a un qubit té dues dimensions, o sigui que l'estat d'un qubit es descriu amb dos nombres complexos. Normalment es fa servir la base formada pels vectors $\begin{pmatrix} 1 \\ 0 \end{pmatrix} = |0\rangle$ i $\begin{pmatrix} 0 \\ 1 \end{pmatrix} = |1\rangle$, i ens referim a ella com la base computacional. La notació $|0\rangle$ simplement ens indica que aquest element es tracta d'un vector. En la literatura sobre computació quàntica es diu que $|0\rangle$ és un *ket*, i és la manera com s'escriuen els vectors en la notació de Dirac. Òbviament, $|0\rangle$ i $|1\rangle$ són l'equivalent del qubit al 0 i l'1 del bit clàssic.

Així doncs, l'estat d'un qubit és un vector

$$|\Psi\rangle = \alpha|0\rangle + \beta|1\rangle = \alpha \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \beta \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$$

Aquí ja veiem la primera de les propietats "màgiques" que abans comentava. El qubit, a diferència del bit, pot estar en 0, en 1 o en una superposició (superposició és només una manera de dir combinació lineal) arbitrària de 1 i 0.

A partir d'ara faré servir els termes vector i estat indistintament.

4.2. Evolution Postulate

Aquest postulat ens descriu com evoluciona en el temps un sistema quàntic. Concretament ens diu que l'evolució d'un sistema quàntic tancat ve descrita per un operador unitari U . Un operador unitari és un operador lineal, això vol dir que només cal descriure com evolucionen els estats de la base per saber com evoluciona qualsevol estat, i que preserva les normes unitàries dels vectors sobre els quals s'aplica.

Formalment, una transformació lineal sobre un espai vectorial H és una transformació $T : H \rightarrow H$ de l'espai vectorial a ell mateix (mapeja vectors d' H a vectors d' H). Aquests operadors s'escriuen com matrius quadrades $n \times n$ on n és el nombre de dimensions de l'espai de Hilbert sobre el que s'aplica l'operador. Per exemple, en un espai de dos dimensions:

$$U = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

Un operador U és unitari si $U^\dagger = U^{-1}$, on U^\dagger és l'adjunt o *Hermitean conjugate* de U i es defineix com:

$$U^\dagger = (U^*)^T = \begin{pmatrix} a^* & c^* \\ b^* & d^* \end{pmatrix}$$

I U^{-1} és la matriu inversa tal que $UU^{-1} = I$

L'estat $|\Psi\rangle$ després d'aplicar-li l'operador U és el resultat de multiplicar la matriu U per l'esquerra a $|\Psi\rangle$:

$$|\Psi'\rangle = U|\Psi\rangle$$

En la literatura sobre computació quàntica sol referir-se als operadors com a portes quàntiques, ja que de forma similar a com actuen les portes lògiques clàssiques, les portes quàntiques modifiquen l'estat del sistema, així que a partir d'ara faré servir els termes operador, transformació i porta indistintament.

Aquí tenim algunes de les portes més importants que actuen sobre un qubit i com actuen sobre els elements de la base computacional:

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad I|0\rangle = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |0\rangle, \quad I|1\rangle = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle$$

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad X|0\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle, \quad X|1\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |0\rangle$$

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \quad Y|0\rangle = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} i \\ 0 \end{bmatrix} = i|0\rangle,$$

$$Y|1\rangle = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ -i \end{bmatrix} = -i|1\rangle$$

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \quad Z|0\rangle = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |0\rangle,$$

$$Z|1\rangle = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ -1 \end{bmatrix} = -|1\rangle$$

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad H|0\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \frac{|0\rangle + |1\rangle}{\sqrt{2}},$$

$$H|1\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \frac{|0\rangle - |1\rangle}{\sqrt{2}}$$

4.3.Measurement Postulate

Aquest postulat ens diu que no és possible saber l'estat d'un sistema quàntic. Recordem que els postulats anteriors parlàvem de sistemes tancats, és a dir, aïllats d'altres sistemes. Llavors, quan intentem esbrinar l'estat del sistema el que fem és interactuar amb ell i el sistema deixa d'estar aïllat. Aquest postulat diu que mesurar implica pertorbar el sistema, concretament ens diu que si mesurem un qubit $|\Psi\rangle$ el resultat de la mesura serà o bé $|0\rangle$ o bé $|1\rangle$ encara que abans estigués en una superposició $\alpha|0\rangle + \beta|1\rangle$. És més, obtindrem $|0\rangle$ amb una probabilitat $\|\alpha\|^2$ i $|1\rangle$ amb una probabilitat $\|\beta\|^2$. Recordem del primer postulat que els vectors que representen un estat quàntic són unitaris, per tant, la probabilitat total sempre és 1:

$$\|\alpha\|^2 + \|\beta\|^2 = 1$$

En realitat aquest postulat és una mica més complicat, ja que podem fer mesures respecte diferents bases de forma que el resultat de la mesura és un dels elements de la base, però en aquest projecte només faré servir mesures respecte la base computacional, o sigui que, d'ara endavant, donaré per suposat que el resultat de la mesura d'un sistema quàntic és un element de la base computacional (0 o 1 per a un sistema d'un qubit).

Tornant al que deia al principi, el *measurement postulate* ens diu que no podem esbrinar els coeficients α i β d'un qubit ja que la superposició d'estats només es dona quan el sistema està aïllat, també ens diu que aquests coeficients determinen les probabilitats d'obtenir 0 o 1 en fer una mesura. Per últim ens diu que, un cop realitzada la mesura, l'estat del sistema resta en l'estat mesurat, és a dir, si en mesurar un qubit el resultat és 0, l'estat del qubit resta $|0\rangle$ i successives mesures seguiran donant 0.

4.4.Composition of Systems Postulate

Fins ara hem vist quines implicacions tenen els tres primers postulats sobre el sistema més simple possible: un qubit. El quart postulat ens diu com descriure sistemes complexos a partir de sistemes més simples. Formalment, un sistema quàntic compost de dos sistemes més simples té associat un espai de Hilbert H que és el producte tensorial dels espais de Hilbert H_1 i H_2 associats als subsistemes que el componen:

$$H = H_1 \otimes H_2$$

En general, l'operació de producte tensorial es defineix així:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \otimes \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} & a_{12} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} \\ a_{21} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} & a_{22} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} \end{pmatrix}$$

$$= \begin{pmatrix} a_{11}b_{11} & a_{11}b_{12} & a_{12}b_{11} & a_{12}b_{12} \\ a_{11}b_{21} & a_{11}b_{22} & a_{12}b_{21} & a_{12}b_{22} \\ a_{21}b_{11} & a_{21}b_{12} & a_{22}b_{11} & a_{22}b_{12} \\ a_{21}b_{21} & a_{21}b_{22} & a_{22}b_{21} & a_{22}b_{22} \end{pmatrix}$$

L'espai vectorial resultant d'un producte tensorial de dos subespais es defineix a partir de les bases dels subespais, concretament, els elements de la nova base són el producte tensorial de totes les parelles de les bases dels subespais.

Per exemple un sistema de dos qubits és representat per un espai de Hilbert que és el producte tensorial de dos espais de Hilbert associats a un qubit. Abans hem vist que l'espai vectorial d'un qubit té dues dimensions i que els dos elements de la seva base computacional son $|0\rangle$ i $|1\rangle$. Per tant els elements de la base de l'espai d'un sistema de dos qubits són 4 nous vectors:

$$|0\rangle \otimes |0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ 0 \begin{pmatrix} 1 \\ 0 \end{pmatrix} \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$|0\rangle \otimes |1\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \begin{pmatrix} 0 \\ 1 \end{pmatrix} \\ 0 \begin{pmatrix} 0 \\ 1 \end{pmatrix} \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$$

$$|1\rangle \otimes |0\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ 1 \begin{pmatrix} 1 \\ 0 \end{pmatrix} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

$$|1\rangle \otimes |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \begin{pmatrix} 0 \\ 1 \end{pmatrix} \\ 1 \begin{pmatrix} 0 \\ 1 \end{pmatrix} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

Per tant, l'estat d'aquest sistema es descriu amb un vector de quatre nombres complexos. Fent un abús de la notació podem dir que aquestes expressions son equivalents:

$$|0\rangle \otimes |0\rangle = |0\rangle |0\rangle = |00\rangle = |0\rangle = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad |0\rangle \otimes |1\rangle = |0\rangle |1\rangle = |01\rangle = |1\rangle = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$$

$$|1\rangle \otimes |0\rangle = |1\rangle |0\rangle = |10\rangle = |2\rangle = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \quad |1\rangle \otimes |1\rangle = |1\rangle |1\rangle = |11\rangle = |3\rangle = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

És fàcil veure que el nombre de dimensions d de l'espai de Hilbert d'un sistema creix exponencialment amb el nombre de qubits n del sistema: $d = 2^n$. Més endavant comentaré les implicacions que això comporta, però per ara dir que aquest és el principal motiu de que sigui tan costós simular un sistema quàntic en un ordinador convencional.

En el segon postulat hem vist que els operadors que actuaven sobre un qubit eren matrius de 2×2 , doncs bé, els operadors que actuen sobre sistemes d' n qubits són matrius de $2^n \times 2^n$. Els operadors sovint també es poden descriure a partir de la composició d'operadors més simples a través del producte tensorial. Per exemple, si en

un sistema de 2 qubits li volem aplicar una X al primer, quin és operador que descriu l'evolució del sistema? Doncs seria el producte tensorial de $X \otimes I$ ja que "no fer res" a un qubit és com aplicar-li la transformació indentitat.

Aquesta situació es pot formalitzar de la següent manera:

$$U_1|\Psi_1\rangle \otimes U_2|\Psi_2\rangle = (U_1 \otimes U_2)(|\Psi_1\rangle \otimes |\Psi_2\rangle)$$

D'aquesta expressió també podem deduir que l'estat $|\Psi\rangle$ d'un sistema compost és el producte tensorial dels estats dels subsistemes. Per exemple en un sistema de 2 qubits, si el primer esta en l'estat $|\Psi_1\rangle = \alpha|0\rangle + \beta|1\rangle$ i el segon en l'estat $|\Psi_2\rangle = \gamma|0\rangle + \delta|1\rangle$, llavors l'estat del sistema compost per tots dos qubits és:

$$|\Psi_1\rangle \otimes |\Psi_2\rangle = \alpha\gamma|00\rangle + \alpha\delta|01\rangle + \beta\gamma|10\rangle + \beta\delta|11\rangle$$

Tot i que hem vist com podem generar operadors i estats a partir d'elements més simples, és important fer notar que no tots els estats ni tots els operadors de sistemes compostos es poden generar així. Per exemple, una de les portes més importants és la CNOT o *controlled*-NOT que opera sobre dos qubits. Aquesta porta aplica una NOT (la transformació X que hem vist anteriorment) al segon qubit si el primer està a 1 i el deixa tal qual si el primer qubit està a 0. Podem escriure la matriu que defineix la CNOT fàcilment a partir de l'acció sobre els elements de la base:

$$|00\rangle \rightarrow |00\rangle; |01\rangle \rightarrow |01\rangle; |10\rangle \rightarrow |11\rangle; |11\rangle \rightarrow |10\rangle$$

$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Aquesta matriu no es pot obtenir a partir del producte tensorial de dues matrius 2×2 .

De manera semblant, no tots els estats es poden descriure com la composició d'estats més simples. Per exemple el següent estat en un sistema de dos qubits és perfectament vàlid:

$$\frac{|00\rangle + |11\rangle}{\sqrt{2}}$$

Però no podem descriure aquest estat a partir de la combinació de dos estats d'un qubit: $\alpha\gamma|00\rangle + \alpha\delta|01\rangle + \beta\gamma|10\rangle + \beta\delta|11\rangle$. Podem veure que si els coeficients de $|01\rangle$ i $|10\rangle$ són zero llavors $|00\rangle$ o $|11\rangle$ també tindran coeficient zero.

Ara que hem vist sistemes de més d'un qubit donaré una definició més general del *measurement postulate*. En general l'estat d'un sistema de n qubits es pot escriure com:

$$|\Psi\rangle = \sum_{i=0}^{2^n} \alpha_i |i\rangle$$

Llavors, en mesurar l'estat obtindrem i amb una probabilitat $P(i) = \|\alpha_i\|^2$

Podem realitzar mesures parcials, és a dir, podem fer una mesura només sobre un subconjunt dels qubits del sistema. Com he explicat abans, en la mecànica quàntica mesurar significa modificar el sistema, i si fem una mesura parcial podem estar canviant els coeficients de l'estat, o sigui, canviant les probabilitats dels qubits que encara no hem mesurat. Ho veurem més clar en aquest exemple: suposem que tenim un sistema de dos qubits en l'estat que hem vist abans:

$$\frac{|00\rangle + |11\rangle}{\sqrt{2}}$$

Podem veure que tenim un 50% de probabilitats de mesurar 00 i un altre 50% de mesurar 11, per tant les probabilitats dels dos qubits per separat són d'un 50% de mesurar 0 i un 50% de mesurar 1. En el moment en que fem la mesura d'un dels dos

qubits, però, les probabilitats de l'altre qubit han canviat. Si el resultat de la mesura és 0 llavors l'estat del sistema és $|00\rangle$ i per tant les probabilitats del qubit que encara no hem mesurat són d'un 100% que surti 0. Si mesurem 1, llavors l'estat del sistema és $|11\rangle$ i segur que mesurem 1 en el segon qubit. Aquesta mena d'estats es diu que són *entangled states*.

En canvi, si tenim aquest estat:

$$\frac{|00\rangle + |01\rangle + |10\rangle + |11\rangle}{2}$$

en un principi cada qubit pot donar 0 o 1 amb un 50% de probabilitats, igual que abans. La diferència és que després de la mesura l'altre qubit segueix tenint un 50% de treure 0 o 1 independentment del resultat de la primera mesura.

4.5.Representació gràfica d'un qubit: Esfera de Bloch

Com hem vist abans, l'estat d'un qubit és un vector en un espai de dos dimensions sobre els nombres complexos. Per tant, per descriure'l són necessaris dos nombres complexos, o sigui quatre nombres reals, la qual cosa fa difícil visualitzar l'estat d'un qubit. Per sort aquests nombres complexos no poden ser arbitraris, hi ha un parell de condicions que ens permetran reduir els graus de llibertat i obtenir una representació gràfica d'un qubit.

En general, un nombre complex és un vector en el pla dels nombres complexos i es pot escriure com: ae^{ib} on a és el mòdul i e^{ib} és la direcció del vector (b és l'angle del vector). Així doncs l'estat d'un qubit és:

$$|\Psi\rangle = a_0 e^{ib_0} |0\rangle + a_1 e^{ib_1} |1\rangle$$

Segons el primer postulat s'ha de complir:

$$\|a_0 e^{ib_0}\|^2 + \|a_1 e^{ib_1}\|^2 = a_0^2 + a_1^2 = 1$$

Fent ús de la identitat $\cos(\theta)^2 + \sin(\theta)^2 = 1$, l'estat d'un qubit ens queda com

$$|\Psi\rangle = e^{ib_0} \cos \theta |0\rangle + e^{ib_1} \sin \theta |1\rangle$$

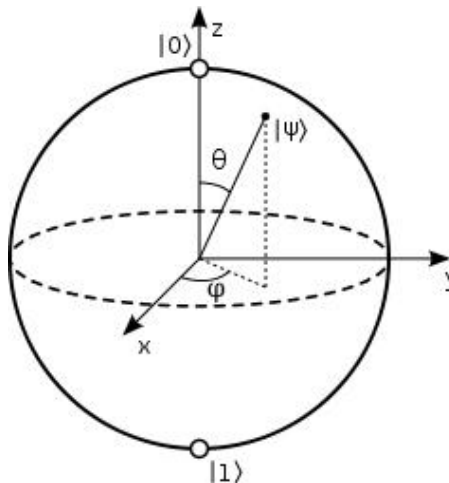
Podem reescriure la formula anterior com:

$$|\Psi\rangle = e^{ib_0} \cos \theta |0\rangle + e^{i(b_0+\varphi)} \sin \theta |1\rangle = (\cos \theta |0\rangle + e^{i\varphi} \sin \theta |1\rangle) e^{ib_0}$$

El terme e^{ib_0} té mòdul 1 i es pot interpretar com una fase global del sistema. El més important és veure que aquest factor no influeix per res en l'estat a l'hora d'aplicar operadors unitaris (sempre anirem "arrossegant" aquesta fase global) i tampoc a l'hora de mesurar el sistema ja que multiplica els mòduls dels coeficients de les bases per 1. En resum, podem prescindir d'aquest terme i escriure l'estat d'un qubit com:

$$|\Psi\rangle = \cos \theta |0\rangle + e^{i\varphi} \sin \theta |1\rangle$$

Ara ens queda l'estat del qubit en funció de dos angles i podem representar-ho gràficament com un punt en una esfera de radi 1 anomenada esfera de Bloch:



Per a que la fórmula quadri amb el dibuix l'hem de reescriure com

$$|\Psi\rangle = \cos\left(\frac{\theta}{2}\right) |0\rangle + e^{i\varphi} \sin\left(\frac{\theta}{2}\right) |1\rangle$$

Aquesta representació és molt útil per veure com actuen les portes quàntiques d'un qubit, per exemple, les portes X, Y, Z (que juntament amb I conformen un conjunt anomenat portes de Pauli) que hem vist en el postulat 2 corresponen a rotacions al voltant dels eixos x, y i z de l'esfera de Bloch.

És important remarcar, però, que l'esfera de Bloch només ens serveix per representar sistemes d'un qubit. No podem representar un sistema d' n qubits amb n esferes de Bloch ja que no té sentit parlar dels coeficients de $|0\rangle$ i $|1\rangle$ per a cada qubit perquè, com hem vist abans, un sistema d' n qubits no té perquè ser una composició de sistemes més simples, i menys una composició d' n sistemes d'un qubit.

4.6. Computació reversible

Recordem que quan he descrit el *Evolution Postulate* hem vist que un operador és unitari si $U^\dagger = U^{-1}$, la qual cosa implica que $UU^\dagger = I$, el que dit en paraules significa que per cada operador unitari U en tenim un altre que aplica la transformació inversa U^\dagger que, si l'apliquem després de U , deixa l'estat igual que al principi.

Donat l'estat inicial d'un sistema i una sèrie d'operadors unitaris obtenim un estat final que, si li apliquem els inversos dels operadors en l'ordre invers, obtenim l'estat inicial. I això sempre es pot fer, per aquest motiu es diu que la computació quàntica és reversible. Fixem-nos que en general això no és possible en la computació clàssica, per exemple, donada la sortida d'una operació AND no tenim cap manera de saber el valor de les dues variables inicials.

També cal remarcar que en la computació quàntica la mida de l'entrada sempre és igual la mida de la sortida.

Moltes vegades ens interessarà poder computar una funció clàssica sobre un registre quàntic, o sigui, aplicar un operador U_f que ens faci la transformació:

$$U_f: |x\rangle \rightarrow |f(x)\rangle$$

Però això en general no és possible ja que $f(x)$ pot ser no reversible, cosa que passarà per exemple (tot i que no exclusivament) quan $f(x)$ sigui injectiva.

Fixem-nos també que no hi ha cap operador unitari que ens permeti fer cap d'aquestes transformacions:

$$|x\rangle|y\rangle \rightarrow |x\rangle|x\rangle$$

$$|x\rangle \rightarrow |0\rangle$$

És a dir, un qubit no es pot copiar i tampoc es pot esborrar.

Per tant, veiem que el tractament de la informació en la computació quàntica és més delicat que en la computació clàssica. Tot i això, fent servir alguns trucs i una mica més de recursos (en temps i espai), serem capaços de computar qualsevol funció clàssica en el model quàntic.

Més formalment, donada una implementació clàssica (per exemple amb portes AND, OR i NOT) que computa una funció $f(x)$, és possible construir una implementació reversible equivalent amb una quantitat polinòmica de recursos addicionals. En general, la implementació reversible serà de la forma:

$$|x\rangle|y\rangle \rightarrow |x\rangle|y \oplus f(x)\rangle$$

Per tant, si l'entrada y la inicialitzem a zero tenim

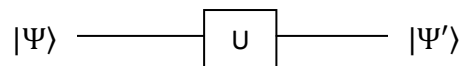
$$|x\rangle|0\rangle \rightarrow |x\rangle|f(x)\rangle$$

O sigui que ja tenim una manera de computar funcions clàssiques en el context reversible de la computació quàntica.

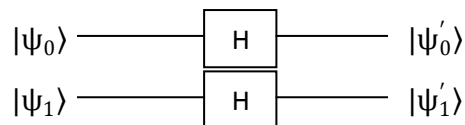
4.7. Framework de portes quàntiques

Per descriure els algorismes quàntics és molt habitual utilitzar un llenguatge visual que és el model de circuits. En aquests diagrames els qubits d'un sistema s'ordenen verticalment i es representen mitjançant cables horitzontals. Els operadors (o portes) es representen com caixes que es col·loquen sobre els cables (qubits) sobre els que actua la porta. L'evolució del sistema avança cap a la dreta, col·locant-se a l'esquerra l'estat inicial i a la dreta l'estat final. Les portes s'apliquen en l'ordre que apareixen avançant des de l'estat inicial fins al final.

Per exemple, la situació $|\Psi'\rangle = U|\Psi\rangle$ l'expressarem de la següent manera amb portes quàntiques:

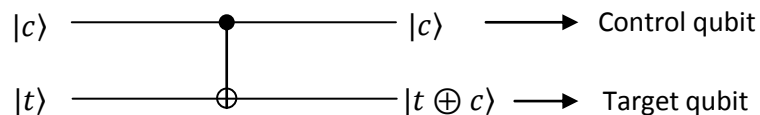


Un altre exemple:

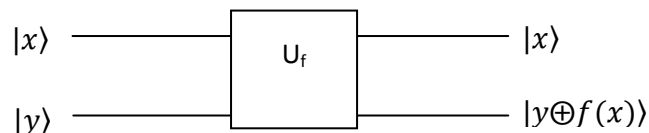


Aquest diagrama equival a l'expressió $(H \otimes H)(|\psi_0\rangle|\psi_1\rangle) = |\psi'_0\rangle|\psi'_1\rangle$

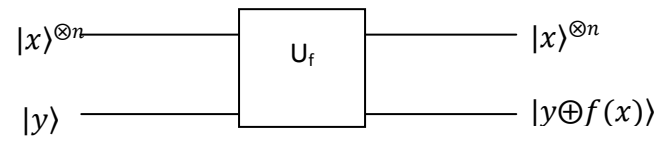
Les portes control tenen una representació especial, per exemple, la CNOT:



Les portes que implementen una funció $f(x)$ s'expressen així:



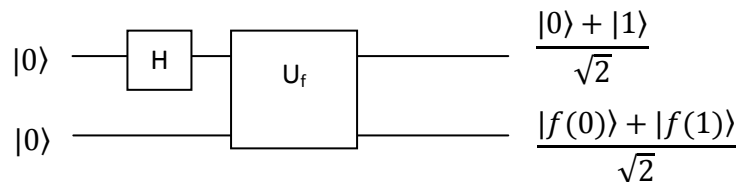
De vegades s'agrupen uns quants qubits en un sol "cable" per formar una mena de registre quàntic, per exemple si la funció $f(x)$ accepta n bits d'entrada:



5. Algorismes quàntics

Ara que hem vist com funciona la computació quàntica anem a veure com podem utilitzar-la per solucionar problemes.

La computació quàntica té un gran poder i una gran desgràcia. El gran poder són en realitat dos. Per una banda tenim que l'estat d'un sistema d' n qubits és un vector de 2^n elements (els coeficients de les bases de l'espai de Hilbert associat de dimensió 2^n), és a dir que d'alguna manera, amb n qubits podem emmagatzemar $O(2^n)$ bits d'informació. I per una altra banda, quan apliquem una transformació sobre un sistema estem actuant simultàniament sobre tots els 2^n elements (els coeficients de les bases es modifiquen "en paral·lel"). Fixem-nos en el següent diagrama:



Aquest algorisme calcula totes les $f(x)$ en un sol pas de computació!

La gran desgràcia de la computació quàntica és el *Measurement Postulate*, el qual ens diu que la mesura destrueix la superposició. Si ens fixem en l'exemple d'abans al final el segon qubit és una superposició de totes les $f(x)$, però quan el mesurem simplement obtindrem alguna de les $f(x)$ i l'estat "col·lapsarà" a $|f(x)\rangle$, és a dir, que tot el que havíem aconseguit amb la superposició d'estats finalment no serveix de res perquè en la mesura perdem gran part de la informació.

La clau per obtenir informació útil és fer alguna transformació més abans de la mesura que faci que la probabilitat de la mesura s'acumuli en valors que ens aportin informació (gràcies al fenomen d'interferència quàntica).

Per exemple, si recordem com actua la porta hadamard sobre un qubit:

$$H|0\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \quad H|1\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}}$$

Veiem que d'alguna forma està codificant informació de l'estat entrant en la fase de l'estat sortint:

$$H|x\rangle = \frac{1}{\sqrt{2}}(|0\rangle + (-1)^x|1\rangle) = \frac{1}{\sqrt{2}} \sum_{z \in \{0,1\}} (-1)^{xz} |z\rangle$$

Una altra de les característiques de la porta hadamard és que és la seva pròpia inversa, és a dir $HH = I$, per tant, si tornem a aplicar la hadamard a l'estat anterior tornarem a tenir l'estat $|x\rangle$ que teníem al principi. D'aquesta manera podem pensar que la porta hadamard ha descodificat informació que teníem guardada en la fase.

Una altra tècnica per a codificar informació en les fases és el que es coneix com phase kick-back. Veiem-ne un exemple concret i després veurem com es pot generalitzar. Imaginem que apliquem una CNOT sobre un sistema de dos qubits on el *control qubit* està en l'estat $|1\rangle$ i el *target qubit* està en l'estat $\frac{|0\rangle - |1\rangle}{\sqrt{2}}$:

$$\begin{aligned} CNOT: |1\rangle \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}} \right) &\rightarrow |1\rangle \left(NOT \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}} \right) \right) \rightarrow |1\rangle \left((-1) \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}} \right) \right) \\ &\rightarrow -|1\rangle \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}} \right) \end{aligned}$$

L'últim pas d'aquest càlcul és conseqüència d'una propietat dels productes tensorials que diu el següent:

$$c(|\psi_1\rangle \otimes |\psi_2\rangle) = (c|\psi_1\rangle) \otimes |\psi_2\rangle = |\psi_1\rangle \otimes (c|\psi_2\rangle)$$

Vegem que passa quan el control qubit està a zero:

$$CNOT: |0\rangle \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}} \right) \rightarrow |0\rangle \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}} \right)$$

Això es pot resumir d'aquesta manera:

$$CNOT: |b\rangle \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}} \right) \rightarrow (-1)^b |b\rangle \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}} \right)$$

Si en comptes d'una CNOT ho generalitzem a una porta U_f qualsevol que implementa una funció $f: \{0,1\} \rightarrow \{0,1\}$ de manera que $U_f: |x\rangle|y\rangle \rightarrow |x\rangle|y \oplus f(x)\rangle$ tenim el següent:

$$\begin{aligned} U_f: |x\rangle \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}} \right) &\rightarrow \left(\frac{U_f|x\rangle|0\rangle - U_f|x\rangle|1\rangle}{\sqrt{2}} \right) \\ &\rightarrow \left(\frac{|x\rangle|0 \oplus f(x)\rangle + |x\rangle|1 \oplus f(x)\rangle}{\sqrt{2}} \right) \\ &\rightarrow |x\rangle \left(\frac{|0 \oplus f(x)\rangle + |1 \oplus f(x)\rangle}{\sqrt{2}} \right) \end{aligned}$$

Analitzem aquesta expressió per casos:

$$f(x) = 0: |x\rangle \left(\frac{|0 \oplus f(x)\rangle + |1 \oplus f(x)\rangle}{\sqrt{2}} \right) = |x\rangle \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}} \right)$$

$$f(x) = 1: |x\rangle \left(\frac{|0 \oplus f(x)\rangle + |1 \oplus f(x)\rangle}{\sqrt{2}} \right) = |x\rangle \left(\frac{|1\rangle - |0\rangle}{\sqrt{2}} \right) = -|x\rangle \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}} \right)$$

Els dos casos només es diferencien per un factor -1 que depèn del valor de $f(x)$.

En una sola expressió ens queda així:

$$U_f: |x\rangle \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}} \right) \rightarrow (-1)^{f(x)} |x\rangle \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}} \right)$$

Tot seguit veurem alguns exemples d'algorismes que utilitzen aquestes tècniques.

5.1. Algorisme de Deutsch

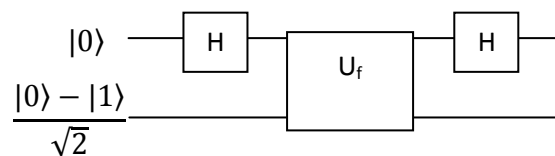
L'algorisme de Deutsch soluciona el següent problema:

Donada una caixa negra que computa una funció desconeguda $f : \{0,1\} \rightarrow \{0,1\}$, determinar si f és constant (totes les $f(x)$ tenen la mateixa sortida, ja sigui 1 o 0) o balancejada (si per exactament la meitat de les entrades obtenim 0 i per l'altra meitat 1).

Fixem-nos que un algorisme clàssic hauria d'avaluar la funció per les dues entrades possibles 0 i 1.

Per contra, l'algorisme de Deutsch utilitza la tècnica de *phase kick-back* posant al qubit entrada una superposició de totes les possibles entrades i amb una sola crida a la funció pot determinar si és constant o balancejada.

El circuit que implementa l'algorisme de Deutsch és aquest:



I ara vegem com evoluciona l'estat del qubit d'entrada pas a pas:

$$\begin{aligned}
 |0\rangle &\rightarrow \left(\frac{|0\rangle + |1\rangle}{\sqrt{2}} \right) \\
 &\rightarrow \left(\frac{(-1)^{f(0)}|0\rangle + (-1)^{f(1)}|1\rangle}{\sqrt{2}} \right) \\
 &\rightarrow \left(\frac{(-1)^{f(0)} \left(\frac{|0\rangle + |1\rangle}{\sqrt{2}} \right) + (-1)^{f(1)} \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}} \right)}{\sqrt{2}} \right) \\
 &= \frac{1}{2} [(-1)^{f(0)} + (-1)^{f(1)}]|0\rangle + [(-1)^{f(0)} - (-1)^{f(1)}]|1\rangle
 \end{aligned}$$

Podem veure com hem aconseguit calcular les imatges de $f(0)$ i $f(1)$ en una sola avaluació de U_f . L'última hadamard d'alguna manera ens descodifica la informació amagada a la fase de manera que si f és balancejada l'estat final és $|1\rangle$ i per tant mesurarem 1 amb un 100% de probabilitats, i si f és constant mesurarem 0 amb un 100% de probabilitats ja que l'estat final serà $|0\rangle$.

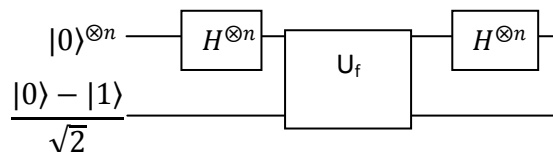
5.2. Algorisme de Deutsch-Josza

Aquest algorisme és una generalització de l'algorisme de Deutsch. Soluciona el mateix problema però en aquest cas la funció f té com a entrada una cadena d' n bits:

$$f : \{0,1\}^n \rightarrow \{0,1\}$$

La gràcia és que aquest algorisme és exponencialment més ràpid que un algorisme clàssic. Mentre que l'algorisme de Deutsch-Josza només necessita avaluar la funció un cop, un algorisme clàssic necessitaria en el cas pitjor $O(2^n)$ *queries* a f .

El circuit que el soluciona és pràcticament igual que el que hem vist abans:



En aquest cas no desenvoluparé els càlculs ja són més frágosos i la idea és exactament la mateixa que abans.

5.3. Algorisme de Shor

L'algorisme de Shor és una de les fites més importants de la computació quàntica. Tot seguit exposaré els conceptes més importants però no el presentaré en profunditat, ja que ocuparia moltíssimes pàgines i considero que no és necessari que ho faci en aquesta memòria. Si el lector està interessat en conèixer els detalls recomano l'article "*Shor's Algorithm for Factoring Large Integers*" de C. Lavor, L.R.U. Manssur, i R. Portugal.

L'algorisme de Shor té una part clàssica i una part quàntica. El que fa la part quàntica és trobar l'ordre d'una funció periòdica del tipus $f(x) = a^x \bmod N$. El que fa la part clàssica de l'algorisme és reduir el problema de factorització al problema de trobar l'ordre d'una funció. Explicaré primer la part clàssica.

Suposem que volem trobar els factors de N , comencem per escollir a l'atzar un nombre x menor que N i calculem el $MCD(x, N)$. Si no ens dona 1 vol dir que hem trobat un factor de N , així doncs l'apuntem i tornem a escollir una altra x . Així fins que trobem una x que sigui coprimeira amb N . Definim el mòdul de x com l'enter positiu r més petit tal que

$$x^r = 1 \bmod N$$

Si r és parell tenim que

$$x^{r/2} = y \bmod N \Rightarrow y^2 = 1 \bmod N \Rightarrow (y + 1)(y - 1) = 0 \bmod N$$

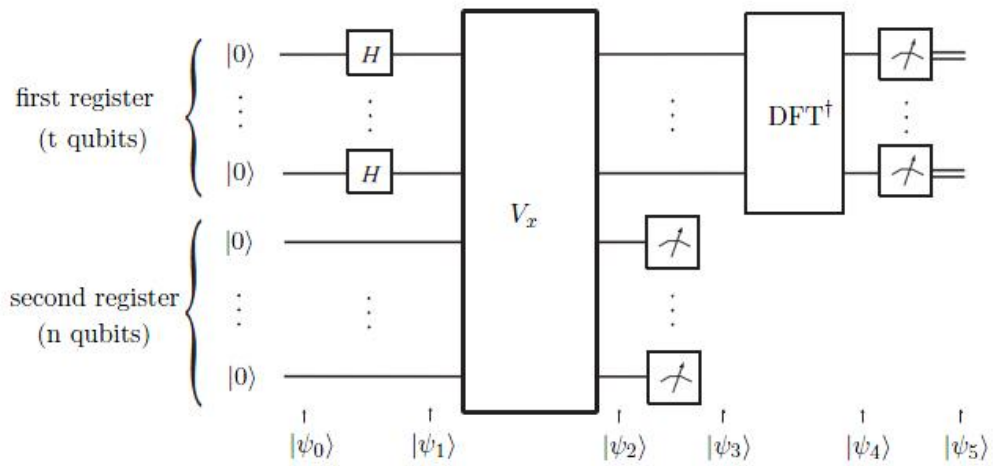
O sigui que N divideix el producte $(y + 1)(y - 1)$. Si $y < N - 1$ tenim que $y - 1 < y + 1 < N$, o sigui que N no pot dividir ni a $(y - 1)$ ni a $(y + 1)$ ja que són més petits que ella. Llavors l'única opció és que tant $y - 1$ com $y + 1$ tenen factors de N . Així, calculant el $MCD(N, y \pm 1)$ obtenim factors de N .

Si alguna de les condicions necessàries no es compleix (que r sigui parell i que y sigui més petit que $N-1$) tornem a començar escollint un nou x a l'atzar.

Ara només ens falta veure la part quàntica de l'algorisme per trobar el període r . Per aquest propòsit haurem d'introduir una nova porta, la Quantum Fourier Transform o QFT:

$$QFT_n : |x\rangle \rightarrow \frac{1}{\sqrt{2^n}} \sum_{y=0}^{2^n-1} e^{2\pi i \frac{xy}{2^n}} |y\rangle$$

A continuació mostro el circuit que implementa l'algorisme per trobar el període r :



Els dos primers passos de l'algorisme ens haurien de ser familiars. Preparem el registre d'entrada en una superposició de tots els valors possibles i apliquem una porta que ens calcula la funció $f(x) = a^x \bmod N$ per a totes les entrades de forma simultània. O sigui que després del segon pas de computació tenim l'estat:

$$|\Psi_2\rangle = \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} |j\rangle |x^j \bmod N\rangle$$

És important adonar-se que en el segon registre només hi ha r valors diferents (ja que $x^j \bmod N$ és una funció periòdica amb període r) i que cada un d'aquests valors agrupa totes les entrades j amb la mateixa imatge. Per això fem la mesura del segon registre en el següent pas. En el punt $|\Psi_3\rangle$ tindrem un valor fix en el segon registre (el valor que haguem mesurat en el pas previ) i en el registre d'entrada hi tindrem un estat periòdic, en el sentit que totes les bases tindran coeficient zero excepte algunes que estaran distribuïdes de forma periòdica amb un període r i un *offset* b_0 .

Llegir el registre d'entrada ara o ens donaria cap informació útil. Abans de mesurar aplicarem la Transformada de Fourier Quàntica, la qual cosa ens deixarà l'estat final

en un nou estat periòdic amb un període proporcional a $1/r$ i sense cap *offset*. Ara si que podem mesurar l'estat i extreure'n informació útil. El que haurem de fer és utilitzar un algorisme per calcular fraccions contínues i obtenir així candidats de r . Un cop tenim un candidat és fàcil comprovar si realment és el període. Si ho és, comprovem que r sigui parell i en traiem els factors de N , sinó, tornem a començar l'algorisme.

L'algorisme de Shor té un 40% de probabilitats de finalitzar amb èxit.

6.Computació quàntica i complexitat

Durant el transcurs del projecte he estudiat de forma superficial la relació entre la computació quàntica i les classes de complexitat clàssiques. En els últims anys s'ha estudiat molt la computació quàntica des del punt de vista de la teoria de la computació per esbrinar quins són els seus límits, però la veritat és que no s'ha pogut demostrar massa cosa.

S'ha definit una nova classe de complexitat anomenada BQP (*bounded error, quantum, polynomial time*) que es defineix com el conjunt de problemes que es poden solucionar en temps polinòmic amb una probabilitat alta, cal dir que tots els algorismes quàntics són intrínsecament probabilístics, tot i que de vegades puguem saber la solució amb una certesa del 100%. Per exemple, la factorització de nombres estaria dins d'aquesta classe.

La relació entre BQP i les classes de complexitat clàssiques, ja siguin probabilístiques (com BPP) o no (P, NP...), no està clara.

Una cosa que sí que està clara és que una computadora quàntica no pot solucionar cap problema que no pugui solucionar una computadora basada en el model de Turing, ja que les computadores clàssiques poden simular sistemes quàntics, per tant, tot i que poden solucionar alguns problemes més ràpidament, les computadores quàntiques no són capaces de solucionar problemes indecidibles com el halting problem.

També se sap que BQP està estrictament dins de PSPACE, però això, tot i posar un sostre més proper, tampoc és cap sorpresa. El repte real de la computació quàntica és saber si serà capaç de solucionar problemes NP-Complets en temps polinòmic, és a dir, si la classe NP està dins de BQP.

Existeix la falsa idea de que la computació quàntica pot solucionar problemes NP-Complets, ja que sempre es parla de que els algorismes quàntics aconseguen una “millora exponencial” a l’hora de solucionar certs problemes i això s’identifica immediatament amb problemes NP-Complets. I és cert que la computació quàntica aconseguix una millora exponencial en alguns problemes, però cap d’aquests problemes pertany la classe dels NP-Complets. La factorització de nombres és un problema que no s’ha demostrat que pertanyi a NP-Complet i tampoc a P.

L’algorisme de Grover és interessant perquè soluciona un problema que caracteritza els problemes NP-Complets. El problema SAT es pot reduir fàcilment al problema de Grover. La qüestió és que “només” aconseguix una millora quadràtica respecte els algorismes clàssics de cerca per força bruta, el que significa que no reduïm la complexitat.

Soc de la opinió que si finalment la computació quàntica no és capaç de solucionar problemes NP-Complets en temps polinòmic serà de molt poca utilitat en el món real.

Es pot pensar que la millora quadràtica que proporciona Grover per als problemes NP-Complets és prou important, però a la pràctica, els computadors quàntics haurien d’arribar a les velocitats de processament dels ordenadors actuals per a que aquest fet deixi de ser una curiositat matemàtica i sigui realment útil.

Tot i que no s’ha demostrat, tot sembla indicar que la computació quàntica no podrà solucionar problemes NP-Complets més ràpidament que l’algorisme de Grover.

7.Llenguatges de programació quàntica

7.1.Introducció

Com hem vist en seccions anteriors, els algorismes quàntics se solen representar com un circuit, on el qubits es representen com "cables" i els operadors com portes que actuen sobre els qubits. Tots segueixen més o menys el mateix esquema: preparar l'estat inicial del computador quàntic, aplicar un conjunt de transformacions i mesurar el sistema.

Tot i que el framework dels circuits quàntics es prou expressiu com per que qualsevol algorisme quàntic es pugui formular d'aquesta manera, és natural preguntar-se si hi ha altres maneres d'expressar-los. Sembla evident que si algun dia s'arriben a construir computadors quàntics escalables, aquests hauran de ser programats en algun llenguatge d'alt nivell. En paraules del propi Deutsch:

"quantum computers raise interesting problems for the design of programming languages"

Abans de començar a parlar de possibles llenguatges vull fer varies observacions. Primer de tot, com ja sabem, els computadors quàntics útils encara trigaran varies dècades en poder ser desenvolupats. Aquest fet, que pot semblar descoratjador, pot ser un avantatge en el sentit que tenim l'oportunitat de desenvolupar una base teòrica sòlida que ens permeti dissenyar els llenguatges abans d'implementar-los, al contrari del que ha passat sovint amb la informàtica "clàssica", on molts cops els llenguatges s'han dissenyat abans (o al mateix temps) que les bases teòriques, provocant algun mal de cap als enginyers.

Una altra incògnita que sorgeix és quina mena d'arquitectura tindrà el hardware d'un computador quàntic. Això fa que no puguem assumir res sobre un possible llenguatge de baix nivell tipus assemblador que doni instruccions concretes al computador quàntic, ja que aquests llenguatges tenen una forta dependència de l'arquitectura de la màquina. Per tant l'únic que podem fer és començar a pensar en llenguatges d'alt nivell ja que son completament independents del hardware.

Una cosa que sembla bastant clara avui dia és que la computació quàntica no substituirà a la clàssica. El més probable és que acabin coexistint en una mateixa màquina, amb una part clàssica i una altra part quàntica, una mena de subsistema controlat per la part clàssica que serà utilitzat per a realitzar els càlculs que la part clàssica no pugui fer eficientment. Això ens porta a pensar que un llenguatge de programació quàntica d'alt nivell ha d'estar integrat en un llenguatge clàssic, la qual cosa comporta un primer dubte en el disseny.

Com hem vist, la computació quàntica té poc a veure amb la computació tradicional; per entendre els algorismes quàntics cal comprendre els postulats de la mecànica quàntica. Així doncs, com poden coexistir maneres de programar amb una base teòrica tan diferent en un sol llenguatge? Com veurem més endavant la solució més generalitzada és encapsular l'acció de la computació quàntica en un objecte especial anomenat registre quàntic sobre el que podrem aplicar operadors unitaris, mesurar-lo, etc. Aquest punt de vista tan "orientat a objectes" té un risc i és que potser no accentua prou les diferències entre la computació quàntica i la clàssica, ja que li dóna un enfocament molt clàssic. Crec que això pot induir a confusió o a l'error de pensar que no cal estudiar la teoria de la mecànica quàntica per entendre un algorisme quàntic.

De totes maneres, el model de circuits, que també és una representació d'un algorisme quàntic però en un llenguatge visual, tampoc ens aporta cap "coneixement" sobre el que

està passant en l'algorisme. Tan sols ens descriu l'estat inicial, una seqüència de transformacions, i un estat final. Quan veiem una caixa que posa QFT sabem que hem d'aplicar la transformada de Fourier quàntica, però no ens indica realment què està fent aquesta porta. Així doncs, el model de portes quàntiques és un model molt abstracte, podríem dir que de molt alt nivell, ja que descriu un procés en forma de llista ordenada de passos a seguir, de la mateixa manera que un algorisme clàssic. Per tant, veiem que la traducció d'un algorisme quàntic representat amb el model de circuits a un llenguatge més proper als llenguatges de programació clàssica és bastant directe, almenys a alt nivell. Tot i això, s'han d'estudiar bé les diferents opcions que es plantegen, veure què es pot fer i què no, quina és la millor manera d'integrar la computació quàntica i la computació clàssica...

Últimament, i sobretot a partir de l'any 2000, han sortit molts articles parlant i discutint sobre possibles aproximacions al problema. Per a fer aquest estudi m'he basat en tres articles que resumeixo a continuació.

7.2. Structured Quantum Programming

"Structured Quantum Programming" de Bernhard Ömer (Vienna University of Technology, 2003) és un article molt complet i totalment autocontingut que fa una extensa introducció a la computació quàntica i dona la seva visió de com haurien de ser els llenguatges de programació quàntics, especificant un llenguatge anomenat QCL que comentaré més endavant i del qual n'he agafat moltes idees.

En aquest article l'autor fa moltíssimes reflexions interessants tot i que és molt difícil de seguir els detalls ja que fa totes les definicions amb un llenguatge matemàtic molt formal. Fa una classificació dels algorismes quàntics en *Deterministic Sequential Algorithms* i *General Probabilistic Algorithms*. Els primers són algorismes que troben la solució amb un 100% de probabilitats d'èxit i per tant són deterministes; un exemple

d'aquests és l'algorisme de Deutsch-Josza. Els *General Probabilistic Algorithms* són els que troben la solució correcta amb certa probabilitat i s'han d'executar diversos cops (típicament amb un nombre constant de cops ja tenim una alta probabilitat) per trobar la solució. També introdueix el concepte de *Finite Quantum Programs* com un autòmat finit que controla el flux d'instruccions d'un computador quàntic i diu que qualsevol algorisme quàntic es pot expressar d'aquesta manera.

7.3. Quantum Programming Languages

"Quantum Programming Languages. Survey and Bibliography" de Simon J Gay (Universitat de Glasgow, 2005). Aquest article m'ha servit per tenir una visió global i històrica dels llenguatges de programació quàntica. L'autor fa un repàs cronològic de les diferents propostes que s'han fet en els últims 20 anys, relacionant noms i dates i classificant les diferents propostes respecte tres criteris: *Programming language design* (classifica les propostes segons el paradigma del llenguatge proposat, bàsicament en llenguatges imperatius i llenguatges funcionals), *semantics* i *compilation*.

Atribueix el primer model abstracte orientat a la computació quàntica a Deutsch, amb la definició d'una màquina de Turing Quàntica l'any 1985.

7.4. Toward an architecture for quantum programming

"Toward an architecture for quantum programming" de S.Bettelli, T.Calarco i L.Serafini (2003).

En aquest article es comença a parlar de l'arquitectura d'un hipotètic computador quàntic com una màquina híbrida amb una part clàssica i una part quàntica, concretament es basen en el model QRAM, on la part quàntica és un subsistema d'un computador clàssic, el qual fa el pre i el post processat de les dades i controla l'execució de la part quàntica.

Proposen un sistema de gestió de la memòria quàntica (els qubits) basat en referències i comptadors de referències de la mateixa manera que funciona la memòria dinàmica en un ordinador actual. M'he basat en algunes de les seves idees per implementar els subregistres en el meu llenguatge.

També fan apunts interessants sobre detalls de la implementació física d'un computador quàntic que podrien afectar a la complexitat real dels algorismes que s'hi executin. Per exemple, resulta que físicament és molt difícil implementar una transformació sobre varis qubits si aquests no estan col·locats de forma contigua. Per tant, si una màquina quàntica no és capaç d'implementar aquestes transformacions, hauria de fer operacions de *swap* entre qubits abans i després de cada operació, la qual cosa alentiria moltíssim el temps d'execució dels algorismes.

Els autors també presenten un desiderata, és a dir un conjunt de característiques que serien desitjables per a un llenguatge de computació quàntica, que presento a continuació ja que ha sigut la meua base a l'hora de prendre certes decisions sobre el meu llenguatge:

Completeness: *the language must be powerful enough to express the quantum circuit model. This means that it must be possible to code every valid quantum algorithm and, conversely, every piece of code must correspond to a valid quantum algorithm.*

Classical extension: *the language must include (i.e. be an extension of) a high level classical computing paradigm in order to integrate quantum computing and classical pre and post-processing with the smallest effort. Ad hoc languages, with a limited implementation of classical primitives and facilities, would inevitably fall behind whenever "standard" programming technologies improve.*

Separability: *the language must keep classical programming and quantum programming separated, in order to be able to move to a classical machine all those computations which do not need, or which do not enjoy any speedup in being executed on, a quantum device.*

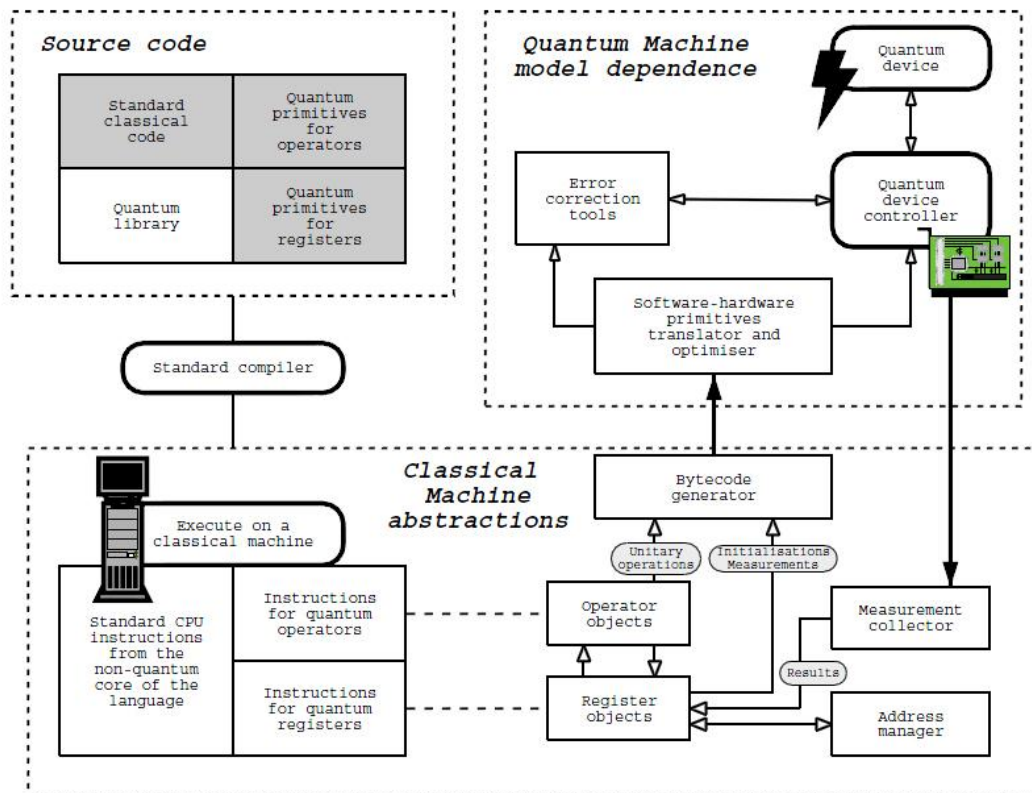
Expressivity: *the language must provide a set of high level constructs which make the process of coding quantum algorithms closer to the programmer's way of thinking and to the pseudo-code modular notation of current research articles. The language must allow an automated scalable procedure for translating and optionally optimising the*

high level code down to a sequence of low level control instructions for quantum machines.

Hardware independence: the language must be independent from the actual hardware implementation of the quantum device which is going to be exploited. This allows “recompilation” of the code for different quantum architectures without the programmer’s intervention.

En els capítols posteriors especifiquen el seu llenguatge Q Language.

En aquest esquema es mostra l’arquitectura que proposen per a un possible computador quàntic:



7.5. Propostes de llenguatges de programació quàntica

7.5.1. qGCL

Aquest llenguatge ha estat desenvolupat per Paolo Zuliani en la seva tesi de doctorat l’any 2001. Està basat en pGCL (n’és una extensió), un llenguatge dissenyat per Dijkstra per a implementar algorismes probabilístics. Segons el propi Zuliani, un programa en qGCL és un programa pGCL que invoca el que ell anomena quantum procedures, que

poden ser de tres tipus: d'inicialització (permet inicialitzar estats directament en una superposició), d'evolució (aplicar operadors), i de finalització (mesura de l'estat final).

No he trobat exemples de codi en aquest llenguatge

7.5.2.QCL

QCL és el llenguatge proposat per Bernhard Ömer (l'autor de l'article *Structured Quantum Programming*) i és probablement el llenguatge quàntic amb la implementació més desenvolupada i completa. Té una sintaxi molt semblant a C però no és una extensió d'aquest llenguatge. QCL té el seu propi intèrpret que utilitza la llibreria de simulació qlib.

A continuació mostro un exemple de codi en QCL:

```
qureg x1[2]; // 2-qubit quantum register x1
qureg x2[2]; // 2-qubit quantum register x2
H(x1); // Hadamard operation on x1
H(x2[1]); // Hadamard operation on the first qubit
of the register x1
```

Un dels desavantatges més importants que crec que té QCL és que no és una extensió d'un llenguatge clàssic. És purament un llenguatge orientat a la computació quàntica al qual se li han afegit alguns elements de programació clàssica com instruccions de control de seqüència, possibilitat de definir subrutines, etc. Crec que això limita molt el llenguatge sobretot pel que fa a pre i post processat de les dades. Tot i això crec que la sintaxi és molt clara i he intentat imitar-la a l'hora de dissenyar el meu llenguatge.

Una altra cosa molt interessant és la possibilitat d'aplicar l'inversa d'una transformació posant-li el signe ! al davant:

```
operator diffuse(qureg q) {
    H(q); // Hadamard Transform
    Not(q); // Invert q
    CPhase(pi,q); // Rotate if q=1111..
    !Not(q); // undo inversion
    !H(q); // undo Hadamard Transform
}
```

7.5.3.QML

Es tracta d'un llenguatge de programació quàntica basat en Haskell (un llenguatge purament funcional) i desenvolupat per Altenkirch i Grattage. Algunes de les característiques interessants del llenguatge és que permet copiar qubits de forma nativa. Com hem vist al principi, copiar i esborrar qubits són operacions prohibides en computació quàntica. El que fa aquí és implementar la transformació:

$$|\psi\rangle \rightarrow |\psi\rangle|\psi\rangle$$

La qual es pot dur a terme afegint un registre de la mateixa mida que el que es vol copiar inicialitzat a $|0\rangle$ i aplicant portes CNOT que fan l'acció de copiar $|\psi\rangle$ al nou registre.

Una funcionalitat molt interessant d'aquest llenguatge són els operadors de control quàntics, ja que és l'únic llenguatge quàntic que ho suporta. Això vol dir que té dos tipus de sentències `if`, la clàssica, i una quàntica que realment no acabo d'entendre com funciona.

7.5.4.QPL i cQPL

Quantum Programming Language és un llenguatge dissenyat per Peter Selinger amb la peculiaritat que té una extensió per a implementar protocols de comunicació quàntica anomenat cQPL (*communication capable QPL*).

A continuació un exemple de codi QPL:

```
new qbit q := 0;
q *= H;
measure q then {
    print "Head!"
} else {
    print "Tail:(";
};
```

I un exemple de comunicació entre Alice i Bob de cQPL:

```
module Alice {
    new qbit a := 0;
    new qbit b := 0;
    send qbit a to Bob;
    send qbit b to Bob;
}

module Bob {
    receive q1:qbit from Alice;
    receive q2:qbit from Bob;
    dump q1;
    dump q2;
}
```

Un dels inconvenients més grans d'aquest llenguatge és la dificultat d'ús, ja que el compilador del llenguatge genera codi C++ que després s'ha de linkar amb una llibreria de simulació libqc i tornar a compilar per a obtenir un executable.

7.5.5.LanQ

Aquest llenguatge ha estat desenvolupat per Hynek Mlnarik (2006) amb una sintaxi semblant a C. Suporta les funcionalitats bàsiques d'un llenguatge de programació quàntica i a més, suporta la possibilitat de crear subprocessos que es comuniquin tant amb protocols clàssics com amb protocols quàntics.

Aquest programa d'exemple crea dos processos fills (angela i bert) que es comuniquen mitjançant un canal quàntic amb el protocol BBC+93:

```
void main() {
    qbit A, B;
    EPR aliasfor [ A, B];
    channel[int] c withends [c0, c1];
    EPR = createEPR();
    c = new channel[int]();
    fork bert(c0, B);
    angela(c1, A);
}

void angela(channelEnd[int] c1, qbit ats) {
    int r;
    qbit _;
    _ = doSomething();
    r = measure (BellBasis, _, ats);
    send (c1, r);
}
```

```
qbit bert(channelEnd[int] c0, qbit stto) {  
    int i;  
    i = recv (c0);  
    if (i == 0) {  
        opB0(stto);  
    } else if (i == 1) {  
        opB1(stto);  
    } else if (i == 2) {  
        opB2(stto);  
    } else {  
        opB3(stto);  
    }  
    doSomethingElse(stto);  
}
```

8. PyQu

8.1. Definició

PyQu és el software que he desenvolupat per completar aquest projecte. De fet, encara està en desenvolupament. En el moment de la finalització del pfc PyQu es troba en la versió 0.2 camí de la 0.3.

PyQu no és més que un mòdul per al llenguatge Python que proporciona estructures de dades i funcions que permeten implementar algorismes quàntics. Des d'un altre punt de vista es pot considerar PyQu com un llenguatge de programació quàntica pythonic (terme utilitzat per a definir el codi que segueix els principis de Python de llegibilitat i transparència), de fet la sintaxi s'assembla molt al llenguatge QCL. El mòdul està escrit en C i utilitza una llibreria de simulació de sistemes quàntics anomenada libquantum i desenvolupada per Björn Butscher i Hendrik Weimer.

L'objectiu principal del llenguatge desenvolupat és que sigui gairebé una descripció textual d'un circuit quàntic, és a dir, que quan algú vegi un tros de codi del llenguatge es pugui fer ràpidament una imatge mental del circuit que representa i viceversa. Això pot semblar una contradicció amb la meua intenció inicial, ja que volia fer un llenguatge més abstracte que el llenguatge de portes quàntiques, però hi ha dos fets que m'han fet canviar d'opinió. Primer, després d'estudiar els llenguatges de programació que s'han proposat en els últims anys he vist que la majoria segueixen aquesta idea, i dels que estan implementats, crec que tots la segueixen. Segon, m'he donat compte que el model de circuits està en un nivell d'abstracció més alt del que em pensava.

El disseny del llenguatge l'he fet basant-me en el llenguatge QCL, d'alguna manera el que he fet és adaptar un subconjunt de QCL a Python (tot i que tampoc és una adaptació rigorosa). També he agafat idees de l'article de S.Bettelli, T.Calarco i L.Serafini. En la secció especificació ho comentaré amb més detall. A més he seguit el seu desiderata:

Completeness. Amb el llenguatge dissenyat, és possible implementar qualsevol algorisme expressat en el model de circuits i qualsevol troç de codi es pot traduir a un circuit.

Classical extension. Això ja ve "de sèrie" en ser un extensió del llenguatge Python.

Separability. La part quàntica dels algorismes s'encapsula en l'objecte Qureg, els seus mètodes i les funcions que implementen els operadors unitaris i la mesura.

Expressivity. En seguir l'estil de Python, la programació quàntica amb PyQu és com la programació clàssica. A més, la traducció de les operacions d'alt nivell a "primitives" que son enviades al computador quàntic (en aquest cas a una llibreria que simula un computador quàntic), es fan en la capa que hi ha per sota de Python, justament la que jo he programat, i és aquí on és possible optimitzar les instruccions que ens arriben de la capa superior (tot i que en aquest projecte no s'ha tingut en compte).

Hardware independence. L'única cosa que he assumit de la possible arquitectura del computador quàntic és que segueix el model de QRAM. A part d'això el llenguatge és totalment independent del hardware.

8.2.Decisions sobre tecnologies usades

8.2.1.Python

En el camp de la programació quàntica hi ha unanimitat en que els llenguatges de programació quàntica han de ser extensions d'un llenguatge clàssic, per tant, per en

dissenyar i implementar el meu llenguatge calia escollir entre algun dels llenguatges disponibles actualment que fos extensible. Però la majoria dels programes més utilitzats actualment són extensibles.

Hi ha varies raons que m'han portat a triar Python per a realitzar el meu projecte. Primer de tot haig de dir que és un llenguatge que m'agrada molt ja que te una sintaxi molt amigable amb el programador, té una corba d'aprenentatge molt suau (és molt fàcil d'aprendre al principi, però no per això està limitat) i és molt fàcil d'utilitzar. Té una llibreria estàndard molt gran i una gran quantitat de mòduls que amplien el llenguatge gràcies a una enorme comunitat que ajuda a desenvolupar-lo.

Es tracta d'un llenguatge interpretat, la qual cosa comporta avantatges i inconvenients, tot i que per a mi pesen molt més els avantatges. En els pros del llenguatges interpretats hi ha el fet que són totalment independents de hardware i sistema operatiu, els programes funcionen igual en qualsevol plataforma.

Un altre dels avantatges més grans és la immediatesa; en ser un llenguatge interpretat no cal compilar, linkar, fer makefiles... el codi font s'executa directament, estalviant un temps gens despreciable al desenvolupador. Un altre avantatge és la possibilitat d'executar l'interpret de Python i escriure un programa "al vol" escrivint instruccions pas a pas i en funció dels resultats anteriors. D'aquesta manera molts cops es fa servir Python com una calculadora científica i m'atrau la idea de que PyQu pugui ser utilitzat com una mena de calculadora quàntica. Aquest sistema també es pot fer servir com una mena de "debugger quàntic" ja que permet executar un algorisme quàntic pas a pas i anar consultant l'estat del sistema en qualsevol moment.

Per contra, una crítica comú que tenen els llenguatges interpretats és que necessiten d'un intèrpret, o sigui, que l'usuari té la molèstia d'haver d'instal·lar-se un programa addicional per a poder fer servir el teu programa. Des del meu punt de vista és una

molèstia relativa ja que només s'ha de fer un cop, a part, Python permet construir (compilar) executables per a moltes plataformes per a que l'usuari no hagi d'instal·lar-se l'interpret de Python per executar el programa. Des del punt de vista del programador tampoc és una crítica vàlida, ja que els llenguatges no interpretats, si bé no necessiten un interpret, necessiten un compilador que generi el codi màquina a partir del codi font.

El que sí és cert és el consum de recursos addicional que comporta haver d'interpretar el codi en temps d'execució. Però aquí és on Python destaca sobre altres llenguatges interpretats, perquè Python és com una capa per sobre de C. Totes les funcions i objectes de la llibreria estàndard estan implementats en C, reduint al mínim els costos addicionals en interpretar el llenguatge.

A més Python permet estendre el llenguatge amb mòduls escrits en Python o amb mòduls escrits en C per a que siguin més eficients.

També cal dir que en començar el projecte acabava de sortir l'última versió de Python, la 3. Incomprendiblement, aquesta nova versió no és compatible amb versions anteriors, així que vaig haver de decidir si fer el mòdul per a les versions 2.x o 3.x. Crec que el més lògic és escollir sempre la última versió disponible, però la majoria de sistemes Unix tenen instal·lada alguna versió 2.x. De totes formes em vaig acabar decidint per la versió 3.x ja que el software desenvolupat no serà de gran consum, i una versió completa i estable del mateix encara tardarà bastant en sortir.

I, per acabar, una altra raó que em va portar a escollir Python és que no vaig trobar cap simulador de computació quàntica en aquest llenguatge, i em seduïa la idea de ser el primer en fer-ho.

8.2.2.Llibreries de simulació quàntica

Un cop decidit que ho faria en Python hi havia dues opcions.

Una, fer el mòdul purament en Python, però aquesta opció té dos inconvenients bastant importants: ho hauria d'implementar tot des de zero i probablement el rendiment seria bastant pobre. De fet, per a fer una prova de rendiment vaig implementar un petit simulador purament en Python on es pot crear un registre de qubits i aplicar-li la operació hadamard. Cal dir que és una implementació intencionadament ingènua; per a cada operació que s'aplica a un o varis qubits es fan tots els productes tensorials corresponents fins a crear una matriu $2^n \times 2^n$ que s'aplica a tot l'estat. Amb un registre de 12 qubits la matriu de l'operador té $2^{12} \times 2^{12} = 16777216$ elements (nombres complexos) i el programa tarda aproximadament 30 segons en executar-se, o sigui que queda bastant clar que aquesta opció és inviable.

La segona opció és la de fer el mòdul en C. Aquesta opció té l'inconvenient que és bastant més difícil d'implementar que un mòdul purament Python, però l'avantatge que no cal implementar-ho des de zero, ja que hi ha moltíssimes llibreries C/C++ per a la simulació de sistemes quàntics. A més, el problema del rendiment es redueix moltíssim (tot i que el temps de la simulació segueix augmentant exponencialment amb el nombre de qubits), d'una banda perquè està implementat en C, i de l'altra perquè les llibreries existents incorporen algunes millores de rendiment.

8.2.3.Q++

Q++ és una llibreria C++ desenvolupada per Chris Lomont i va ser la primera que vaig utilitzar per al meu mòdul. És una llibreria senzilla i que s'adaptava a les meves necessitats ja que està orientada a la simulació d'algorismes quàntics. Però, tot i que a la documentació deia que la llibreria estava acabada, faltaven certes parts i fins i tot hi

havia algun *bug*. En el moment en que em vaig trobar refent alguns trossos de codi interns de la llibreria vaig decidir que havia d'utilitzar una altra.

8.2.4.libquantum

Es tracta d'una llibreria C desenvolupada per Björn Butscher i Hendrik Weimer de la universitat de Munich. És una llibreria de simulació de sistemes quàntics en general, tot i que en un començament estava orientada a la simulació de computació quàntica. Això fa que s'adapti perfectament a les meves necessitats. L'únic però és que la documentació no és prou extensa, només està documentada l'API externa, però no el funcionament intern (hi ha un document en alemany que segurament explica alguna cosa del funcionament intern, però no he trobat cap traducció). Això ha provocat que moltes vegades he hagut d'esbrinar coses de la llibreria mentre programava el mòdul amb el mètode prova-error.

8.2.5.SWIG

Swig és una eina per a generar wrappers de llibreries C a multitud de llenguatges (entre ells Python) de forma automàtica. Al principi vaig creure que seria útil per al meu projecte, però ho vaig descartar de seguida per diferents motius. Un wrapper serveix per a poder cridar funcions implementades en un llenguatge des d'un altre llenguatge, però això només és una part del que jo volia fer. Així que vaig creure que ell millor seria fer el mòdul des de zero. Així també tenia l'oportunitat d'aprendre com funciona Python internament amb bastant profunditat.

8.3.Especificació

8.3.1.Creació i gestió de registres quàntics

L'objecte que aglutina totes les funcionalitats és el registre quàntic. El registre quàntic és una col·lecció ordenada d' n qubits i és possible accedir als qubits individuals com si fos una estructura d'array clàssic. En l'equivalent model de circuits, el registre quàntic són

els cables on s'apliquen portes lògiques i es fan mesures. És important dir que els qubits estan numerats de $n-1$ a 0 , o sigui, a la inversa de com s'etiqueten normalment en el model de circuits. En un circuit es representa el qubit de més amunt com el 1er qubit, és a dir, que si representem el circuit com un array el qubit de més amunt hauria de ser indexat pel 0 i el qubit de més avall hauria de ser indexat per $n-1$. Aquest fet provoca que el qubit de més pes sigui el $n-1$ i el de menys pes el 0 .

Això es degut a que la llibreria que utilitzo segueix aquest conveni però tinc intenció de canviar-ho en futures versions ja que pot provocar bastant confusió.

Per posar un exemple, considerem el següent circuit:

$$\begin{array}{l} |0\rangle \text{ —————} \\ |1\rangle \text{ —————} \end{array}$$

L'estat del sistema és $|0\rangle \otimes |1\rangle = |01\rangle$. En PyQu això és un registre de 2 qubits on el qubit 0 està a $|1\rangle$ i el qubit 1 està a $|0\rangle$. Una mesura del sistema ens retornaria un 1.

Una altra característica especial és la possibilitat de crear subregistres. Un subregistre és un subconjunt de qubits consecutius d'un registre i és tractat com un registre normal ja que té els mateixos mètodes i operacions, tot i que està limitat en alguns casos com veurem més endavant. Els subregistres ens proporcionen un grau més d'abstracció i més comoditat a l'hora de programar algorismes quàntics ja que moltes vegades ens interessa etiquetar un grup de qubits com l'*input* d'un algorisme i un altre grup com l'*output* i operar sobre ells com si fossin variables clàssiques. Els subregistres es poden solapar, és a dir, un qubit o conjunt de qubits pot estar referenciat per molts subregistres diferents, i operar sobre qualsevol d'aquests subregistres significa modificar-los tots.

8.3.1.1.Creació d'un registre quàntic:

Qureg([width,initVal])

Paràmetres:

- width (opcional) [nombre enter més gran que zero]: nombre de qubits del registre quàntic. Valor per defecte 1.
- initVal (opcional) [nombre enter més gran o igual que zero]: valor en que s'inicialitza l'estat del registre. Valor per defecte 0.

Retorna:

Un registre quàntic amb \$width\$ qubits en l'estat \$initVal\$.

Exemples:

```
>>> m = Qureg() #m és un qubit en l'estat |0>
>>> m = Qureg(2) #m és un registre de dos qubits en l'estat |00>
>>> m = Qureg(8, 255) #m és un registre de 8 qubits en l'estat |1111111>
>>> m = Qureg(8, 0b10101010) # m és un registre de 8 qubits en l'estat
|10101010> (el prefix 0b indica a Python que interpreti el nombre en base 2)
```

8.3.1.2.Creació de subregistres

La creació de subregistres es fa amb l'operador d'accés a arrays `q[]` (on `q` és un registre creat amb `Qureg()` o un subregistre). La sintaxi de dins els claudàtors és molt variada:

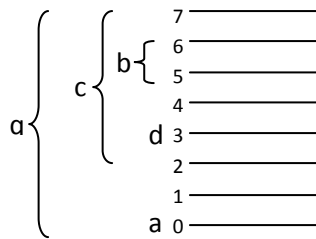
- `m[i]` retorna el qubit ièssim del registre `m`. Si `i` és negatiu retorna el qubit ièssim començant per la cua.
- `m[i:j]` retorna un subregistre format per tots els qubits entre `i` i `j-1` del registre `m`. Els nombres negatius tenen el mateix comportament que abans. Si `j-i >= 0` retorna `None` (tipus de dades especial de Python per representar `NULL`).
- `m[:i]` retorna un subregistre format per tots els qubits de `m` des de l'ièssim fins al final.

- $m[i]$ retorna un subregistre format per tots els qubits de m des del 0 fins a i-1.
- $m[:]$ retorna un subregistre amb tots els qubits de m

Exemples:

```
>>> q = Qureg(8)
>>> a = q[0]
>>> b = q[5:7]
>>> c = q[-6:]
>>> d = c[1]
```

Aquest codi és equivalent al següent circuit:



8.3.1.3.Composició de registres

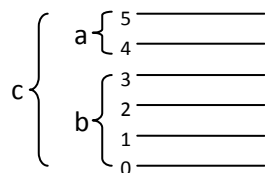
També podem compondre registres per formar "macroregistres" fent el producte tensorial de dos registres. No s'accepten subregistres per a realitzar aquesta operació:

- $a**b$ retorna un nou registre que és el producte tensorial dels registres a i b, els quals passen a ser subregistres del nou registre.

Exemples:

```
>>> a = Qureg(2)
>>> b = Qureg(4)
>>> c = a**b
```

Aquest codi és equivalent al següent circuit:



8.1.3.4.Mètodes dels registres (i subregistres)

width()

Paràmetre implícit

Retorna:

Nombre de qubits del registre

Exemples:

```
>>> Qureg(4).width() #retorna 4
>>> Qureg(4)[:2].width() #retorna 2
```

size()

Paràmetre implícit

Retorna:

Nombre d'elements de la base amb coeficient diferent de 0

Exemples:

```
>>> q = Qureg(8)

>>> q.size() #retorna 1 (l'únic element de la base amb coeficient diferent de 0 és
|00000000>)

>>> H(q) #apliquem una hadamard al registre sencer (més endavant veurem la
definicó)

>>> q.size() #retorna 28=256 (la porta hadamard ens ha creat una superposició
de tots els elements de la base)
```

coef()

Paràmetre implícit

Retorna:

Una llista ordenada de nombres complexos amb els coeficients dels elements de la base. La longitud de la llista és 2^n .

Observacions:

Si el paràmetre implícit és un subregistre es retornaran els coeficients de l'estat sencer (com hem vist en el capítol d'introducció a la computació quàntica, en

general un estat no es pot expressar com la composició d'estats més simples).

Exemples:

```
>>> Qureg(3).coef() #retorna una llista de 8 nombres complexos on tots son zero
excepte el primer que val 1

>>> Qureg(3)[0].coef() #retorna el mateix que abans (no té sentit parlar dels
coeficients d'un qubit en un sistema compost)
```

prob()

Paràmetre implícit

Retorna:

Una llista p de nombres reals on l'element i correspon a la probabilitat d'obtenir i en mesurar el registre.

Observacions:

En un registre sencer la llista retornada per prob() equival a elevar al quadrat cadascun dels elements de la llista retornada per coef(). Un dels principals avantatges d'utilitzar prob() és que sí que és possible calcular les probabilitats de mesura d'un subregistre.

Exemples:

```
>>> q = Qureg(2)

>>> H(q)

>>> q.prob() #retorna [0.25, 0.25, 0.25, 0.25]

>>> q[0].prob() #retorna [0.5, 0.5]
```

reverse()

Paràmetre implícit

Retorna:

None.

Aquesta funció inverteix el pes dels qubits. Això només afecta a l'hora de

realitzar la mesura, però no canvia l'etiquetatge intern dels qubits. És a dir, el qubit 0 segueix sent el mateix després d'aplicar reverse. Aquest mètode és útil en l'algorisme de Shor, ja que la porta QFT inverteix el pes dels qubits del registre sobre el que s'aplica.

Exemples:

```
>>> q = Qureg(8,1)
>>> measure(q) #retorna 1 (00000001)
>>> measure(q[0]) #retorna 1
>>> q.reverse()
>>> measure(q) #retorna 128 (10000000)
>>> measure(q[0]) #retorna 1 (igual que abans)
```

8.3.2. Operadors

H(reg)

Paràmetres:

reg [Qureg]: registre o subregistre sobre el que s'aplica la transformació.

Retorna:

None.

Aquesta funció aplica la transformació H a tots els qubits del registre \$reg\$

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

X(reg)

Paràmetres:

reg [Qureg]: registre o subregistre sobre el que s'aplica la transformació.

Retorna:

None.

Aquesta funció aplica la transformació X (NOT) a tots els qubits del registre

\$reg\$

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Y(reg)

Paràmetres:

reg [Qureg]: registre o subregistre sobre el que s'aplica la transformació.

Retorna:

None.

Aquesta funció aplica la transformació Y a tots els qubits del registre *\$reg\$*

$$Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$

Z(reg)

Paràmetres:

reg [Qureg]: registre o subregistre sobre el que s'aplica la transformació.

Retorna:

None.

Aquesta funció aplica la transformació Z a tots els qubits del registre *\$reg\$*

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

T(reg)

Paràmetres:

reg [Qureg]: registre o subregistre sobre el que s'aplica la transformació.

Retorna:

None.

Aquesta funció aplica la transformació T ($\frac{\pi}{8}$ gate) a tots els qubits del registre

$\$reg\$$

$$T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{pmatrix}$$

Rx(reg)

Paràmetres:

reg [Qureg]: registre o subregistre sobre el que s'aplica la transformació.

angle [nombre real]: Angle en radians

Retorna:

None.

Aquesta funció aplica una rotació de $\$angle\$$ respecte l'eix x de l'esfera de Bloch a tots els qubits del registre $\$reg\$$

$$R_x(\theta) = \begin{pmatrix} \cos \frac{\theta}{2} & -i \sin \frac{\theta}{2} \\ i \sin \frac{\theta}{2} & \cos \frac{\theta}{2} \end{pmatrix}$$

Ry(reg)

Paràmetres:

reg [Qureg]: registre o subregistre sobre el que s'aplica la transformació.

angle [nombre real]: Angle en radians

Retorna:

None.

Aquesta funció aplica una rotació de $\$angle\$$ respecte l'eix y de l'esfera de Bloch a tots els qubits del registre $\$reg\$$

$$R_y(\theta) = \begin{pmatrix} \cos \frac{\theta}{2} & -\sin \frac{\theta}{2} \\ \sin \frac{\theta}{2} & \cos \frac{\theta}{2} \end{pmatrix}$$

Rz(reg)

Paràmetres:

reg [Qureg]: registre o subregistre sobre el que s'aplica la transformació.

angle [nombre real]: Angle en radians

Retorna:

None.

Aquesta funció aplica una rotació de \angle respecte l'eix z de l'esfera de Bloch

a tots els qubits del registre reg

$$R_z(\theta) = \begin{pmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{pmatrix}$$

Cnot(controlReg, targetReg)

Paràmetres:

controlReg [Qureg]: subregistre de control

targetReg [Qureg]: subregistre de destí

Retorna:

None.

Aquesta funció aplica una transformació CNOT sobre tots el qubits del

$targetReg$ agafant tots els qubits de $controlReg$ com a qubits de control.

Observacions:

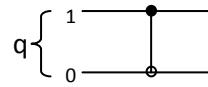
$controlReg$ i $targetReg$ han de ser subregistres del mateix registre i no es

poden solapar.

Exemple:

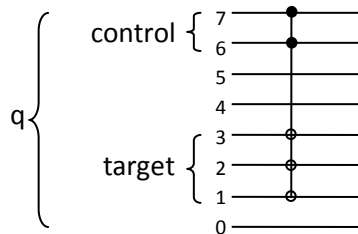
```
>>> q = Qureg(2)
>>> Cnot(q[1], q[0])
```

Aquest codi és equivalent al següent circuit:



```
>>> q = Qureg(8)
>>> control = q[6:]
>>> target = q[1:4]
>>> Cnot(control, target)
```

Aquest codi és equivalent al següent circuit:



QFT(reg)

Paràmetres:

reg [Qureg]: registre o subregistre sobre el que s'aplica la transformació.

Retorna:

None.

Aquesta funció aplica la Transformada de Fourier Quàntica a \$reg\$

$$QFT_n : |x\rangle \rightarrow \frac{1}{\sqrt{2^n}} \sum_{y=0}^{2^n-1} e^{2\pi i \frac{x}{m} y} |y\rangle$$

QFTinv(reg)

Paràmetres:

reg [Qureg]: registre o subregistre sobre el que s'aplica la transformació.

Retorna:

None.

Aquesta funció aplica la inversa de la Transformada de Fourier Quàntica a \$reg\$

$$QFT_n^{-1} : |x\rangle \rightarrow \frac{1}{\sqrt{2^n}} \sum_{y=0}^{2^n-1} e^{-2\pi i \frac{x}{m} y} |y\rangle$$

UexpModN(inputReg, outputReg, x, N)

Paràmetres:

inputReg [Qureg]: subregistre on hi ha la informació d'entrada de la funció.

outputReg [Qureg]: subregistre on es desarà la sortida de la funció. Hauria d'estar inicialitzat a 0.

x [nombre enter més gran que 1]: base de la funció.

N [Nombre enter més gran que 1]: mòdul de la funció.

Retorna:

None.

Aquesta funció aplica la transformació U_f al registre \$reg\$

$$U_f : |j\rangle|0\rangle \rightarrow |j\rangle|x^j \bmod N\rangle$$

Exemple:

```
>>> q = Qureg(10)
>>> input = q[5:]
>>> output = q[:5]
>>> H(input)
>>> UexpModN(input, output, 2, 21) #deixa a output una superposició de totes les
f(j) = 2^j mod 21 amb j=[0,...,2^5-1]
```

measure(reg)

Paràmetres:

reg [Qureg]: registre o subregistre a mesurar.

Retorna:

Un nombre enter i que representa l'element de la base computacional resultant de la mesura. Aquesta funció col·lapsa l'estat a $|i\rangle$

Exemple:

```
>>> q = Qureg(8)

>>> measure(q) #retorna 0 ja que l'estat era |0>

>>> H(q) #crea una superposició equiprobable de tots els nombres entre 0 i  $2^8-1$ 

>>> measure(q[0]) #retorna 0 o 1 amb un 50% de probabilitats. Si el resultat és 1,
l'estat q passa a ser una superposició de tots els nombres senars entre 0 i  $2^8-1$ . Si
el resultat és 0, l'estat q passa a ser una superposició de tots els nombres parells
entre 0 i  $2^8-1$ .
```

8.3.3. Definir subrutines i funcions

Com que el llenguatge està totalment integrat en Python és molt fàcil definir subrutines i funcions quàntiques.

Per exemple podem definir una subrutina que ens faci el *swap* de dos qubits a partir de tres portes Cnot:

```
>>> def Swap(reg, i, j):
>>>     Cnot(reg[i], reg[j])
>>>     Cnot(reg[j], reg[i])
>>>     Cnot(reg[i], reg[j])
>>> q = Qureg(2,1)
>>> measure(q) #retorna 1 ja que l'estat és |01>
>>> swap(q, 1, 0)
>>> measure(q) #retorna 2 ja que l'estat és |10>
```


O podem definir una funció que ens retorni un 1 o un 0 aleatòriament:

```
>>> def rand():
>>>     q = Qureg()
>>>     H(q)
>>>     ret = measure(q)
>>>     del q
>>>     return ret
>>> rand() #retorna 0 o 1 amb un 50% de probabilitats
```

8.4.Implementació

En aquest capítol em proposo comentar parts de la implementació de PyQu, decisions tècniques i peculiaritats de l'estructura interna de Python.

Primer cal aclarir que la implementació que aquí comento és la de la versió 0.2 de PyQu, que és la que hi ha en el CD adjunt a aquest document. La versió 0.2 no compleix exactament amb l'especificació del llenguatge d'aquesta memòria, comentaré les diferències concretes més endavant. La implementació d'aquesta versió i el codi aquí mostrat són susceptibles de modificacions en futures versions del mòdul. El gruix del codi es pot trobar en el fitxer pyqumodule.c. Les capçaleres de les funcions estan definides en el fitxer pyqumodule.h així com també els tipus de dades necessaris.

8.4.1.Entendre i estendre Python

Per a fer a fer un mòdul de Python amb C és necessari saber com funciona el llenguatge internament. Python proporciona una API C molt extensa que ens permetrà fer gairebé qualsevol cosa: crear nous tipus de dades, cridar funcions Python des de C, estendre classes de la llibreria estàndard...

Recordem que Python és una capa que està per sobre de C i que ens permet programar a un nivell d'abstracció més alt. En el fons l'únic que fa l'interpret és traduir "al vol" les

instruccions que nosaltres escrivim en Python i cridar la funció C corresponent. Per tant, quan es fa una extensió en C, bàsicament el que es fa és crear una correspondència entre instruccions d'alt nivell i funcions C.

Després només hem d'escriure un petit fitxer de configuració en Python que, en executar-lo, ens farà tota la feina fosca de compilar, linkar, empaquetar i finalment, instal·lar o generar el paquet de distribució per a diferents plataformes (veure el capítol Distribució del mòdul).

8.4.1.1. Tipus de dades en Python

8.4.1.1.1. Dynamic Typing

Una de les característiques de Python és que utilitza *dynamic typing*, és a dir, que les variables no es declaren com a un tipus concret (int, string...) sinó que simplement s'inicialitzen amb un valor qualsevol i el propi llenguatge li assigna el tipus corresponent en temps d'execució. Això ens permet, per exemple, canviar el tipus de la mateixa variable més endavant:

```
>>> a = 1.1  
>>> a = "Hello"
```

L'oposat a *dynamic typing* és *static typing*, és a dir que el tipus de les variables s'ha de declarar explícitament en el codi, com en C/C++, Java, etc. La majoria (per no dir tots) de llenguatges interpretats utilitzen *dynamic typing* ja que *l'static typing* serveix principalment per a facilitar la detecció d'errors en temps de compilació (per exemple quan intentem comparar un string amb un int). Però com que un llenguatge interpretat no es compila, aquests errors es detecten en temps d'execució, per tant, no es guanya res fent que sigui *static typing*, i sí molta flexibilitat si és *dynamic typing*.

No confondre la dicotomia *static/dynamic* amb *weak/strong typing*. *Strong typing* significa que les variables, a part d'emmagatzemar les dades que corresponen al seu valor actual, emmagatzema metadades que indiquen quin tipus de dades són els que hi ha en aquell espai de memòria. Python té *strong typing*. Per contra, C té *weak typing* ja que les variables només emmagatzemen dades. Per últim comentar que en Python totes les conversions (excepte de classe a subclasse) són explícites, de forma contrària a molts llenguatges interpretats.

8.4.1.1.2.L'objecte PyObject i altres estructures de dades

Python implementa el dynamic typing de la següent manera: cada tipus de dades en Python corresponen a una estructura de dades (struct) diferent en C. Aquesta estructura conté metadades del tipus de dades Python, com per exemple el nom, el tamany que ocupa en memòria, les funcions C que s'han de cridar quan es crea o es destrueix una instància d'aquest tipus, etc. Cada variable en Python es representa en C com una estructura amb una capçalera i un cos. El cos és on hi ha les dades de la variable i la capçalera és un objecte PyObject. PyObject és essencialment una estructura amb un contador de referències a la variable i un punter a l'estructura que guarda metadades. Tots els objectes en Python són extensions de PyObject.

Veiem un exemple d'això en el meu codi, aquesta és la definició del tipus Qureg:

```
typedef struct {
    PyObject_HEAD
    PyQuantumReg *pyqr;
    unsigned int first;
    unsigned int last;
    char reverse; //Boolean
} Qureg;
```

PyObject_HEAD és una macro que ens crea els camps de la capçalera que tot PyObject ha de tenir, la resta de camps són les dades pròpies de Qureg.

Per a definir les metadades necessitem crear un objecte del tipus PyTypeObject i omplir els camps corresponents segons les nostres necessitats, aquestes són les metadades per a l'objecte Qureg:

```
static PyTypeObject QuregType = {
    PyObject_HEAD_INIT(NULL)
    "pyqu.Qureg",          /* tp_name */
    sizeof(Qureg),         /* tp_basicsize */
    0,                     /* tp_itemsize */
    (destructor)Qureg_dealloc, /* tp_dealloc */
    0,                     /* tp_print */
    0,                     /* tp_getattr */
    0,                     /* tp_setattr */
    0,                     /* tp_reserved */
    0,                     /* tp_repr */
    &QuregNumber_methods,  /* tp_as_number */
    0,                     /* tp_as_sequence */
    &QuregMapping_methods, /* tp_as_mapping */
    0,                     /* tp_hash */
    0,                     /* tp_call */
    0,                     /* tp_str */
    0,                     /* tp_getattro */
    0,                     /* tp_setattro */
    0,                     /* tp_as_buffer */
    Py_TPFLAGS_DEFAULT |
    Py_TPFLAGS_BASETYPE,  /* tp_flags */
    "Quantum register",   /* tp_doc */
    0,                     /* tp_traverse */
}
```

```

0, /* tp_clear */
0, /* tp_richcompare */
0, /* tp_weaklistoffset */
*/
0, /* tp_iter */
0, /* tp_iternext */
Qureg_methods, /* tp_methods */
0, /* tp_members */
0, /* tp_getset */
0, /* tp_base */
0, /* tp_dict */
0, /* tp_descr_get */
0, /* tp_descr_set */
0, /* tp_dictoffset */
(initproc)Qureg_init, /* tp_init */
0, /* tp_alloc */
Qureg_new, /* tp_new */

};

```

La macro `PyObject_HEAD_INIT` ens defineix un punter a una funció que s'executa cada cop que es crea una instància de `Qureg` i inicialitza adequadament la capçalera `PyObject` de la instància. Podem veure que aquesta taula conté informació sobre el tipus de dades `Qureg` com el nom, el tamany en memòria i d'altres. Entre aquests altres hi ha alguns de destacables com els punters a les funcions creadora (`Qureg_dealloc`) i destructora (`Qureg_dealloc`) i un punter a una taula `Qureg_methods` que mostro a continuació:

```

static PyMethodDef Qureg_methods[] = {
    {"__getitem__", (PyCFunction)
    QuregMap_Subscript, METH_O|METH_COEXIST,
    "Returns a 'subregister' of the actual
    register"
    },

```

```

    {"size", (PyCFunction)Qureg_size,
    METH_NOARGS,

    "Returns the number of non-zero vectors"

    },

    {"width", (PyCFunction)Qureg_width,
    METH_NOARGS,

    "Returns the number qubits"

    },

    {"coef", (PyCFunction)Qureg_coef,
    METH_NOARGS,

    "Returns the coefficients of the vectors
    of the computational basis"

    },

    {"prob", (PyCFunction)Qureg_prob,
    METH_NOARGS,

    "Returns the probabilities of the
    vectors of the computational basis"

    },

    {"reverse", (PyCFunction)Qureg_reverse,
    METH_NOARGS,

    "Reverses the weight of the bits of the
    register"

    },

    {NULL} /* Sentinel */
};

```

Aquesta taula li diu a l'interpret quines funcions s'han d'executar quan des de Python s'accedeix als mètodes de Qureg.

Si seguim inspeccionant l'estructura PyTypeObject veurem que hi ha dos punters més a les taules QuregNumber_methods i QuregMapping_methods, que no copiaré aquí (veure codi per més detall). Aquestes taules defineixen les funcions que s'han de cridar

quan des de Python s'executen instruccions més complicades com quan s'utilitzen els operadors d'accés a array [] o operadors matemàtics com **.

8.4.1.2. Passar paràmetres de Python a C i viceversa

Com he dit abans, en Python tots els objectes són PyObject a nivell C, fins i tot tipus bàsics com enters, o booleans. Els arguments de les funcions ens arriben en forma de llista de punters a PyObject i s'han de convertir dinàmicament a tipus bàsics de C o subclasses de PyObject. De la mateixa manera els resultats de les funcions s'han de convertir a algun objecte PyObject. Per sort, la API de Python ens proporciona un conjunt de funcions per parsejar aquestes llistes d'arguments i per construir PyObjects a partir de tipus bàsics.

Per exemple la funció UexpModN espera com a arguments dos registres quàntics i dos enters, per parsejar els arguments ho fem amb la següent funció:

```
PyArg_ParseTuple(args, "O!O!ii", &QuregType,
&inputReg, &QuregType, &outputReg, &x, &N);
```

El paràmetre "O!O!ii" indica al parser que els valors esperats són dos objectes PyObject i dos enters. El signes d'exclamació indiquen al parser que li passem com a paràmetre un punter al tipus de dades que s'han de convertir. Si el parsing falla es genera una excepció i la funció retorna null.

Per construir un objecte Python a partir d'un tipus bàsic ho podem fer amb la funció:

```
PyObject *ret = Py_BuildValue("K", size);
```

En aquest cas estem construint un enter a partir d'un unsigned long long.

Les funcions en Python sempre retornen alguna cosa, inclús a alt nivell, si definim una funció sense especificar una instrucció de retorn, en realitat està retornant None, un tipus de dades especial que representa el valor nul.

Si volem implementar aquest comportament en C, la API de Python ens proporciona la macro `Py_RETURN_NONE` que retorna un `PyObject` del tipus `None`.

8.4.1.3. Gestió de la memòria

En Python totes les variables es guarden en memòria dinàmica i mai es passen per valor, sempre per referència. Per exemple:

```
>>> q = Qureg()
>>> m = q
```

La segona assignació no fa una còpia de `q`, sinó fa que `m` sigui una referència al mateix `PyObject`. Com hem vist anteriorment, `PyObject` manté un comptador de referències i cada cop que es crea una nova referència s'incrementa el comptador (i cada cop que s'elimina una referència es decrementa). Quan Python detecta que el comptador arriba a zero llavors allibera la memòria del seu *heap* particular. És molt important tenir això sempre en ment ja que quan creem o destruïm referències en C hem d'incrementar o decrementar el comptador manualment amb les macros `Py_INCREF` i `Py_DECREF`.

En la documentació de Python recomana que sempre que es necessiti memòria per allotjar-hi nous objectes es demani a la memòria dinàmica de Python i desaconsella utilitzar les funcions *malloc* i *free* pròpies de C perquè utilitzen espai fora de Python en demanar memòria directament al sistema operatiu.

Per a sol·licitar memòria dinàmica a Python hi ha moltes funcions, com per exemple `PyObject_New` (`PyObject_Del` per alliberar-la). Totes les variables desades en el *heap* propi han de ser subclasses de `PyObject`. Això em va suposar un problema ja que la llibreria *libquantum* fa un ús extensiu de les funcions *malloc* i *free* i no sabia fins a quin punt el fet de no seguir les recomanacions de la documentació de Python podia provocar errors. Finalment he deixat la llibreria tal qual i dins el codi del mòdul he evitat utilitzar *malloc* en benefici de `PyObject_New`.

8.4.2. Definir el mòdul

En Python tot són objectes, fins i tot les funcions i els mòduls com PyQu. Per tant, en la definició de PyQu hem de crear un objecte que sigui la implementació del mòdul com a subclasse de PyQu. Això ho fem creant una estructura del tipus PyModuleDef:

```
static struct PyModuleDef pyqumodule = {
    PyModuleDef_HEAD_INIT,
    "pyqu",
    NULL,
    -1,
    PyquMethods
};
```

En aquesta taula, entre altres coses, hi definim el nom del mòdul i un punter a una taula on s'hi defineixen les funcions estàtiques del mòdul (veure codi per més detall).

També hem de definir la funció que crea el mòdul:

```
PyMODINIT_FUNC PyInit_pyqu(void)
{
    PyObject* m;
    if (PyType_Ready(&QuregType) < 0)
        return NULL;
    m = PyModule_Create(&pyqumodule);
    if (m == NULL)
        return NULL;
    Py_INCREF(&QuregType);
    PyModule_AddObject(m, "Qureg",
        (PyObject *) &QuregType);
}
```

```

    srand(time(0));

    return m;

}

```

Aquesta funció bàsicament crea una instància de mòdul a partir de la taula que hem definit abans i li adhireix el tipus de dades Qureg. La instrucció `srand(time(0))` és perquè la llibreria `libquantum` utilitza les funcions de generació de nombres pseudo-aleatoris de C i si no la cridéssim es generarien els mateixos nombres aleatoris en cada execució del programa.

8.4.3.Simulació de la computació quàntica utilitzant libquantum

8.4.3.1.Implementació de subregistres

Recordem de l'especificació del llenguatge que a partir d'un registre quàntic podem crear subregistres amb l'operador d'accés a array `[]` i composar-ne amb l'operador `**` (tot i que aquest últim no està implementat en la versió 0.2 de PyQu).

Aquesta funcionalitat la he implementat a nivell intern creant un tipus de dades que extén `PyObject` i que l'usuari no pot crear directament. Aquest objecte és definit per aquest struct:

```

typedef struct {
    PyObject_HEAD
    quantum_reg qr;
    int refcount;
} PyQuantumReg;

```

Veiem que només té un camp de dades, `quantum_reg`, que és l'objecte que representa un registre quàntic a la llibreria `libquantum`. És sobre aquest objecte que s'efectuaran totes les operacions. També podem observar un camp `refcount`, però és una rèmor que ha quedat de la versió 0.1 i que desapareixerà en la versió 0.3. En resum podem dir que aquest objecte és un *wrapper* de `quantum_reg` a un objecte `PyObject`.

Després tenim la definició de Qureg:

```
typedef struct {  
    PyObject_HEAD  
    PyQuantumReg *pyqr;  
    unsigned int first;  
    unsigned int last;  
    char reverse; //Boolean  
} Qureg;
```

Aquest és l'objecte que es pot crear i manipular des de Python. Tant els registres com els subregistres són representats per aquest objecte. Podem veure que conté un punter a una estructura PyQuantumReg que és on s'emmagatzema la informació quàntica del registre. També té una parella d'enters, first i last, que indiquen quins són el primer i l'últim qubits del subregistre. Quan l'objecte representi un registre sencer first tindrà valor 0 i last valdrà el nombre de qubits del registre menys 1.

Per exemple, considerem el següent codi:

```
>>> q = Qureg(3)  
>>> m = q[:2]
```

La primera instrucció crea un objecte Qureg amb un punter a un objecte PyQuantumReg que també s'acaba de crear. First té valor 0 i last té valor 2. La segona instrucció crea un nou objecte Qureg amb un punter al mateix objecte PyQuantumReg abans creat i amb els valors first i last a 0 i 1 respectivament.

Quan des de Python es s'apliquen operacions sobre els objectes Qureg, aquests el que fan és aplicar aquestes operacions sobre el quantum_reg que hi ha en l'objecte PyQuantumReg del qual tenen la referència. D'aquesta manera els canvis en un subregistre també afecten als subregistres que referencien el mateix registre. A alt nivell, tot aquest embolic d'estructures de dades és transparent ja que tota l'acció s'encapsula en un únic objecte Qureg.

El camp `reverse` és un booleà que ens indica si el bit de més pes és el last (si `reverse` val 1) o el first (si `reverse` val 0), només es fa servir a l'hora de construir el nombre enter resultant d'una mesura.

8.4.3.2.Implementació d'operadors unitaris i mesures

Aquesta és una de les tasques més senzilles des del meu punt de vista ja que delego tota la feina a la llibreria `libquantum`, la qual ofereix moltes funcions que implementen les portes quàntiques més importants. Per exemple, la funció que implementa la porta hadamard queda així:

```
static PyObject *pyqu_H(PyObject *self, PyObject *args)
{
    int i;
    Qureg *reg;
    if(!PyArg_ParseTuple(args, "O!", &QuregType,
        &reg))
        return NULL;

    for(i = reg->first; i <= reg->last; i++)
        quantum_hadamard(i, &reg->pyqr->qr);

    Py_RETURN_NONE;
}
```

El que faig és parsejar els arguments de la funció (en aquest cas simplement un registre quàntic) i aplicar la funció `quantum_hadamard` per a tots els qubits del registre.

Un altre exemple més complicat és la implementació de la Transformada de Fourier Quàntica a partir de portes hadamard i conditional phase:

```

static PyObject *pyqu_QFT(PyObject *self, PyObject *args)
{
    int i,j;
    Qureg *reg;
    if(!PyArg_ParseTuple(args, "O!", &QuregType, &reg))
        return NULL;

    int first = reg->first;
    int last = reg->last;
    for(i=first; i <= last; i++)
    {
        quantum_hadamard(i, &reg->pyqr->q);
        for(j=i+1; j<=last; j++)
            quantum_cond_phase(j, i, &reg->pyqr->q);
    }
    Py_RETURN_NONE;
}

```

Un altre exemple interessant és com s'aplica la porta UexpModN:

```

static PyObject *pyqu_QFT(PyObject *self, PyObject *args)
{
    int i,j;
    Qureg *reg;
    if(!PyArg_ParseTuple(args, "O!", &QuregType, &reg))
        return NULL;

    int first = reg->first;
    int last = reg->last;
    for(i=first; i <= last; i++)
    {
        quantum_hadamard(i, &reg->pyqr->q);
        for(j=i+1; j<=last; j++)
            quantum_cond_phase(j, i, &reg->pyqr->q);
    }
}

```

```

    }

    Py_RETURN_NONE;
}

```

La llibreria libquantum necessita qubits addicionals per a poder calcular la transformació a causa de la seva implementació interna, concretament necessita multiplicar per tres la mida del registre de sortida i afegir dos qubits més. Per afegir els qubits utilitzo la funció `quantum_addscratch` i els esborro amb `quantum_bmeasure` un cop s'ha realitzat la transformació.

En aquesta versió falta comprovar que els registres d'entrada i de sortida no se solapin.

La funció que implementa la mesura és la següent:

```

static PyObject *pyqu_measure(PyObject *self, PyObject *args)
{
    int i,r;
    unsigned int m = 0;
    Qureg *reg;
    if(!PyArg_ParseTuple(args, "O!", &QuregType, &reg))
        return NULL;
    if(!reg->reverse)
    {
        for(i = reg->first; i <= reg->last; i++)
        {
            r = quantum_bmeasure_bitpreserve(i, &reg->pyqr->qr);
            m = m | (r << (i - reg->first));
        }
    } else {

```

```

        for(i = reg->first; i <= reg->last; i++)
        {
            r = quantum_bmeasure_bitpreserve(i, &reg->pyqr->q);
            m = m | (r << (reg->last - i));
        }
    }
    return Py_BuildValue("l", m);
}

```

El que fa és mesurar tots els qubits del registre un a un i construir el valor resultant de la mesura tenint en compte el pes de cada bit.

8.4.4 Diferències entre la implemetació i l'especificació

Com hem anat dient, la versió actual de PyQu encara no compleix totes les especificacions, llisto aquí les diferències entre la versió 0.2 i l'especificació:

- El mètode size de Qureg sempre retorna 2^n on n és el nombre de qubits del registre. En canvi a l'especificació diu que retorna el nombre d'elements de la base amb coeficient diferent de zero. Això té sentit si parlem de registres sencers, però no de subregistres. Pendent de solucionar.
- No està implementada l'operació de composició ******. Encara no he decidit que passa amb els subregistres quan es combinen dos registres.
- La transformació Cnot no està implementada per a registres de més d'1 qubit. En cas de passar registres de més d'1 qubit s'agafarà el qubit 0 del registre.
- Falten implementar les transformacions de rotació així com la QFT inversa.

8.4.5.Distribució del mòdul

Un cop hem programat el mòdul l'hem de compilar per poder-lo provar o distribuir per a que altres usuaris el puguin fer servir. Python ve dotat d'un mòdul per a distribuir mòduls molt complet: distutils. Aquest paquet ens permet generar "builds" del nostre mòdul per a testejar, generar releases o instal·lar el mòdul entre altres coses. Només hem d'escriure un petit fitxer setup.py que en el cas de PyQu és el següent:

```
from distutils.core import setup, Extension

pyqu_module = Extension('pyqu',
                        libraries = ['quantum'],
                        library_dirs = ['lib'],
                        include_dirs = ['inc'],
                        sources = ['src/pyqumodule.c'])

setup( name = 'Pyqu',
      version = '0.2',
      description = 'Module for quantum computation simulation',
      author = 'Eric Marcos Pitarch',
      author_email = 'ericmarcos.p@gmail.com',
      url = 'http://code.google.com/p/pyqu/',
      long_description = "",
      ext_modules = [pyqu_module])
```

A part d'algunes metadades com el nom, la versió, l'autor, etc, hem d'indicar rutes a llibreries, includes i codi font per a que pugui compilar el mòdul amb èxit.

Un cop creat aquest fitxer el podem executar passant-li diferents paràmetres segons que és el que volem fer:

```
$>python setup.py build
```


Aquesta comanda simplement compila el mòdul en un fitxer pyqu.so. Si executem Python des del directori on hi ha aquest fitxer ja podem importar el mòdul:

```
>>> from pyqu import *
```

També podem crear distribucions del mòdul, és a dir, paquets pensats per a que se'ls baixi la gent per instal·lar el mòdul. Es poden crear dos tipus de distribucions: *source distributions*, que ve a ser un fitxer comprimit amb el codi font i llibreries necessaris per compilar el mòdul, i *build distributions*, que són executables específics per a cada plataforma i són la manera més fàcil per a l'usuari d'instal·lar-se la extensió.

```
$>python setup.py sdist
```

Genera un *source distribution*.

```
$>python setup.py bdist
```

Genera un *dumb build distribution* (el fitxer pyqu.so dins un .tar.gz).

```
$>python setup.py bdist_rpm
```

Genera un *build distribution* amb un instal·lador per Linux.

```
$>python setup.py bdist_wininst
```

Genera un *build distribution* amb un instal·lador per sistemes Windows.

Per instal·lar el mòdul a partir d'un *source distribution* només cal executar la següent comanda:

```
$>python setup.py install
```

9. Planificació del projecte

9.1. Planificació inicial

Durada total del projecte: 24 setmanes, del 12 de gener al 26 de juny

Càrrega de treball: 25 hores setmanals

Càrrega total: 600 hores

El model de desenvolupament que em proposo seguir és anar completant petites iteracions del cicle

Recerca -> Disseny -> Implementació -> Test

La primera iteració és la que requereix una etapa de recerca més llarga, però la resta d'etapes són molt curtes. Posteriors iteracions requeriran menys temps a recerca (tot i que segueix sent fonamental) i més de la resta. A continuació llisto els objectius a assolir en cada iteració i les dates de finalització de cada una:

· Iteració 1 (15/4/09): 1a versió rudimentària de PyQu. Bàsicament, s'ha de poder cridar la majoria de les funcions de Q++ des de Python

· Iteració 2 (20/5/09): Disseny del llenguatge propi i implementació. Definició de nous tipus de dades i operadors. En aquest punt hauria de ser possible executar Shor.

· Iteració 3 (10/6/09): En funció de la iteració anterior fer canvis e el disseny i ampliar funcionalitats per a poder executar altres algorismes (tot i que de moment no tinc previst implementar correcció d'errors quàntics).

- Iteració 4 (25/6/09): Aplicar millores d'eficiència en la llibreria Q++.

També considero la possibilitat d'introduir iteracions intermèdies en funció dels objectius parcials assolits en cada moment.

9.2.Desviació de la planificació inicial

Han existit diverses causes que han provocat que no hagi pogut assolir els *timings* plantejats inicialment.

La causa principal és que no he pogut seguir la càrrega de treball proposada. Aquí ha pesat molt el fet de que treballa unes 30 hores setmanals i, com que el projecte no és modalitat empresa, les 25 hores setmanals assignades a la realització del PFC són addicionals. Senzillament, no he pogut portar aquest ritme.

Un altre problema és que, conscient de la situació comentada abans, no només no ho he arreglat intentant distribuir millor el temps sinó que he acumulat la major part de la feina al final, havent de dedicar una mitjana d'unues 35 hores setmanals durant les últimes 6 setmanes.

Finalment les hores totals dedicades al PFC han sigut 420 (repartides en un 66% en tasques d'investigació/documentació i 33% en tasques de disseny/programació), el que correspon a una càrrega mitjana de 17 hores a la setmana.

Una altra causa del retard és el fet d'haver de canviar de llibreria (passar de Q++ a libquantum) a mig desenvolupament, concretament a principis de maig.

Tot seguit llisto les iteracions realitzades (i dates de finalització):

- Iteració 1 (25/4/09): 1a versió rudimentària de PyQu amb la llibreria Q++.

- Iteració 2 (1/6/09): Disseny del llenguatge propi i implementació de certes funcionalitats bàsiques utilitzant la llibreria libquantum.
- Iteració 3 (18/6/09): Correcció d'errors de la iteració anterior. Incorporació de més funcionalitats. Implementació de l'algorisme de Shor.

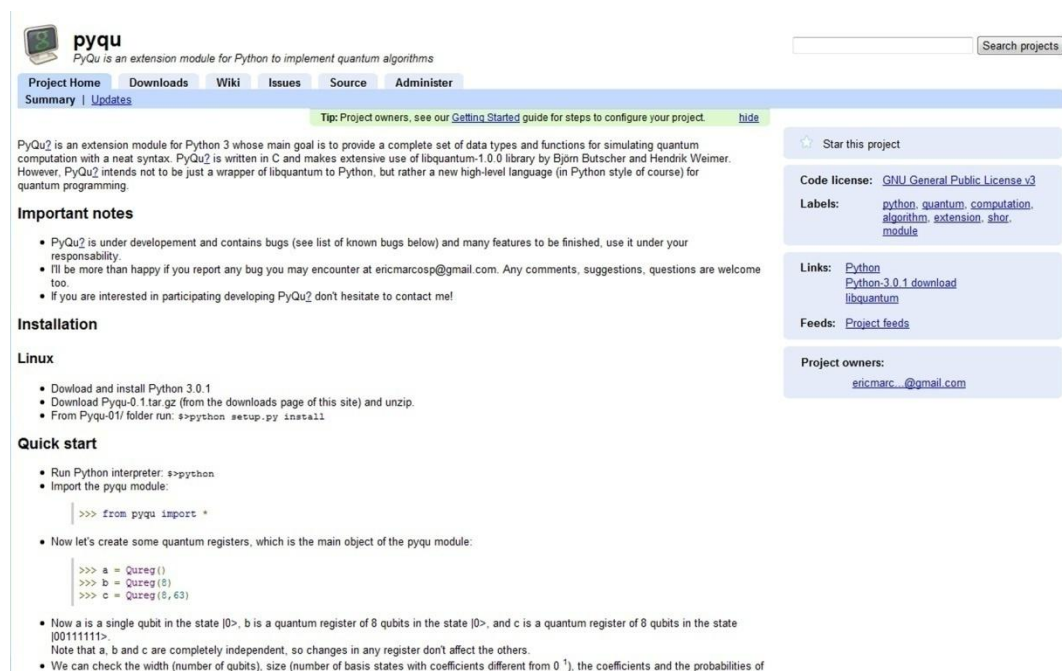
10. Gestió del projecte

La gestió del projecte s'ha fet des de Google code, un servei gratuït de Google per allotjar projectes open source. Proporciona un repositori subversion, una eina molt útil per al desenvolupament de projectes que permet tenir el codi en un servidor central i que guarda totes les modificacions dels fitxers amb un sistema de versions. És ideal per a projectes desenvolupats per molts programadors alhora. Cal destacar que hi ha tota una interfície web per navegar pel repositori, consultar versions anteriors, comparar versions, etc.

També proporciona una secció de descàrregues on hi ha els releases oficials i una wiki on documentar el projecte.

La web del projecte és code.google.com/p/pyqu

La llicència que he escollit per a PyQu és la GNU-v3.



The screenshot shows the Google Code project page for PyQu. At the top, there's a header with the project name 'pyqu' and a description: 'PyQu is an extension module for Python to implement quantum algorithms'. Below this is a navigation bar with links: Project Home, Downloads, Wiki, Issues, Source, and Administer. A search bar is on the right. The main content area is divided into sections: Summary, Important notes, Installation, and Quick start. The 'Important notes' section lists several points about the project's development status and how to report bugs. The 'Installation' section provides instructions for Linux users, including downloading Python 3.0.1 and the PyQu tar.gz file, and running a setup script. The 'Quick start' section shows how to run the Python interpreter, import the PyQu module, and create quantum registers. On the right side, there's a sidebar with a 'Star this project' button, a 'Code license' section (GNU General Public License v3), a 'Labels' section with links to the project's labels, a 'Links' section with links to the Python website, the PyQu download page, and the libquantum library, and a 'Feeds' section with a link to the project's feeds. At the bottom of the sidebar, there's a 'Project owners' section with the email address 'ericmarc...@gmail.com'.

pyqu
PyQu is an extension module for Python to implement quantum algorithms

Project Home Downloads Wiki Issues Source Administer

Summary | Updates

Tip: Project owners, see our [Getting Started](#) guide for steps to configure your project. [hide](#)

PyQu2 is an extension module for Python 3 whose main goal is to provide a complete set of data types and functions for simulating quantum computation with a neat syntax. PyQu2 is written in C and makes extensive use of libquantum-1.0.0 library by Björn Butscher and Hendrik Weimer. However, PyQu2 intends not to be just a wrapper of libquantum to Python, but rather a new high-level language (in Python style of course) for quantum programming.

Important notes

- PyQu2 is under development and contains bugs (see list of known bugs below) and many features to be finished, use it under your responsibility.
- I'll be more than happy if you report any bug you may encounter at ericmarcosp@gmail.com. Any comments, suggestions, questions are welcome too.
- If you are interested in participating developing PyQu2 don't hesitate to contact me!

Installation

Linux

- Download and install Python 3.0.1
- Download Pyqu-0.1.tar.gz (from the downloads page of this site) and unzip.
- From Pyqu-01/ folder run: `$python setup.py install`

Quick start

- Run Python interpreter: `$python`
- Import the pyqu module:

```
>>> from pyqu import *
```
- Now let's create some quantum registers, which is the main object of the pyqu module:

```
>>> a = Qureg()
>>> b = Qureg(8)
>>> c = Qureg(8,63)
```
- Now a is a single qubit in the state $|0\rangle$, b is a quantum register of 8 qubits in the state $|0\rangle$, and c is a quantum register of 8 qubits in the state $|00111111\rangle$.
Note that a, b and c are completely independent, so changes in any register don't affect the others.
- We can check the width (number of qubits), size (number of basis states with coefficients different from 0), the coefficients and the probabilities of

Star this project

Code license: [GNU General Public License v3](#)

Labels: [python](#), [quantum](#), [computation](#), [algorithm](#), [extension](#), [alter](#), [module](#)

Links: [Python](#), [Python 3.0.1 download](#), [libquantum](#)

Feeds: [Project feeds](#)

Project owners: ericmarc...@gmail.com

11. Futur del projecte

La versió actual de PyQu és la 0.2, però la versió 0.3 està prevista per mitjans de juliol. Així doncs, la feina més immediata és acabar la versió 0.3 de PyQu, que com a principal prioritat té acabar d'implementar totes les especificacions d'aquest document.

Una de les noves funcionalitats que tindrà serà la possibilitat de simular els efectes de la decoherència i aplicar correcció d'errors quàntics. Una altra funcionalitat nova serà la possibilitat d'aplicar operadors definits per l'usuari.

En futures versions seria convenient poder passar al computador quàntic una funció booleana com a oracle per a poder simular l'algorisme de Grover.

Una altra millora interessant seria proporcionar una api C per a poder utilitzar el mòdul des de altres mòduls Python escrits en C.

Per falta de temps m'he quedat sense fer un estudi del rendiment de PyQu en comparació altres llenguatges, així com investigar possibles millores de rendiment en el mòdul C.

12. Conclusions

Una de les meves hipòtesis abans començar el projecte era que la computació quàntica s'havia quedat estancada i una possible causa era la dificultat i diversitat de notacions que s'utilitzen. Creia que un llenguatge d'alt nivell, més compacte i abstracte, ajudaria a comprendre millor els algorismes actuals i a avançar més ràpidament en el disseny de nous algorismes. Però només començar el projecte ja vaig veure que no era un problema de llenguatge. La computació quàntica té molts aspectes intrínsecs que la fan difícil d'entendre i que no depenen de com s'expressen, com per exemple la seva naturalesa complexa (em refereixo a nombres complexos), la condició de reversabilitat i sobretot el fet que la mesura trenca la superposició quàntica i colapsa l'estat. També haig de dir que era una hipòtesi agosarada i feta gairebé des de la ignorància, per tant, era molt probable que fos falsa.

Una altra hipòtesi era que, almenys per a un informàtic, el fet de poder programar una màquina quàntica a nivell pràctic, ajudaria a entendre més ràpidament el funcionament dels algorismes quàntics. En aquest sentit crec que el software que he desenvolupat té una gran utilitat pedagògica, sobretot per a alumnes provinents de la banda d'informàtica que moltes vegades necessitem programar i obtenir resultats per veure el funcionament dels algorismes. Com ja indico en la introducció del treball, jo mateix he pogut comprovar que això és cert. Implementar l'algorisme de Shor en el meu llenguatge m'ha servit per entendre els detalls que a nivell teòric se m'escapaven.

A nivell personal puc dir que he assolit els objectius que m'havia proposat al principi del projecte, que es poden resumir en aprofundir els meus coneixements en dos vessants molt diferents. D'una banda el teòric, el de la computació quàntica i els possibles llenguatges de programació, i de l'altre el tècnic, com estendre Python amb mòduls C.

També vull comentar aquí breument els aspectes negatius del meu PFC. Primer, el fet que la base del projecte fos un conjunt de temes i tècniques dels quals només en tenia una vaga idea va fer que la planificació fos molt difícil de fer i de seguir. I segon, les condicions en les que he realitzat el projecte no eren les més idònies, ja que treballo 30 hores setmanals. Això ha suposat que el temps que he dedicat al projecte hagi sigut d'un 70% del temps que tenia planificat i, a més, un 50% del volum de treball acumulat en les últimes 6 setmanes. Amb això no vull posar cap excusa, sinó més aviat constatar que m'he planificat bastant malament.

Tot i això he complert l'objectiu bàsic de dissenyar i implementar un nou llenguatge de programació quàntica i simular l'algorisme de Shor amb el llenguatge propi.

Bibliografia i referències

"Structured Quantum Programming". Bernhard Ömer (2009)

"Quantum Programming Languages. Survey and Bibliography". Simon J Gay (2005)

"Toward an architecture for quantum programming".

S.Bettelli, T.Calarco i L.Serafini (2003).

"Shor's Algorithm for Factoring Large Integers".

C. Lavor, L.R.U. Manssur, and R. Portugal (2003)

"Quantum programming with mixed states". Paolo Zuliani (2005)

"A functional quantum programming language". Jonathan James Grattage (2006)

"Introduction to LanQ – an Imperative Quantum Programming Language".

Hynek Mlnarík (2006)

"One Complexity Theorist's View of Quantum Computing". Lance Fortnow (2008)

"The Hidden Subgroup Problem – Review and Open Problems". Chris Lomont (2004)

"Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a

Quantum Computer". Peter W Shor (1994)

"An Introduction To Quantum Computing". Phillip Kaye, Raymond Laflamme, Michelle

Mosca (2007)

"Parallel implementation of a quantum computing simulator". Marek Sawerwain, Jakub

Pilecki (2005)

Agraïments

Al Lluís Ametller per la seva paciència

A l'Adrià Gascón i al Guillem Godoy pel seu interès i ajuda

Al meus amics per estar al meu costat

Als meus pares i familiars per la seva insistència, per la seva comprensió

