Simulateur d'ordinateur quantique

Bruno BORRI Thomas CLAVEIROLE Valentin DAVID
Loïc FOSSE Vincent GOUZON David MANCEL
Giovani PALMA Jérôme POUILLER Marco TESSARI
Niels VAN VLIET Clément VASSEUR

EPITA - Juin 2004

Table des matières

1 Introduction **2**
1.1 Introduction . . . 2
1.2 Historique . . . 2
1.3 Solutions existantes . . . 3
1.4 Objectifs . . . 3
1.5 Découpage du projet . . . 3

2 Présentation d'un ordinateur quantique **6**
2.1 Introduction . . . 6
2.1.1 La base de la théorie . . . 6
2.1.2 Traitement sur les qbits et portes . . . 7
2.2 Problème de la création d'un ordinateur quantique . . . 8

3 Technologies utilisées **9**
3.1 Le C++ . . . 9
3.2 Le SWIG et le Python . . . 10
3.3 Les Autotools . . . 10
3.4 Le LaTeX . . . 11
3.5 Système de version . . . 11
3.6 La mailing list . . . 12
3.6.1 Organisation des mises à jour . . . 12
3.7 Coding style . . . 15

4 Implémentation **17**
4.1 Les outils . . . 17
4.1.1 Les complexes . . . 17
4.1.2 Les matrices . . . 18
4.2 Les registres quantiques . . . 19
4.2.1 Structure interne . . . 19
4.2.2 Les opérations de bases . . . 21
4.2.3 Lecture d'une composante . . . 25

4.3 Algorithmes de test 25
4.3.1 Shor 25
4.3.2 Grover 27

5 Conclusion **28**

Chapitre 1

Introduction

1.1 Introduction

Un ordinateur quantique opère ses calculs grâce à la superposition quantique d'états quantiques. De petits ordinateurs quantiques ont déjà été récemment construits et des progrès sont en cours. Beaucoup de gouvernements et d'organisations militaires telles que l'OTAN sponsorisent des universités et des centres de recherches pour développer un tel ordinateur à des fins cryptographiques et de surveillance, pouvant servir par exemple au renseignement militaire.

1.2 Historique

Dans les années 70 et 80, les premiers ordinateurs quantiques naissent dans l'esprit des physiciens tels que Richard Feynman, Paul Benioff, David Deutsch ou Charles Bennett. Pendant longtemps les physiciens ont douté que l'ordinateur quantique puisse exister. Mais en 1994, Peter Shor, un scientifique de AT&T montre qu'il est possible de factoriser des grands nombres dans un temps raisonnable à l'aide d'un ordinateur quantique. En 1996, Lov Grover, invente un algorithme basé sur les ordinateurs quantiques permettant de trouver une entrée dans une base de données non triée en $O(N1/2)$. Puis, en 1998, IBM est le premier à présenter un ordinateur quantique de 2 qbits. En 1999, l'équipe d'IBM utilise l'algorithme de Grover sur un ordinateur de 3 qbits et battent leur record l'année suivante avec un ordinateur de 5 qbits. Le 19 Décembre 2001, IBM crée un ordinateur quantique de 7 qbits et factorise le nombre 15 grâce à l'algorithme de Shor.

1.3 Solutions existantes

IBM possède à titre très expérimental quelques registres quantiques, mais tout le monde ne peut pas se permettre de faire tester des algorithmes dessus. (De plus, ils ne sont pas suffisamment puissant pour être réellement utilisés). La communauté scientifique a donc besoin de tester les algorithmes quantiques qu'elle produit sur des machines habituelles. Nous avons donc besoin de simuler le fonctionnement d'un ordinateur quantique. Il existe à l'heure actuelle quelques bibliothèques permettant de simuler des portes quantiques. Citons la `libquantum` qui est sûrement la plus complète parmi les bibliothèques libres.

1.4 Objectifs

Notre objectif est de réaliser une bibliothèque de simulation d'un ordinateur quantique qui soit facilement accessible à tout le monde et facilement utilisable. L'objectif principal est qu'elle soit rapide et simple d'utilisation.

1.5 Découpage du projet

Les différentes tâches à effectuer pour le projet sont celles ci :

- rechercher de la documentation,
- comprendre le fonctionnement d'un ordinateur quantique,
- mettre en place la logistique (mailing liste, subversion, etc.),
- rédiger le rapport de communication,
- rédiger le rapport du projet (partie théorique et pratique),
- modéliser et définir l'interface du projet,
- implémenter une bibliothèque de matrices,
- implémenter la représentation d'un registre quantique,
- implémenter les opérations de base sur les registres,
- écrire une batterie de tests,
- implémenter des algorithmes utilisant la bibliothèque,
- préparer la soutenance de communication,
- préparer la soutenance de calculabilité.

Le projet étant à réaliser dans un groupe composé de nombreuses personnes, il est important de savoir les tâches qui sont parallélisables et au contraire les tâches qui doivent être exécutées consécutivement. Le calendrier utilisé est présenté en figure 1.1 et 1.2.

Dans un premier temps nous avons tous fait des recherches afin de récupérer des informations sur les ordinateurs quantiques, pour que tout le monde sache de

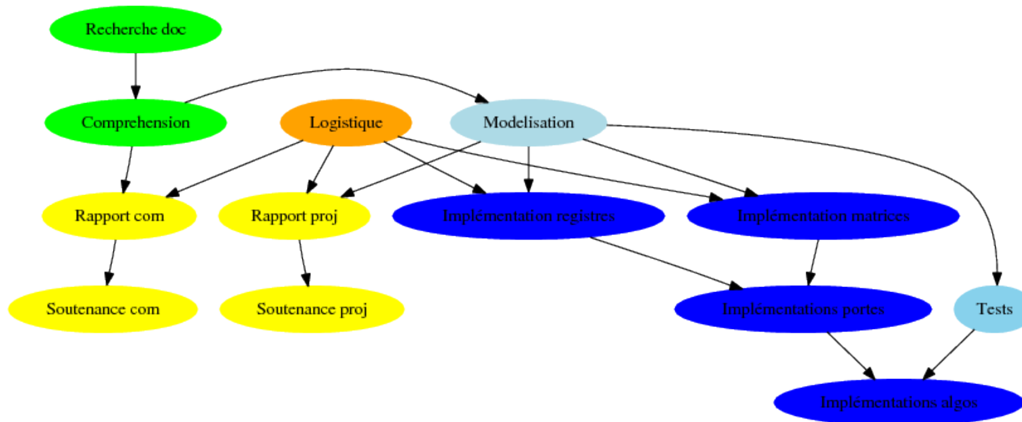

FIG. 1.1 – Dépendances des tâches.

quoi on parle et ce qu'il faut faire. Une première réunion a eu lieu pour mettre au clair nos connaissances.

Ensuite nous avons découpé le groupe en plusieurs équipes selon les affinités de chacun. Selon les disponibilités nous avons établi le planning de façon à créer un flux tendu pour qu'il y ait toujours quelqu'un qui travaille sur le projet afin qu'il avance et que les tâches successives soient faites dans le bon ordre.

- L'équipe travaillant sur le rapport est composée de quatre personnes : Niels, Jérôme, Marco et Bruno.
- L'équipe travaillant sur le code est composée de quatre personnes : Thomas, Loïc, Giovanni et Clément. Le responsable de la qualité (relecture du code, tests) est Giovanni.
- Les responsables de la préparation de la soutenance sont Valentin et David.
- Le responsable du rapport de communication est Vincent.
- Le responsable administratif et communication du groupe est Loïc.

Chacun s'est occupé principalement de ce dont il était responsable mais toute la partie réflexion et conception s'est faite en commun. De plus les personnes étant peu chargées par leurs tâches se sont rendues disponibles auprès des autres et ont aidé quand ils avaient besoin d'aide, comme David et Vincent qui ont participé à la rédaction du rapport.

Loïc, le responsable communication du groupe ayant eu des problèmes de santé, il y eu un passage à vide au sein du groupe qui aurait pu compromettre le projet. La communication fut presque nulle pendant une certaine période et l'avancement du projet a donc été beaucoup ralentie. Mais grâce au système de flux tendu, les choses sont très vite reparties après un ressaisissement du groupe.

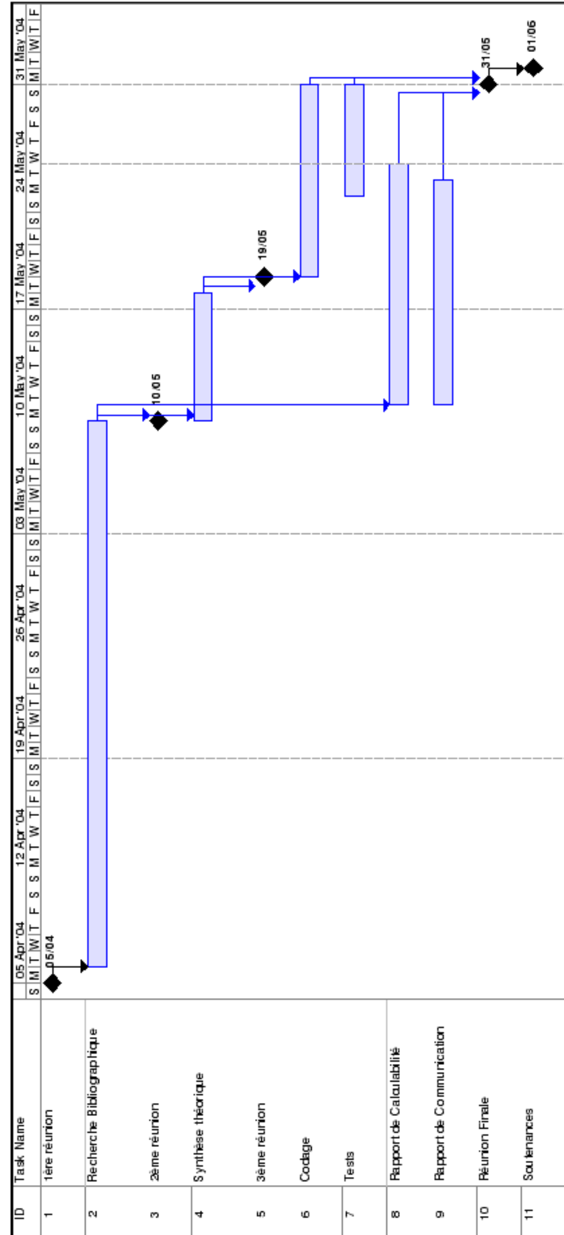

FIG. 1.2 – Diagramme de Gant du projet.

Chapitre 2

Présentation d'un ordinateur quantique

2.1 Introduction

L'informatique quantique est un champ de recherche s'intéressant à l'exploitation des phénomènes quantiques. Pendant la seconde guerre mondiale, le problème du décryptage des messages en grand nombre cryptés par des méthodes de clés reposant sur la difficulté algorithmique de la décomposition en nombres premiers s'est posé. Par nature même, les ordinateurs quantiques seraient capables de factoriser des grands nombres en un temps exponentiellement court, ce qui fait d'eux une arme terrible puisqu'ils pourraient décoder tous les cryptages actuels très rapidement. Ils permettent de nouveaux types d'algorithmes capables de résoudre des problèmes très difficiles pour l'informatique actuelle. Pour ces raisons, beaucoup de domaines s'intéressent aujourd'hui au quantique.

2.1.1 La base de la théorie

La théorie quantique repose sur le double état potentiel des bits quantiques ou *qbit* (pour *quantum bit*). En effet, les bits logiques sont booléens et ne peuvent donc prendre que les deux valeurs 0 ou 1, représentant respectivement *vrai* et *faux*. Au contraire, un qbit peut être dans une superposition cohérente de ses deux états. Néanmoins, la lecture du qbit positionne sa valeur sur un état selon une loi probabilité.

Par extension, si un registre est composé d'un certain nombre de qbits, il peut lui-même être dans une superposition cohérente de différents états. Dans un ordinateur quantique, les registres sont composés de qbits, et peuvent donc prendre plusieurs valeurs à la fois, ce qui permet l'exploration simultanée de situations

correspondantes aux différentes valeurs du registre. La seule contrainte se situe au niveau de la lecture. La difficulté consiste à trouver une manière intelligente de traiter les qbits pour que la valeur lue ai une bonne probabilité d'être la bonne réponse. Le parallélisme des ordinateurs quantiques permet donc l'exploration d'un espace bien plus grand pour un nombre d'opérations donné.

2.1.2 Traitement sur les qbits et portes

Pour comprendre le traitement des qbits, nous allons procéder par l'exemple. Supposons que l'on travaille sur trois qbits. L'ensemble des valeurs possibles est le suivant :

$$\begin{array}{c} 000 \\ 001 \\ 010 \\ \dots \\ 111 \end{array}$$

On associe à chaque possibilité un coefficient complexe. La probabilité d'avoir une valeur correspond au carré du module du coefficient. Ceci se note comme suit :

$$0.5 + 0i|.000.\rangle$$

La probabilité d'avoir 000 est de $|0.5 + 0i|^2 = 0.25$.

$$0.25 + 0.25i|.001.\rangle$$

La probabilité d'avoir 001 est de $|0.25 + 0.25i|^2$ et ainsi de suite...

Une opération sur notre registre quantique permet de modifier les coefficients. Il existe toute sorte de portes qui vont permettre de traiter les coefficients. Par exemple, la porte de Hadamard permet de positionner les coefficients de manière à ce que les différentes valeurs soient équiprobables.

On peut donc représenter un registre quantique de N qbits comme un vecteur de 2^N complexes. Chaque porte peut être représentée comme une matrice. L'application d'une porte sur un registre consiste alors à une simple multiplication. Par exemple, la porte de Hadamard d'une maniére physique est une lame demi-onde à 22,5 degrés. D'une manière plus pratique, on représente la porte de Hadamard sur un qbit (donc, un vecteur de 2 valeurs complexe) ainsi :

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

Parmis les portes les plus importante, citons aussi la porte cnot. cnot permet d'assigner la valeur d'un qbit en fonction des valeurs des autres qbits du registre. À partir du résultat de cette porte, on peut en déduire la valeur du registre. On peut donc, grâce à cette porte, récupérer la valeur des qbits sans avoir à lire le registre de départ.

Il existe aussi une porte permettant d'appliquer un déphasage sur les coefficient[1] si une condition est vérifée.

Grace à ces portes, on peut effectuer différents traitements et créer des algorithmes tels que Shor et Grover (que nous étudirons dans la partie sur la batterie de tests et les applications)

2.2 Problème de la création d'un ordinateur quantique

Les superpositions quantiques peuvent être non-séparables (on dit aussi *intriquées*), c'est-à-dire que seul l'état de l'ensemble du registre quantique est connu sans que l'on puisse déterminer l'état d'un seul qbit. Or, pour que les calculs n'échouent pas, il est très important de conserver la *cohérence*. Une décohérence peut intervenir dès que l'un des qbits interagit avec l'environnement. Ainsi, les qbits, tout en étant couplés entre eux, doivent être découplés du monde extérieur. Malheureusement, plus le nombre de qbits est grand, plus la probabilité d'interaction avec l'environnement est grande. Le problème de la décohérence est le principal obstacle à la réalisation d'ordinateurs quantiques de taille intéressante.

Les meilleures solutions actuelles utilisent la résonance magnétique nucléaire. Les qbits sont les états de spin d'atomes dans une molécule, qui peuvent prendre les valeurs $+1$ ou -1. Ils sont très bien isolés du monde extérieur et ont donc des temps de cohérence longs. Chaque spin a sa propre fréquence de résonance et peut donc être modifié par une impulsion bien choisie. De plus, les spins des différents noyaux sont couplés, et la fréquence d'un spin peut dépendre de celles des spins voisins. On peut donc réaliser des transformations conditionnelles des spins, ce qui est l'ingrédient de base des portes quantiques. Malheureusement, pour l'instant, on ne parvient au mieux à une dizaine ou une quinzaine de qbits.

[1] C'est a dire une rotation du coefficient

Chapitre 3

Technologies utilisées

Nous allons ici récapituler les technologies et outils que nous avons été amenés à employer et pourquoi elles ont été choisies. Nous nous attarderons aussi sur certaines normes de travail que nous nous sommes imposées afin de travailler efficacement en groupe.

Nos différents choix ont été guidés par certaines contraintes majeurs :

- Le groupe est composé de 11 personnes, ce qui implique que tout le monde doit connaître, de près ou de loin, toutes les technologies employées ou dû moins qu'elle soit compréhensibles et abordables facilement.
- Les outils doivent être libres ou gratuits afin que tout le monde puisse en bénéficier et que d'un point de vue éthique tout le monde y trouve son compte : la plupart des membres du groupe étant des utilisateurs de logiciels libres exclusivement.

3.1 Le C++

Le langage utilisé pour l'implémentation de la bibliothèque à été choisi afin de respecter plusieurs critères :

- une **bibliothèque standard** complète et optimisée permettant l'accès à des structures de données comme des listes, des vecteurs, des listes associatives, mais aussi des complexes et des matrices.
- nécessité d'un **langage objet**, car cela rend la modélisation, du moment que l'on est habitué à penser en objet, plus intuitive et plus propre. L'objet permet aussi une meilleure modularité et donc un meilleur découpage des tâches.
- l'interface d'utilisation de la bibliothèque devait être simple et accessible. Pour cela des techniques comme **la surcharge d'opérateurs ou de méthodes** peuvent grandement simplifier l'utilisation de la bibliothèque.

- la **portabilité** et le fait que le langage soit **communément utilisé** sont des atouts majeurs pour qu'une bibliothèque ait un quelconque succès.
- la **rapidité** est aussi importante car la bibliothèque est vouée à être la couche la plus basse du programme et donc être fréquemment appelée. C'est une partie critique qui doit être optimisée et rapide.

Le C a été écarté car, bien que rapide, ne possède ni bibliothèque standard décente, ni objets, ni surcharge. Ce qui aurait donné des programmes rapides mais longs et désagréables à lire, à écrire et à utiliser.

Le CAML quant à lui est aussi rapide et possède une bonne bibliothèque standard, mais est peu utilisé, de plus le fonctionnel n'est pas très recommandé pour ce genre d'application qui regorge d'effets de bords.

Eiffel aussi nous avait séduit par sa beauté mais il est peu utilisé et peu maîtrisé par les membres de l'équipe.

C'est donc le **C++** qu'il a été choisi d'utiliser car il correspond exactement à nos critères et est maîtrisé par tous les membres de l'équipe (et même plus par une grande partie de l'équipe).

3.2 Le SWIG et le Python

Une des difficultés rencontrées est de créer une bibliothèque rapide et agréable à utiliser. Pour cela nous avons opté pour du C++ en interne interfacé vers un langage plus agréable, et pour cela nous avons choisi Python.

Pour ce qui est de l'interfaçage entre le C++ et le Python le choix est assez restreint et seul SWIG convient techniquement à nos besoins.

Python a été choisi car c'est un langage de script dynamique, qui nous permet de ne pas recompiler les programmes à chaque modification. Evidement il est lent mais c'est justement pour cela qu'il ne sert que de surcouche à la bibliothèque. La bibliothèque reste utilisable en C++ directement et donc compilable si besoin est. Ainsi on peut imaginer la phase de développement de l'algorithme, rapide et agréable, en Python puis une conversion en C++ quand l'algorithme est au point.

Parmi les langages de script le choix s'est tourné sur Python car, il est bien plus agréable à lire et moins ambigü que le Perl, et plus utilisé que Ruby. De plus au sein du groupe Python est plus populaire et mieux maîtrisé.

3.3 Les Autotools

La chaîne de compilation est gérée à l'aide des Autotools (Automake, Autconf, ...), car une fois que l'on sait s'en servir l'écriture des `Makefile`'s et des outils

de compilation séparée est beaucoup plus rapide. De plus, le système garantit une bonne portabilité même vers des architectures anciennes.

Nous utilisons également des bibliothèques dynamiques et celles-ci sont réputées pour être difficilement portables, nous utilisons donc l'outil Libtool qui s'avère beaucoup plus facilement manipulable à travers les Autotools qu'à la main.

Une fois que l'on a pris l'habitude de les utiliser il devient difficile d'en faire autrement tellement cela permet de gagner du temps et de maximiser les chances d'être portable.

3.4 Le LaTeX

LaTeXest de loin l'outil le plus puissant pour rédiger des rapports scientifiques. Dès qu'il y a des formules mathématiques il devient pénible d'utiliser autre chose.

La présentation produite avec LaTeXest sobre et professionnelle ce qui est exactement ce que l'on attend d'un article scientifique. Il n'y a que très peu d'alternatives de ce côté là.

C'est un outil qui n'est pas très simple d'emploi au début, mais lorsqu'on a de la pratique il s'avère bien plus efficace et précis que toutes les autres solutions à base d'édition graphique (Word, OpenOffice, etc.) et bien plus souple que le HTML ou le Makeinfo.

Comme tous les membres sont habitués à utiliser le LaTeX, le choix a été sans appel.

3.5 Système de version

Dans un grand groupe il est difficile de pouvoir séparer les tâches à un tel point que personne ne travaille sur les mêmes fichiers. De plus travailler séparément demande souvent des phases de mise en commun du travail qui peuvent s'avérer très vite fastidieuses. Enfin un échange de fichiers continu, sans endroit de stockage commun est très vite intolérable dans un grand groupe de travail. Pour toutes ces raisons nous avons choisi d'utiliser un système de version performant. Le système retenu est Subversion, le plus récent et techniquement le plus abouti des systèmes libres.

Là aussi tout le groupe est habitué à travailler avec ce genre de système qui est devenu indispensable à tout le monde.

3.6 La mailing list

Pour que l'information passe le mieux possible une mailing liste a été créée à l'adresse `lambda2005@yahoogroupes.fr`. Le choix de l'hébergeur s'est fait rapidement pour pouvoir immédiatement communiquer et par la suite s'est avéré un très mauvais choix. En effet celui-ci accepte un mail sur deux et ne les renvoie pas immédiatement mais souvent avec des délais inégaux. Néanmoins cela reste un service gratuit.

La mailing liste nous a servi pour déterminer les horaires de réunion, informer tout le monde et archiver les informations importantes telles que les compte rendus et les recherches/informations trouvées lors de la phase de recherche. Elle a accueilli toutes les différentes mises à jour effectuées sur le dépôt Subversion que ce soit pour les rapports ou pour les codes source, ainsi chacun peut être à tout moment informé de l'activité des autres membres du groupe.

3.6.1 Organisation des mises à jour

Utiliser un tel système ne résout pas tous les problèmes, il faut tout de même respecter certaines règles pour que l'efficacité du groupe soit au maximum. Voici ce que nous avons décidé pour organiser les mises à jour.

Chaque mise à jour s'accompagne d'une entrée dans le ChangeLog. Le ChangeLog étant un fichier décrivant fichier par fichier toutes les modifications qui ont été faites et par qui. L'entrée du ChangeLog sert également de commentaire Subversion. Les conventions d'écriture du ChangeLog sont celles décrites par la norme GNU que l'on peut consulter sur `http://www.gnu.org/prep/standards_40.html`

Chacune de ces modifications doivent être le plus atomique possible. Si c'est une correction de bug, un ajout de fonctionnalité et un nouveau test écrit, cela doit faire l'objet de trois mises à jour distinctes, et pas une seule grosse mise à jour fourre-tout.

Cela permet de s'y retrouver dans le ChangeLog, et surtout, cela permet que chaque numéro de version corresponde à une modification précise. Si par la suite, on s'aperçoit par exemple que la correction du bug était inutile, il sera plus facile de la renverser si elle n'était pas fournie avec un gros patch qui modifiait également 300 autres fichiers.

Chaque mise à jour doit s'accompagner d'un post sur une mailing list. Le format est :

- Dans le sujet : faire figurer le numéro de version et le but du patch,
- commentaire. (optionnel).
- nouvelle entrée de ChangeLog,
- patch.

Voici un exemple :

```
From: Thomas Claveirole <clavei_t@lrde.epita.fr>
Subject: [lambda2005] lambda2005 [19] Add C-not and minor fixes.
To: lambda2005@yahoogroupes.fr
Date: Thu, 27 May 2004 22:14:26 +0200
Reply-To: lambda2005@yahoogroupes.fr

2004-05-27  Thomas Claveirole  <thomas.claveirole@lrde.epita.fr>

        * src/gates.hh, src/gates.cc: New. Source for quantum gates. Add cnot.
        * src/gates.i: New. SWIG interface for gates.

Index: src/gates.cc
===================================================================
--- src/gates.cc        (revision 0)
+++ src/gates.cc        (revision 0)
@@ -0,0 +1,21 @@
+#include <gates.hh>
+
+namespace qtm
+
+
+  namespace gates
+
+
+    void
+    cnot(qtm::reg& r, size_t target, size_t control)
+
+      typedef reg::register_data::iterator     iterator;
+
+      for (iterator i = r.data().begin(); i != r.data().end(); ++i)
+       i->first[target] = i->first[target] xor i->first[control];
+
+
+   // End of namespace gates.
+
+ // End of namespace qtm.
+
Index: src/gates.i
===================================================================
```

```
--- src/gates.i (revision 0)
+++ src/gates.i (revision 0)
@@ -0,0 +1,8 @@
+%module gates
+%
+#include "gates.hh"
+%
+
+%include qtm_exception.i
+%include gates.hh
+
Index: src/gates.hh
===================================================================
--- src/gates.hh        (revision 0)
+++ src/gates.hh        (revision 0)
@@ -0,0 +1,19 @@
+#ifndef QTM_GATES_HH
+# define QTM_GATES_HH
+
+# include <register.hh>
+
+namespace qtm
+
+
+  namespace gates
+
+
+    void
+    cnot(qtm::reg& r, size_t target, size_t control);
+
+   // End of namespace gates.
+
+ // End of namespace qtm.
+
+#endif // ! QTM_GATES_HH
```

Ensuite, il faut s'assurer que au moins une personne relise la mise à jour. Ce patch n'est pas là pour rien ! Pour cela le travail a été effectué par binômes. Lorsque il y a des erreurs ou des remarques à faire sur le contenu de la mise à jour, cela doit se faire de manière publique, et l'auteur de la mise à jour doit répondre pour informer les autres qu'il va régler la situation, même si cela consiste

en “OK en effet il y a une erreur, je corrige ça dans mon prochain patch.”

Bien entendu, tout le monde est tenu de rester au courant de l’avancement du projet et de lire *au moins* le commentaire et l’entrée de ChangeLog.

Evidement un script automatisant toute cette procédure a été écrit.

3.7 Coding style

Il y a une grande liberté dans la manière d’écrire du code source. Cela va du très lisible à l’incompréhensible. Dans l’optique d’un travail à plusieurs, il est nécessaire de se mettre d’accord sur une manière de coder homogène. On parle de “coding style”. C’est une sorte de règlement sur les choses à faire et à éviter.

Nous allons voir ici les principales recommandations.

- La première concerne le code inutile. Il est impératif de ne jamais rendre un code qui contient du code mis en commentaire ou qui n’est jamais appelé.
- La duplication du code est à éviter. Un code dupliqué est un code qui est exécuté deux fois moins, et qui donc est deux fois moins éprouvé. A ça s’ajoutent les problèmes de maintenance, les modifications doivent être effectuées à plusieurs endroits du projet, ce qui est propice aux oublis. Les outils spécifiques au langage C++ pour éviter la duplication de code sont l’héritage et les patrons de classes.
- L’utilisation de références constantes évite de nombreuses copies d’objet.
- Le choix entre référence et pointeur se fait ainsi : le choix d’un pointeur comme type signifie que l’on passe la responsabilité de la gestion de la mémoire avec l’objet. Par exemple, “new” retourne un pointeur, il indique qu’il se déresponsabilise de la gestion de la mémoire de l’objet créé. C’est l’appelant qui doit gérer la mémoire.
  Les identifiants commençant par un *underscore* (’`_`’) ne doivent pas être utilisés. Ils sont réservés au compilateur.
- Les noms des classes doivent être en minuscules.
  Les identifiants pour les variables ou les fonctions sont en minuscule et les mots qui les composent sont séparés par des *underscores* : `foo_get()`.
  Les membres de classe privés et protégés sont terminés par un *underscore*.
- Les alias de type créés par *typedef* doivent être du style : `toto_type`. Les identifiant du style `toto_t` sont réservés par *POSIX*.
- Lors d’un héritage, on définit un alias de type nommé `super` qui est en fait le type de la classe dont on hérite. C’est parfois utile.
- L’utilisation d’outils standard tels que “`for_each`”, “`find`”, “`find_if`”, “`transform`”, etc, sont à préférer par rapport à des boucles écrites à la main.
- Il est préférable d’utiliser les versions “méthodes” des algorithmes de la bibliothèque standard :

```
// oui
my_set.find (my_item);

// non
find (my_item, my_set.begin (), my_set.end ());
```

- Les différentes classes sont regroupées en “namespace” selon leur rôle.
- Les instanciations des types de la bibliothèque standard seront renommées en types personnalisés avec un nom explicite.
- Les déclarations des interfaces sont écrites dans les `.hh`, l’implémentation des fonctions en-ligne sera faite dans les `.hxx` et l’implémentation dans les `.cc`. Le découpage `.hh/.hxx` permet de ne pas polluer l’interface.
- Toutes les en-têtes doivent être protégées contre la double inclusion.
- Les membres des classes doivent être ordonnés par visibilité, d’abord les membres publiques, puis protégés puis enfin les privés.
- Chaque classe implémentera une fonction `print()` ainsi que la surcharge de l’opérateur << pour l’affichage et le débogage.
- Dans le code source, tous les identifiants, mais aussi les commentaires doivent être écris en anglais.
- Les commentaires sont écrits de telle manière qu’ils permettent de créer une documentation automatiquement, ceci grâce au logiciel “doxygen”. Ils doivent être concis et dans un style impératif. Les commentaires longs sont dans le style langage C, tandis que les commentaires d’une ligne sont du style langage C++.

Chapitre 4

Implémentation

Nous allons ici décrire l'interface de la bibliothèque afin de présenter nos différents choix d'implémentation. Le code fournit est en C++.

4.1 Les outils

La bibliothèque quantique s'articule sur deux outils mathématiques de base que sont les complexes et les matrices. Notre implémentation fournit donc à l'utilisateur toutes les fonctions dont il a besoin.

4.1.1 Les complexes

L'angle de rotation d'un qbit, appelé amplitude, est représenté sous forme complexe, cette valeur sera intensément modifiée par toutes les transformations que nous allons appliquer sur notre qbit.

La classe de complexe doit donc fournir des opérations de bases telles que l'addition, la soustraction, la multiplication et la division.

En plus de cela nous nous avons besoin de savoir la probabilité d'apparition d'une composante du qbit, celle-ci est le carré de l'amplitude (donc le carré d'un complexe).

Dans notre implémentation nous utilisons la classe complexe `std::complex<double>`. Celle-ci est présente dans la bibliothèque standard, sans bug, bien optimisée et maintenu à jour indépendamment de notre programme. De plus elle est portable ce qui très apréciable.

4.1.2 Les matrices

L'autre outil utilisé dans la bibliothèque est la classe matrice. Nous avons besoin de matrices pour tout ce qui est opération sur les qbits. Une opération sur un qbit est généralement un passage à travers une 'porte' représentée par une matrice.

Les opérations sur ces matrices sont simples et nous n'avons pas besoins d'une interface compliquée pour celles-ci. Les matrices que nous utiliserons sont des matrices de complexes. Voici celle utilisée :

```
/// Matrix class using T as data type.
template <typename T>
struct matrix
{
  /// Ctors.
  matrix(unsigned nlines, unsigned ncols);

  /// Multiplication.
  matrix<T>       operator *(const matrix<T> &m) const;

  /// Const data accessor.
  /// \arg i: line
  /// \arg j: col.
  const T         &operator ()(unsigned i, unsigned j) const;

  /// same as previous but using a point.
  const T         &operator [](const point &p) const;

  /// Non-const data accessor.
  /// \arg i: line
  /// \arg j: col.
  T               &operator ()(unsigned i, unsigned j);

  /// same as previous but using a point.
  T               &operator [](const point &p);

  /// Number of columns.
  unsigned ncols() const;

  /// Number of lines.
  unsigned nlines() const;

  ///Dtor
```

```
  ~matrix();
protected:
  T           *data_;
  unsigned    nlines_;
  unsigned    ncols_;
};
```

4.2 Les registres quantiques

Les registres quantiques représentent l'ensemble de notre mémoire. C'est le coeur de la bibliothèque. C'est sur ces registres que nous allons effectuer toutes les opérations.

4.2.1 Structure interne

Un registre est composé de toutes ses composantes. La principale idée est de ne stocker que les composantes dont l'amplitude (et donc la probabilité de lecture) est non nulle.

Chaque registre travaille sur un certain nombre de qbits et ne stocke que les composantes utiles. On a donc comme interface :

```
/// Quantum register.
struct reg
{
public:
  /// State of a register, i.e. 0000, 0001, 0010, ...
  typedef tools::bitfield                 state;

  /// Amplitude of a state, as a complex number.
  typedef std::complex<double>            amplitude;

  /// List of states and their corresponding amplitude.
  typedef std::vector< std::pair<state, amplitude> >
                                          register_data;

  /// Kind of ``hash'' from state to amplitude.
  typedef std::map<state, amplitude>      register_hash;

  enum { state_size = state::size };
```

```
public:
  /// Constructors.
  //@{
  reg(unsigned value = 0, unsigned size = state_size);
  reg(const reg& r);
  //@}

  /// Measure the i th bit, without destroying it.
  bool  operator [] (size_t i) const;

  /// Measure the i th bit, then destroy it.
  bool  operator () (size_t i);

  /// Measure the register.
  operator state () const;

  /// Get the register's size.
  size_t          size() const;

  /// Print the register.
  std::ostream&
  print(std::ostream& o) const;

  /// Accessors to data.
  ///@{
  const register_data&  data() const;
  register_data&                    data();
  //@}

  /// Build a ``hash'' from states to amplitude.
  register_hash hash() const;

  /// Check that the register has more than n bits.
  bool            is_valid(size_t n) const;

protected:
  /// Add a state and its amplitude to the list.
  void            state_add(const state s, const amplitude a);

otected:
size_t          size_;
```

```
register_data    data_;
};
```

Pour chaque composante le vecteur de qbit est implémenté grâce à un `std::bit_vector` qui est un vecteur ne contenant que des booléens, et dont chaque case tient sur un bit. Ceci va nous permettre d'accéder rapidement à chaque bit (car les opérations se font sur des bits en particuliers) avec un code propre (sans masques intempestifs).

4.2.2 Les opérations de bases

La clé d'un bon ordinateur quantique est de fournir les opérations élémentaires fréquemment utilisées sur les qbits.

Toutes les opérations (sauf la multiplication) sont prototypées selon cette forme :

```
void operation_name(reg&, target , [options]);
```

Le paramètre `target` est le bit cible que nous voulons modifier étant donné que toutes les opérations ne s'appliquent qu'à un bit.

Voici la liste des opérations que nous implémentons :

Kronecker product :

```
reg reg::operator*(const reg& rhs) const;
```

Cette fonction crée un nouveau registre résultant du produit de Kronecker. Le nouveau registre est obtenu à partir de la formule suivante :

Supposons que A est un vecteur à 2^n lignes et B un vecteur à 2^m lignes, alors nous avons cette représentation :

$$|.A.\rangle \otimes |.B.\rangle \equiv \begin{bmatrix} A_1B \\ A_2B \\ \dots \\ A_{2^n}B \end{bmatrix} = \begin{bmatrix} A_1B_1 \\ A_1B_2 \\ \dots \\ A_1B2^m \\ A_2B_1 \\ \dots \\ A_{2^n}B_{2^m} \end{bmatrix}$$

Controlled not :

```
void cnot(reg& r,
          int target,
          int control);
```

Cette opération à l'effet de la porte logique 'not' de façon contrôlée. C'est à dire que le qbit cible n'est inverti que si le qbit de contrôle est actif. Cette opération peut s'écrire comme la résultante de l'application par la matrice :

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

Toffoli (controlled controlled not) :

```
void ccnot(reg& r,
           int target,
           int control1,
           int control2);
```

Cette opération a le même effet que la précédente mais est contrôlée par deux qbits au lieu d'un seul. La matrice de transformation est :

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

Noter que cette fonction surcharge la précédente. Ceci est toujours dans un soucis de simplicité de l'interface de la bibliothèque. L'utilisateur doit pouvoir trouver les opérations de façon intuitive.

Unbounded Toffoli :

```
template <class Iterator>
void toffoli(reg& r,
             int target,
             Iterator ctrl_begin,
             Iterator ctrl_end);
```

Cette opération effectue le même travail que les deux précédentes mais peut avoir un nombre variable de qbits de contrôle. On récupère les bits de controle à l'aide d'un itérateur passé en argument.

Pauli Spin rotation :

```
void sigma_x(reg& r, int target);
void sigma_y(reg& r, int target);
void sigma_z(reg& r, int target);
```

Ces opérations produisent les rotations de Pauli définies comme ceci :

$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

Notez que σ_x n'est rien d'autre que l'opération logique not.

Rotation :

```
void rot_x(reg& r, int target, double gamma);
void rot_y(reg& r, int target, double gamma);
void rot_z(reg& r, int target, double gamma);
```

Ces opération effectuent une rotation selon les axes de la sphère de Bloch. L'angle de rotation est donné par `gamma`. Chaque rotation est la resultante de la transformation par la matrice correspondante suivante :

$$U_{R_x} = \begin{pmatrix} cos\frac{\gamma}{2} & -isin\frac{\gamma}{2} \\ -isin\frac{\gamma}{2} & cos\frac{\gamma}{2} \end{pmatrix} U_{R_y} = \begin{pmatrix} cos\frac{\gamma}{2} & sin\frac{\gamma}{2} \\ sin\frac{\gamma}{2} & cos\frac{\gamma}{2} \end{pmatrix} U_{R_z} = \begin{pmatrix} e^{-i\frac{\gamma}{2}} & 0 \\ 0 & e^{i\frac{\gamma}{2}} \end{pmatrix}$$

Global Phase :

```
void phase_scale(reg& r, int target, double gamma);
```

Cette opération ajoute une phase globale à un qbit, comme indiqué sur la matrice suivante :

$$U = \begin{pmatrix} e^{i\gamma} & 0 \\ 0 & e^{i\gamma} \end{pmatrix}$$

Phase kick :

```
void phase_kick(reg& r, int target, double gamma);
```

Cette opération produit un décalage de phase à un qbit, comme indiqué sur la matrice suivante :

$$U = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\gamma} \end{pmatrix}$$

Hadamard gate :

```
void hadamard(reg& r, int target);
```

Cette opération effectue la transformation par la porte de Hadamard. C'est une opération très utilisée, elle permet d'établir ou de supprimer la superposition d'un qbit. La matrice qui la définie est la suivante :

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

Walsh-Hadamard transform :

```
void walsh(reg& r, int width);
```

La transformation de Walsh-Hadamrd consiste à appliquer la transformation d'Hadamard sur les `width` premiers qbits.

Conditional phase shifts :

```
void cond_phase(reg& r, int control, int target);

void cond_phase(reg& r,
                int control,
                int gamma,
                int target);
```

Ces opérations effectuent des décalages de phase conditionnels.
La première opération est un décalage d'angle $\frac{\pi}{2^k}$ avec $k =$`control - target`.
La matrice est la suivante :

$$U = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i\frac{\pi}{2^k}} \end{pmatrix}$$

La deuxième opération est un décalage d'angle `gamma`. La matrice est la suivante :

$$U = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i\gamma} \end{pmatrix}$$

Arbitrary 1-bit operations :

```
template <class T>
void gate(reg& r, int target, const matrix<T>& m);
```

Cette opération offre une interface pour pouvoir appliquer une matrice personnalisée à un qbit. Pour cela on utilisera l'interface de la classe `matrix<T>` fournit par la bibliothèque.

4.2.3 Lecture d'une composante

Afin de mesurer la valeur d'un registre nous avons trois façons de procéder :
- en lisant le registre en entier, mais on ne voit alors qu'une seule des composantes déterminées selon les probabilités de chacunes ;
- en lisant un bit, celui ci est ensuite détruit par la lecture ;
- en lisant un bit mais en le préservant. Cette opération n'est pas possible sur un vrai ordinateur quantique.

Ces fonctions sont accessibles à travers l'interface suivante du registre :

```
/// Measure the i th bit, without destroying it.
bool        operator [] (size_t i) const;

/// Measure the i th bit, then destroy it.
bool        operator () (size_t i);

/// Measure the register.
operator state () const;
```

Comme on le voit ici aussi nous utilisons fortement la possibilité du C++ de surcharger des opérateurs. Le dernier opérateur est ce que l'on appelle un opérateur de cast.

4.3 Algorithmes de test

Nous avons implémenté quelques algorithmes quantiques dans le but de tester notre implémentation.

4.3.1 Shor

Découvert en 1994, il permet une amélioration exponentielle pour la factorisation. Son but est de factoriser un nombre N arbitraire. Si on arrivait à appliquer cet algorithme avec un nombre raisonnable de qbits, on pourrait casser le cryptage RSA [?] (utilisé dans quasiement toutes les méthode de cryptage)

Initialisons un registre quantique A de M qbits avec les 2^{M-1} valeurs (grâce à la porte de Hadamard). La probabilité d'obtenir n'importe quelle valeur entre 0 et 2^{M-1} est donc identique.

Soit N notre nombre à factoriser. On choisit X aléatoirement entre 1 et $N-1$. Puis, on calcul $X^A \mod N$ et le résultat est placé dans un registre quantique B. A ce moment, le registre quantique B contient tous les résultats possibles. Prenons par exemple $N = 15$ et $X = 2$. Les valeurs de B en fonction de A seront :

Registre A	Registre B
0	1
1	2
2	4
3	8
4	1
5	2
6	4
7	8
8	1
9	2
10	4
11	8
12	1
13	2
14	4
15	8

Remarquez que l'on voit apparaître une séquence dans le registre B : (1, 2, 4, 8). On nomme la période de répétition de la séquence f. Ici $f = 4$.

On remarque que si on a f, alors $P = X^{\frac{f}{2}} + / - 1$ est un facteur de N. Cet algorithme est connu depuis 300 avant JC.

Toute la difficulté réside dans le calcul de f. Or un ordinateur quantique est capable de trouver f avec une forte probabilité.

Pour cela, on applique la transformation de Fourier quantique [**?**] sur B. La transformation de Fourier quantique permet de trouver la période d'une fonction (en $O(1)$!). D'une manière pratique, on applique simplement une porte de changement de phase et une porte de hadamard.

Après l'application de la transformation de Fourier quantique, on obtient une approximation d'un multiple de $\frac{1}{f}$. Il suffit de mettre le résultat de la transformation sous forme irréductible puis de récupérer le dénominateur pour obtenir f. On peut alors facilement tester la solution. Si elle ne convient pas, on recommence avec une autre valeur de X.

4.3.2 Grover

En 1996, Grover [**?**] a mis au point un algorithme quantique permettant de faire une recherche dans une base de données non triée en $O\sqrt(N)$ (contre $O(N)$ pour un ordinateur habituel).

On peut théoriquement utiliser cet algorithme pour cracker du DES en testant toutes les combinaisons possibles. Pour une clef de 56 bits, il est nécessaire de tester 2^{55} combinaisons pour un ordinateur habituel. Avec l'algorithme de Grover, seulement 185 tests sont nécessaires.

Soit un système de $N = 2^n$ états nommés $S_1, \ldots S_n$. On commence par initialiser un registre quantique de n qbits de manière à avoir la même probabilité pour tous les N états.

Testons notre condition sur notre registre et appliquons une rotation de π si la condition est vérifiée[1]. On applique ensuite sur notre registre une transformation qui équivaut à une transformation de Hadamard puis, une rotation et une seconde transformation de Hadamard. On peut aussi définir cette transformation comme l'application d'une matrice D au registre telle que :

$$\begin{cases} D_{ij} = \frac{2}{N} & \text{si } i \neq j, \\ D_{ii} = -1 + \frac{2}{N} & \text{sinon.} \end{cases}$$

Appliquons le déphasage conditionnel et la transformation D suffisamment de fois.

Si la condition est vérifiée sur l'état final alors, l'état final est la solution avec une probabilité d'au moins $\frac{1}{2}$.

[1] la porte conditionnal_phase permet de faire ce traitement

Chapitre 5

Conclusion

Le projet s'inscrit dans la suite des bibliothèques quantiques déjà existantes, mais offre sûrement un plus haut niveau de facilité d'utilisation et des possibilités grâce à l'interfaçage vers des langages de scripts. En effet il est tout à fait envisageable de rajouter de nouveaux langages relativement facilement. Cette fonctionnalité ainsi que le soin apporté au code nous motive à publier la bibliothèque afin d'avoir un retour d'expérience, ainsi que des applications l'utilisant.

Pour être complète et fidèle à la réalité il manque cependant deux choses. Prendre en compte les effets de la décohérence, en simulant une rotation aléatoire sur l'axe z. Celle-ci serait déclenchée à chaque opération sur les registres. Et dans le même ordre d'idée, la correction d'erreurs pour minimiser les effets de la décohérence.

Évidemment on pourrait aussi ajouter un plus grand nombre d'algorithmes proposés par la bibliothèque comme une transformation de Fourier quantique ou le calcul de $f(a) = x^a \mod N$.