ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ
ΥΠΟΛΟΓΙΣΤΩΝ

Προσομοιωτής Κβαντικού Υπολογιστή

με έμφαση στην Κβαντική Βελτιστοποίηση
και στα Κβαντικά Νευρωνικά Δίκτυα

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Παγώνη Α. Αλεξάνδρου

Επιβλέπων: Αριστείδης Θ. Παγουρτζής
Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούνιος 2022

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ
ΥΠΟΛΟΓΙΣΤΩΝ

Προσομοιωτής Κβαντικού Υπολογιστή

με έμφαση στην Κβαντική Βελτιστοποίηση
και στα Κβαντικά Νευρωνικά Δίκτυα

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Παγώνη Α. Αλεξάνδρου

.....	.....	.....
Αριστείδης Παγουρτζής	Δημήτριος Φωτάκης	Μεσσαριτάκης Χάρης
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Καθηγητής Παν. Αιγαίου

.....

Παγώνης Α. Αλέξανδρος

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών ΕΜΠ

Προσομοιωτής Κβαντικού Υπολογιστή

Σύνοψη

Οι κβαντικοί υπολογιστές είναι σε πρώιμο στάδιο και δεν μπορούμε να εκτελέσουμε σημαντικούς αλγορίθμους σε αυτούς. Για αυτό τον λόγο, στην πραγματικότητα δουλεύουμε με προσομοιωτές κβαντικών κυκλωμάτων.

Στην παρούσα διπλωματική εργασία κατασκευάζουμε έναν προσομοιωτή κβαντικών κυκλωμάτων. Ο προσομοιωτής αυτός είναι γραμμένος στην γλώσσα προγραμματισμού C++, είναι πλήρης και κύριο σκοπό έχει την υψηλή ταχύτητα.

Σχεδιαστικές αρχές του είναι: η ευχρηστία του (χρησιμοποιήθηκε το εύκολο συντακτικό της PennyLane), η συμβατότητά του (δεν βασίζεται σε καμία βιβλιοθήκη πέραν της βασικής βιβλιοθήκης της C++), η εύκολη και αποδοτική βελτιστοποίηση κβαντικών κυκλωμάτων, η ευελιξία του και η αρχή του ανοιχτού λογισμικού.

Στην βάση του υπάρχει μια βιβλιοθήκη μιγαδικών αριθμών και γραμμικής άλγεβρας· πάνω σε αυτή τη βάση γράφονται οι αλγόριθμοι που χρειάζονται για την προσομοίωση του κυκλώματος καθώς και για την αποδοτική βελτιστοποίησή του. Τέλος, υπάρχουν αρκετά εργαλεία που δίνουν στον χρήστη την δυνατότητα να εκτελεί πειράματα και να βελτιστοποιεί τα κυκλώματά του αρκετά εύκολα.

Λέξεις-Κλειδιά: Κβαντικός Υπολογιστής, Προσομοιωτής, C++, Κβαντικό Κύκλωμα, Βελτιστοποίηση, Qiskit, PennyLane, Qubit

Quantum Circuit Simulator

Abstract

Quantum Computers are in an early stage of development and we cannot execute meaningful algorithms on them. That is why, in reality we work with Quantum Circuit Simulators.

In this thesis we construct a Quantum Circuit Simulator. This Simulator is written in the C++ programming language, it is complete and it aims for performance.

Its basic Design principles are: user-friendliness (a syntax similar to Pennylane's was used), compatibility (the simulator only uses the basic C++ library), easy and efficient circuit optimization, flexibility and open-source.

In its core there is a library for complex numbers and linear algebra; on top of this core, there are algorithms needed for the circuit simulation and its efficient optimization. At last, many tools are provided to the user, giving him/her the ability to conduct experiments and optimize his/her circuits.

Keywords: Quantum Computing, Simulator, C++, Quantum Circuit, Optimization, Qiskit, PennyLane, Qubit

Ευχαριστίες

Θα ήθελα να ευχαριστίσω τον Ιωάννη Θεοδώνη, που με εισήγαγε στον κβαντικό υπολογισμό και με βοήθησε σε όλη μου την πορεία. Επίσης, θα ήθελα να ευχαριστήστω τον αδερφό μου Κωνσταντίνο Παγώνη για την βοήθειά του στις εικονογραφήσεις καθώς και για τις συμβουλές του. Τέλος, θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή κ. Αριστείδη Παγουρτζή, που μου έδειξε εμπιστοσύνη σε αυτό το δύσκολο πόνημα (τόσο στην επιλογή θέματος, όσο και στην εκτέλεση).

Περιεχόμενα

Κατάλογος Σχημάτων

1 Κβαντική Φυσική

1.1 Εισαγωγή

1.1.1 Η φυσική στον 19ο αιώνα

Στα τέλη του 19ου αιώνα η Φυσική είχε σημειώσει τρομακτική πρόοδο. Η κλασική μηχανική είχε έναν ισχυρό μαθηματικό φορμαλισμό και ο ηλεκτρομαγνητισμός είχε πλέον μια κομψή, και κατά τα φαινόμενα, πλήρη περιγραφή από τις εξισώσεις του Maxwell. Ένας άλλος κλάδος της όμως, η θερμοδυναμική, χρησιμοποίησε ένα νέο εργαλείο: την **στατιστική**. Αυτή ήταν η γέφυρα προν τον μικρόκοσμο. Και πράγματι, πολύ γρήγορα φάνηκε πώς η εικόνα μας για τον μικρόκοσμο ήταν λανθασμένη.

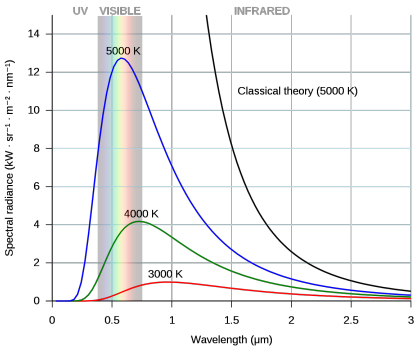
1.1.2 Η αναγκαιότητα της κβαντικής θεωρίας

Οι δύο μεγάλες φυσικές θεωρίες του 19ου αιώνα, ο ηλεκτρομαγνητισμός και η θερμοδυναμική, έδωσαν εξηγήσεις για τα, ως τα τότε, πιο απατηλά φαινόμενα: το φώς και την θερμότητα. Παρά όμως τις σπουδαίες τους επιτυχίες, δεν άργησε να επέλθει η κρίση: Οι θεωρίες της εποχής, προέβλεπαν πως ένα σώμα σε θερμική ισορροπία (με το περιβάλλον) θα έπρεπε να εκπέμπει απεριόριστα ποσά ενέργειας στις υψηλές συχνότητες, κάτι που δεν συμφωνεί με την πραγματικότητα.

Σχήμα 1: Ακτινοβολία λόγω θερμότητος

1.1.3 Καταστροφή του Υπεριώδους

Το όνομα «**Καταστροφή του Υπεριώδους**» δείχνει πόσο σοβαρό ήταν το πλήγμα για τις θεωρίες αυτές. Η λανθασμένη παραδοχή ήταν πως η ύλη στην μικρή κλίμακα μπορεί να ταλαντώνεται σε όλο το φάσμα των συχνοτήτων. Προσπαθώντας να λύσει το πρόβλημα της ασυμφωνίας αυτής, ο **Max Planck** υπέθεσε διακριτό φάσμα συχνοτήτων και το αποτέλεσμα των μαθηματικών συμφωνούσε με τις παρατηρήσεις. Η πρώτη αυτή ιδέα του Planck, πως η ενέργεια μεταφέρεται σε μικρά δέματα (quanta), μας εισάγει στην Κβαντική Φυσική.



Σχήμα 2: Καταστροφή του Υπεριώδους

Φυσικά η επιστήμη προχώρησε πολύ μακρύτερα από τον Planck. Σύντομα ακολούθησε η εργασία του Einstein για το φωτοηλεκτρικό φαινόμενο και μέσα σε 30 χρόνια η κβαντική φυσική είχε δείξει πως ο μικρόκοσμος δεν υπακούει στους «νόμους» της καθημερινής μας εμπειρίας.

1.2 Αιτιοκρατία

1.2.1 «Ο Θεός δεν παίζει ζάρια»

Τα νέα πειράματα έδειξαν πως στον μικρόκοσμο δεν ισχύει **αιτιοκρατία** (ή ισχύει εν μέρει). Η πρώτη εξίσωση που περιέγραφε πειστικά τα σωματίδια στον μικρόκοσμο (η εξίσωση του **Schrödinger**) στηριζόταν ακριβώς σε αυτό. Βασικά φυσικά μεγέθη, όπως η θέση του σωματιδίου, δεν είναι πλήρως καθορισμένα, και αυτό όχι λόγω αδυναμίας διεξαγωγής ακριβούς μετρήσεως, αλλά από την **εγγενή πιθανοκρατική φύση** των σωματιδίων. Η ερμηνεία αυτή αντίκειται στην καθημερινή μας εμπειρία, καθώς ο πιθανοκρατικός χαρακτήρας δεν ανάγεται στην μεγάλη κλίμακα και ως εκ τούτου δεν παρατηρείται υπό κανονικές συνθήκες.

1.2.2 Εξίσωση κίνησης - Εξίσωση Schrödinger

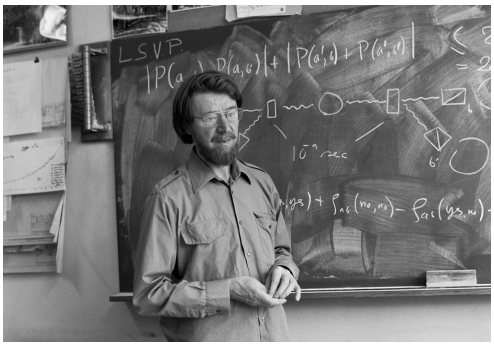
$$i\hbar\frac{\partial\psi}{\partial t} = \left(-\frac{\hbar^2}{2m}\nabla^2 + V(\vec{r})\right)\psi$$

Στην κλασική μηχανική όλα εξαρτώνται από την θέση του σώματος. Γνωρίζοντας την θέση $\vec{r}(t)$ κάθε στιγμή βρίσκουμε την ταχύτητα $\vec{u}(t)$, από εκεί την ορμή, την συνισταμένη δύναμη, τις ενέργειες κλπ. Στην κβαντομηχανική η πρώτη εξίσωση κίνησης (που κατέστρωσε ο Schrödinger [1] και φέρει το όνομα του), δεν μας υπολογίζει την ακριβή θέση ενός σωματιδίου. Η λύση ψ της εξίσωσης, γνωστή και ως **κυματοσυνάρτηση** , χρειάζεται μιαν ερμηνεία η οποία δεν είναι προφανής. Η ερμηνεία που δόθηκε έμοιαζε (και ακόμα μοιάζει) εντελώς αυθαίρετη, ωστόσο η «πραγματικότητα» την επαληθεύει.

1.2.3 Ερμηνεία κυματοσυνάρτησης

Η $|\psi|^2 = \psi^*\psi(\vec{r},t)$ είναι η **συνάρτηση πυκνότητας πιθανότητας** υπάρξης του σωματιδίου στην θέση $\vec{r}$ την στιγμή t. Αυτό σημαίνει οτι το σωματίδιο ακολουθεί μια στατιστική κατανομή και όταν μετρηθεί θα εντοπιστεί στην περιοχή V με πιθανότητα:

$$\iiint_V (\psi^*\psi)dV.$$



Σχήμα 3: John Bell

Ο ίδιος ο Einstein δεν δεχόταν την στατιστική αυτή ερμηνεία της φυσικής· χαρακτηριστική είναι η φράση του «Ο Θεός δεν παίζει ζάρια». Πίστευε, όπως και κάθε σκεπτικός άνθρωπος, πως η τυχαιότητα των μετρήσεων μας δεν μπορεί να οφείλεται σε πραγματική/εγγενή **τυχαιότητα**, αλλά σε κάποιον μηχανισμό άγνωστο προς το παρόν σε εμάς ο οποίος δρά απαρατήρητος και παράγει διαφορετικά αποτελέσματα. Η ιδέα πως το θεμέλιο της πραγματικότητας συμπεριφέρεται τυχαία μοιάζει εντελώς

παράλογη. Ένα από τα σημαντικότερα επιτεύγματα της Κβαντικής Φυσικής είναι αυτό του John Bell, ο οποίος το 1964 πρότεινε ένα ευφυές πείραμα με το οποίο θα φαινόταν αν η τυχαιότητα είναι εγγενής ή όχι. Όπως έδειξαν τα πειράματα, ο Einstein είχε άδικο.

1.3 Ο συμβολισμός του Dirac

1.3.1 Επαλληλία καταστάσεων

Άν δύο συναρτήσεις ψ_1 και ψ_2 είναι λύσεις της εξίσωσης Schrödinger, τότε και κάθε **γραμμικός συνδυασμός** $\alpha\psi_1 + \beta\psi_2$ είναι επίσης λύση. Η γραμμικότητα αυτή των λύσεων θυμίζει διανυσματικό χώρο. Και πράγματι θα ήταν πολύ χρήσιμο να είχαμε έναν χώρο καταστάσεων, όπου οι συναρτήσεις αναπαρίστανται από διανύσματα. Έτσι η κατανόησή μας για ένα σύστημα θα ήταν πιο οργανωμένη και πιο εύχρηστη. Για να έχουμε όμως έναν διανυσματικό χώρο πρέπει να έχουμε ορίσει ένα εσωτερικό γινόμενο μεταξύ των διανυσμάτων - δηλαδή, ένα μέτρο ομοιότητας μεταξύ των καταστάσεων/συναρτήσεών μας.

1.3.2 Συνθήκη Κανονικοποίησης

Μιας και η $|\psi|^2$ είναι συνάρτηση πυκνότητας πιθανότητας της θέσης του σωματιδίου πρέπει να δίνει σε όλο τον χώρο πιθανότητα 1 για την εύρεση του σωματιδίου (αφού το σωματίδιο de facto κάπου θα εμφανιστεί). Ο περιορισμός αυτός ονομάζεται **συνθήκη κανονικοποίησης**:

$$\boxed{\iiint_{\mathbb{R}^3} |\psi|^2 dV = 1} \tag{1}$$

Φυσιολογικά λοιπόν προκύπτει ο **ορισμός του εσωτερικού γινομένου**:

$$\boxed{(\phi, \psi) = \iiint_{\mathbb{R}^3} \phi^* \psi \quad dV} \tag{2}$$

Η σχέση αυτή ορίζει εσωτερικό γινόμενο, αφού πληροί τις προϋποθέσεις:

- $(\phi, \psi)^* = (\psi, \phi)$
- $(\phi, \lambda_1\psi_1 + \lambda_2\psi_2) = \lambda_1(\phi, \psi_1) + \lambda_2(\phi, \psi_2)$
- $(\psi, \psi) \geq 0$ και $(\psi, \psi) = 0 \iff \psi = 0$

1.3.3 Συμβολισμός Dirac

Για αρχή ορίζουμε μια ορθοκανονική **βάση** στον χώρο των καταστάσεων $\{\psi_i\}$ με $(\psi_i, \psi_j) = \delta_{ij}$. Τότε η τυχούσα κατάσταση θα ορίζεται ως:

$$\psi = \sum_i c_i \psi_i \tag{3}$$

ή στην γλώσσα της γραμμικής άλγεβρας:

$$\psi = \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix} \tag{4}$$

Είναι καιρός να παρουσιάσουμε τον συμβολισμό του Dirac με τον οποίο δουλεύει η κβαντομηχανική:

- Οι καταστάσεις συμβολίζονται ως: $|\psi\rangle \equiv \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix}$
- Ορίζεται το βοηθητικό ανάστροφο και συζυγές διάνυσμα: $\langle\psi| \equiv |\psi\rangle^\dagger \equiv \begin{pmatrix} c_1^* & \cdots & c_n^* \end{pmatrix}$
- Το εσωτερικό γινόμενο συμβολίζεται ως: $\langle\psi|\phi\rangle \equiv \langle\psi|\,|\phi\rangle$

1.4 Παρατηρούμενα Μεγέθη - Τελεστές - Μετρήσεις

Η συνάρτηση αντιστοιχεί τιμές σε άλλες τιμές. Ο τελεστής κάνει το ίδιο πράγμα σε συναρτήσεις. Πχ ο τελεστής παραγώγισης δρα πάνω στις παραγωγίσιμες συναρτήσεις και δημιουργεί άλλες συναρτήσεις: $\hat{D}_x f(x) = \frac{d}{dx}\{f(x)\} = f'(x)$. Εμείς θα περιοριστούμε σε γραμμικούς τελεστές, δηλαδή τελεστές με τις ιδιότητες:

$$\begin{aligned} \hat{A}(\lambda_1 |\psi_1\rangle + \lambda_2 |\psi_2\rangle) &= \lambda_1 \hat{A} |\psi_1\rangle + \lambda_2 \hat{A} |\psi_2\rangle \\ (\lambda_1 \langle\phi_1| + \lambda_2 \langle\phi_2|)\hat{A} &= \lambda_1 \langle\phi_1| \hat{A} + \lambda_2 \langle\phi_2| \hat{A} \end{aligned} \tag{5}$$

Όπως φαίνεται στην σύνθετη έκφραση $\langle\phi| \hat{A} |\phi\rangle$ δεν χρειάζονται παρενθέσεις καθώς $(\langle\phi| \hat{A})\,|\phi\rangle \equiv \langle\phi|\,(\hat{A}\,|\phi\rangle)$. Οι καταστάσεις $|\psi\rangle$ είναι απροσπέλαστες σε εμάς. Την στιγμή της μέτρησης η $|\psi\rangle$ καταρρέει σε μια τιμή. Πριν την μέτρηση δεν μπορούμε να προβλέψουμε το αποτέλεσμα, παρά μόνο την μέση τιμή του. Για να εξάγουμε την μέση (ή αναμενόμενη) τιμή για ένα μέγεθος πρέπει να βρούμε τον αντίστοιχο τελεστή $\hat{A}$ και να υπολογίσουμε την ποσότητα $\boxed{\langle\psi| \hat{A} |\psi\rangle}$. Οι τελεστές αυτοί για να αντιστοιχούν σε παρατηρήσιμα μεγέθη πρέπει να έχουν πραγματικές ιδιοτιμές, δηλαδή να είναι Ερμιτιανοί: [1]

$$\hat{A} = \hat{A}^\dagger \tag{6}$$

[1] **Ερμιτιανοί ή αυτοσυζυγείς** λέγονται οι πίνακες που ικανοποιούν την σχέση: $\hat{A}_{nm} = \hat{A}^*_{mn}$. Ισοδύναμα είναι οι τελεστές που ικανοποιούν την $\langle\psi|\hat{A}\phi\rangle \equiv \langle\hat{A}\psi|\phi\rangle\,, \quad \forall\phi,\psi$

2 Κβαντικοί Υπολογιστές

2.1 Εισαγωγή - Qubit

Κατ' αναλογία με τους κλασικούς υπολογιστές, ορίζουμε μια βάση στον χώρο των καταστάσεων (γνωστή ως «υπολογιστική βάση»): τις καταστάσεις $|0\rangle$ και $|1\rangle$. Οι καταστάσεις αυτές αντιπροσωπεύουν μιαν ορθοκανονική **βάση** 2 διαστάσεων και αντιστοιχούν στα συνήθη διανύσματα:

$$|0\rangle = \begin{pmatrix}1\\0\end{pmatrix} \qquad |1\rangle = \begin{pmatrix}0\\1\end{pmatrix} \tag{7}$$

Ένα κβαντικό-bit (**Qubit**) [2] ορίζεται ως ο γραμμικός συνδυασμός των διανυσμάτων βάσης:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle = \begin{pmatrix}\alpha\\\beta\end{pmatrix}, \quad \alpha,\beta \in \mathbb{C} \tag{8}$$

2.2 Υπέρθεση - Σφαίρα Bloch

2.2.1 Κανονικοποίηση

Η φυσική σημασία είναι η ίδια με αυτή της κβαντικής φυσικής: δηλαδή, αν έχουμε το Qubit $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$, τότε λόγω **κανονικοποίησης** (επειδή η μέτρηση θα δώσει μια απο τις δύο καταστάσεις):

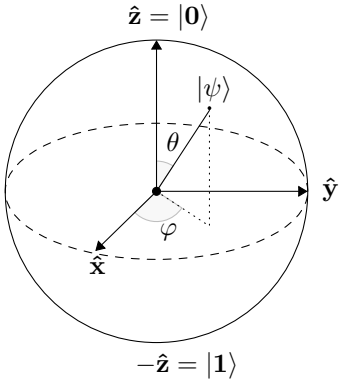
$$\begin{aligned}
&1 = \langle\psi|\psi\rangle \implies \\
&1 = (\alpha^*\langle 0| + \beta^*\langle 1|)(\alpha^*|0\rangle + \beta^*|1\rangle) \implies \\
&1 = |\alpha|^2\langle 0|0\rangle + |\beta|^2\langle 1|1\rangle + \alpha^*\beta\langle 0|1\rangle + \alpha\beta^*\langle 1|0\rangle \overset{\langle\psi_i|\psi_j\rangle=\delta_{ij}}{\implies} \\
&1 = |\alpha|^2 + |\beta|^2
\end{aligned} \tag{9}$$

2.2.2 Υπέρθεση

Το Qubit λοιπόν μπορεί να βρίσκεται μεταξύ των καταστάσεων $|0\rangle$ και $|1\rangle$, αλλά όταν μετρηθεί μόνο μια από τις δύο θα εμφανιστεί. Η ιδιότητα αυτή είναι γνωστή ως **«Υπέρθεση»** και μας δίνει το πλεονέκτημα έναντι των κλασικών υπολογιστών. Τα καλά της αποτελέσματα θα φανούν αργότερα, όταν συνδυαστεί με την δεύτερη βασική ιδιότητα των κβαντικών καταστάσεων: την συμπλοκή.

2.2.3 Σφαίρα του Bloch

Αν χρησιμοποιήσουμε την πολική μορφή των μιγαδικών αριθμών, μια κάτασταση του ενός Qubit μπορεί να γραφεί



Σχήμα 4: Σφαίρα του Bloch

ως:

$$|\psi\rangle = \alpha e^{i\chi} |0\rangle + \beta e^{i\phi} |1\rangle$$

Με βάση την αρχή της κανονικοποίησης: $\boxed{\beta = +\sqrt{1-\alpha^2}}$. Οπότε μπορούμε να αφαιρέσουμε μία μεταβλητή. Στην πραγματικότητα δεν μπορούμε να μετράμε τις κβαντικές καταστάσεις και η καλύτερη μας πρόβλεψη είναι οι αναμενόμενες τιμές των τελεστών που εφαρμόζονται πάνω σε αυτές $\langle A \rangle = \langle \psi | \hat{A} | \psi \rangle$:

$$\langle A \rangle \overset{(A_{ij}=A^*_{ji})}{==} \alpha^2 A_{00} + \beta^2 A_{11} + 2\alpha\beta Re(A_{10} e^{i(\chi-\phi)})$$

Παρατηρούμε οτι η αναμενόμενη τιμή εξαρτάται από την διαφορά φάσεων $(\chi - \phi)$, άρα μπορούμε να αφαιρέσουμε και πάλι μια μεταβλητή. Συμφέρει, λοιπόν, πολλές φορές να χρησιμοποιούμε την ισοδύναμη αναπαράσταση:

$$|\psi\rangle = \cancel{e^{i\delta}} \left(\cos(\frac{\theta}{2}) |0\rangle + e^{i\phi} \sin(\frac{\theta}{2}) |1\rangle \right)$$

όπου η γωνία δ (γνωστή και ως ολική γωνία) μπορεί να διαγραφεί, αφού δεν πρόκειται ποτέ να επηρεάσει τις πιθανότητες του Qubit.

Οι καταστάσεις-διανύσματα λοιπόν, έχουν μέτρο 1 και ανεξαρτησία κινήσεων σε 2 διαστάσεις. Γι΄ αυτό το λόγο μπορούμε να τις απεικονίσουμε γεωμετρικά με την επιφάνεια μιας μοναδιαίας σφαίρας **(Σφαίρα του Bloch)**. Η σύμβαση είναι να τοποθετούμε τις καταστάσεις της βάσης στον άξονα z (όπως τις μετρήσεις του spin). Στον βόρειο πόλο τοποθετούμε την κατάσταση $|0\rangle$ και στον νότιο την $|1\rangle$. Η τυχούσα κατάσταση αντιστοιχεί σε ένα σημείο πάνω στην επιφάνεια της σφαίρας. Για να μετακινήσουμε το σημείο αυτό σε άλλη θέση, εφαρμόζουμε πίνακες στροφής γύρω από τους 3 (ή και άλλους) άξονες.

Οι γωνίες θ και ϕ είναι η πολική γωνία και το αζιμούθιο στην σφαίρα.

2.3 Συμπλοκή - Καταστάσεις Bell

2.3.1 Δύο Qubits - Τανυστικό Γινόμενο

Μέχρι στιγμής περιγράψαμε τον χώρο καταστάσεων του ενός Qubit. Αλλά τί συμβαίνει όταν έχουμε 2 Qubits; Αρχικά πρέπει να κάνουμε μια επέκταση του χώρου για να μπορούμε να περιγράφουμε μεγαλύτερο αριθμό από Qubits. Ας υποθέσουμε πως έχουμε 2 Qubits στις καταστάσεις:

$$\begin{aligned} |\psi_1\rangle &= \alpha |0\rangle + \beta |1\rangle \\ |\psi_2\rangle &= \gamma |0\rangle + \delta |1\rangle \end{aligned} \tag{10}$$

Ο ζητούμενος χώρος θα έχει διάσταση $4 = 2 \times 2$ και η κατάσταση, που περιγράφει το **όλο** σύστημα, υπολογίζεται ως το **τανυστικό γινόμενο** μεταξύ των δύο διανυσμάτων:

$$|\psi\rangle = |\psi_1\rangle \otimes |\psi_2\rangle \tag{11}$$

το οποίο εξ ορισμού είναι το:

$$\begin{pmatrix}\alpha\\\beta\end{pmatrix} \otimes \begin{pmatrix}\gamma\\\delta\end{pmatrix} \equiv \begin{pmatrix}\alpha\begin{pmatrix}\gamma\\\delta\end{pmatrix}\\\beta\begin{pmatrix}\gamma\\\delta\end{pmatrix}\end{pmatrix} = \begin{pmatrix}\alpha\gamma\\\alpha\delta\\\beta\gamma\\\beta\delta\end{pmatrix} \tag{12}$$

Για παράδειγμα, αν το πρώτο Qubit είναι στην κατάσταση $|0\rangle$ και το δεύτερο στην $|1\rangle$, τότε το όλο σύστημα είναι στην:

$$|0\rangle \otimes |1\rangle = \begin{pmatrix}1\\0\end{pmatrix} \otimes \begin{pmatrix}0\\1\end{pmatrix} = \begin{pmatrix}0\\1\\0\\0\end{pmatrix} \tag{13}$$

Ως διανύσματα βάσης προφανώς θα θέσουμε τα τανυστικά γινόμενα των διανυσμάτων βάσης, δηλαδή τα:

$$\begin{aligned} |00\rangle = |0\rangle \otimes |0\rangle = \begin{pmatrix}1\\0\\0\\0\end{pmatrix} \qquad |01\rangle = |0\rangle \otimes |1\rangle = \begin{pmatrix}0\\1\\0\\0\end{pmatrix} \\ |10\rangle = |1\rangle \otimes |0\rangle = \begin{pmatrix}0\\0\\1\\0\end{pmatrix} \qquad |11\rangle = |1\rangle \otimes |1\rangle = \begin{pmatrix}0\\0\\0\\1\end{pmatrix} \end{aligned} \tag{14}$$

Εύκολα αυτή η ιδέα γενικεύεται σε περισσότερα Qubits (μιας και το τανυστικό γινόμενο είναι προσεταιριστική πράξη).

2.3.2 Συμπλοκή

Ας δούμε τώρα το τανυστικό γινόμενο εκ του αντιστρόφου. Ας υποθέσουμε πως ένα σύστημα βρίσκεται στην κατάσταση:

$$\frac{|00\rangle + |11\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\0\\0\\1\end{pmatrix} \tag{15}$$

Προσπαθώντας να αναλύσουμε την κατάσταση στα 2 συστατικά της Qubits διαπιστώνουμε οτι μια τέτοια αναγωγή είναι αδύνατη:

$$\nexists \alpha, \beta, \gamma, \delta \in \mathbb{C} : \qquad \frac{1}{\sqrt{2}}\begin{pmatrix}1\\0\\0\\1\end{pmatrix} = \begin{pmatrix}\alpha \times \gamma\\\alpha \times \delta\\\beta \times \gamma\\\beta \times \delta\end{pmatrix} \tag{16}$$

Δηλαδή, τα Qubits είναι «ενωμένα» με τέτοιο τρόπο που δεν επιτρέπεται η απόπλεξή τους. Το παράδειγμά μας ισοδυναμεί με το να πούμε οτι στην ρίψη δύο νομισμάτων θα έρθουν και τα δύο κορώνα ή και τα δύο γράμματα, ενώ αποκλείονται οι άλλοι δύο συνδυασμοί. Τα Qubits, δηλαδή, συμπλέκονται σε καταστάσεις και μπορούμε να τις επεξεργαστούμε στην ολότητά τους επηρεάζοντας μόνο ένα από αυτά. Η ιδιότητα αυτή μας δίνει μιαν μορφή παραλληλίας πράξεων στο ίδιο «υλικό».

2.3.3 Πολλά Qubits

Οι ιδέες αυτές εύκολα επεκτείνονται στα n Qubits. Το τανυστικό γινόμενο n δισδιάστατων διανυσμάτων είναι εξ ορισμού:

$$\begin{pmatrix}\alpha_1\\ \beta_1\end{pmatrix}\otimes\begin{pmatrix}\alpha_2\\ \beta_2\end{pmatrix}\otimes\cdots\otimes\begin{pmatrix}\alpha_n\\ \beta_n\end{pmatrix}=\begin{pmatrix}\alpha_1\alpha_2\cdots\alpha_{n-1}\alpha_n\\ \alpha_1\alpha_2\cdots\alpha_{n-1}\beta_n\\ \alpha_1\alpha_2\cdots\beta_{n-1}\alpha_n\\ \vdots\\ \beta_1\beta_3\cdots\beta_{n-1}\alpha_n\\ \beta_1\beta_3\cdots\beta_{n-1}\beta_n\end{pmatrix} \tag{17}$$

ή στον συμβολισμό του Dirac:

$$|\Psi\rangle=\bigotimes_{i=1}^{n}|\psi_i\rangle \tag{18}$$

Ανάλογα ορίζουμε και την βάση του χώρου $\{|b_i\rangle\}$:

$$\begin{aligned}|0\cdots0\rangle=|0\rangle\otimes|0\rangle\otimes\cdots\otimes|0\rangle=\begin{pmatrix}1\\0\\\vdots\\0\end{pmatrix}\qquad |0\cdots1\rangle=|0\rangle\otimes|0\rangle\otimes\cdots\otimes|1\rangle=\begin{pmatrix}0\\1\\\vdots\\0\end{pmatrix}\\ \vdots\\ |1\cdots0\rangle=|1\rangle\otimes|1\rangle\otimes\cdots\otimes|0\rangle=\begin{pmatrix}0\\\vdots\\1\\0\end{pmatrix}\qquad |1\cdots1\rangle=|1\rangle\otimes|1\rangle\otimes\cdots\otimes|1\rangle=\begin{pmatrix}0\\\vdots\\0\\1\end{pmatrix}\end{aligned} \tag{19}$$

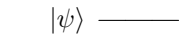
και η τυχούσα κατάσταση γράφεται ως γραμμικός συνδυασμός των διανυσμάτων βάσης:

$$|\Psi\rangle=\sum_{i=0}^{2^n-1}c_i|b_i\rangle\qquad \text{με}\ \sum_{i=0}^{2^n-1}|c_i|^2=1 \tag{20}$$

2.4 Κυκλώματα - Πύλες

2.4.1 Αρχές

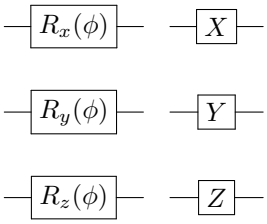
Τα κβαντικά κυκλώματα, όπως τα ηλεκτρικά κυκλώματα, είναι ένας τρόπος απεικόνισης ενός κβαντικού υπολογισμού με «καλώδια», πύλες και μετρήσεις. Ωστόσο, υπάρχει μια βασική διαφορά. Δεν επιτρέπεται η ανάδραση, δηλαδή η χρήση μιας εξόδου ως εισόδου στην ίδια πύλη: το κβαντικό κύκλωμα πάει μόνο «προς τα εμπρός».



Σχήμα 5: Qubit ως «καλώδιο»

2.4.2 Πύλες ενός Qubit

Βασικές πύλες του ενός Qubit είναι οι πύλες στροφής (γύρω από τους 3 άξονες στην σφαίρα του Bloch) και οι πύλες Pauli:



Σχήμα 6: Βασικές Πύλες

$$R_x(\theta) = \begin{pmatrix} \cos(\theta/2) & -i\sin(\theta/2) \\ -i\sin(\theta/2) & \cos(\theta/2) \end{pmatrix} \qquad X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

$$R_y(\theta) = \begin{pmatrix} \cos(\theta/2) & -\sin(\theta/2) \\ \sin(\theta/2) & \cos(\theta/2) \end{pmatrix} \qquad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$

$$R_z(\theta) = \begin{pmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{pmatrix} \qquad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

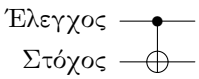
Επίσης γνωστή είναι η πύλη της ισοβαρούς υπέρθεσης, γνωστή και ως πύλη Hadamard :

Σχήμα 7: Πύλη Hadamard

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

2.4.3 Πύλες πολλών Qubit

Η βασικότερη πύλη πολλών Qubit είναι η πύλη CNOT (Controlled NOT - Ελεγχόμενη πύλη άρνησης) και δρά πάνω σε 2 Qubits. Αν το πρώτο Qubit γνωστό και ως Qubit ελέγχου είναι $|1\rangle$, τότε στο άλλο qubit (qubit στόχος) εφαρμόζεται η πύλη X, που αντιστοιχεί στον λογικό τελεστή της άρνησης (αφού $X|0\rangle = |1\rangle$, $X|1\rangle = |0\rangle$):



Σχήμα 8: CNOT

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

Από τον πίνακα φαίνεται πως όταν το πρώτο Qubit είναι μηδέν, τότε δεν αλλάζει το δεύτερο:

$$|00\rangle \longrightarrow |00\rangle\,, \qquad |01\rangle \longrightarrow |01\rangle$$

Ενώ αν το πρώτο Qubit είναι ένα, τότε αλλάζει το δεύτερο:

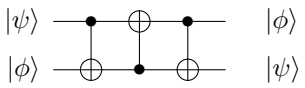
$$|10\rangle \longrightarrow |11\rangle\,, \qquad |11\rangle \longrightarrow |10\rangle$$

Με τις απλές πύλες στροφής και την ελεγχόμενη NOT, μπορούμε να φτιάξουμε οποιαδήποτε πύλη.

2.5 Παραδείγματα - Απλά κυκλώματα

2.5.1 Αντικατάσταση

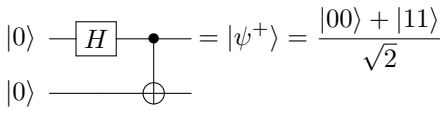
Απλό κύκλωμα που αντιμεταθέτει δύο qubits χρησιμοποιώντας πύλες CNOT. Αρκετά χρήσιμο μιας και τα Qubits δεν είναι πάντα συνδεδεμένα όλα μεταξύ τους.

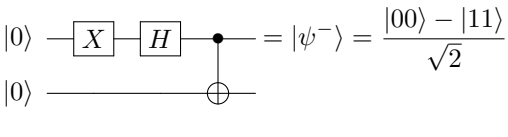


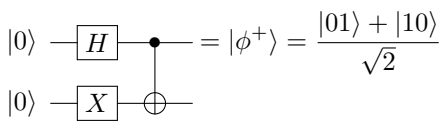
Σχήμα 9: SWAP

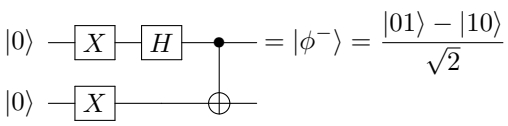
2.5.2 Καταστάσεις Bell

Κυκλώματα που κατασκευάζουν τις καταστάσεις μέγιστης συμπλοκής $|\psi^+\rangle\,, |\psi^-\rangle\,, |\phi^+\rangle\,, |\phi^-\rangle$, γνωστές και ως καταστάσεις Bell:



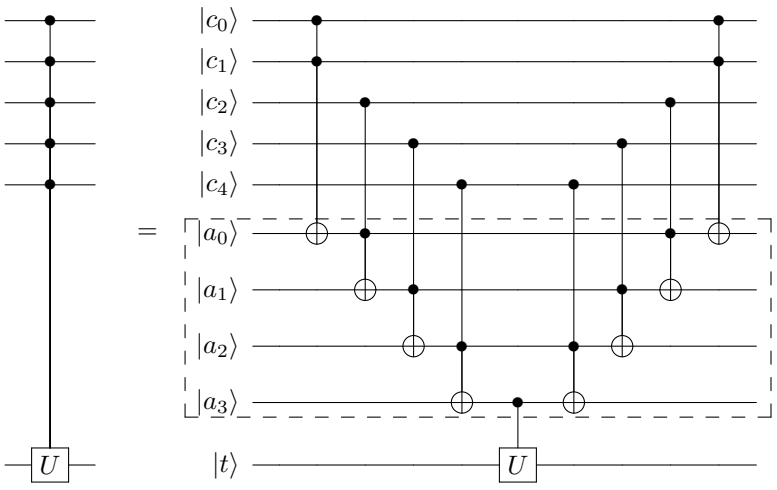






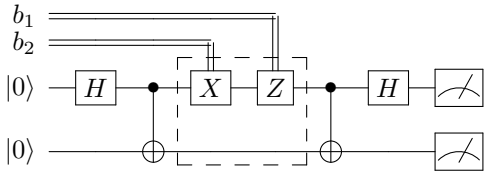
2.5.3 Πολλαπλά Ελεγχόμενη Πύλη

Κύκλωμα που κατασκευάζει πολλαπλά ελεγχόμενη πύλη χρησιμοποιώντας βοηθητικά Qubits.



2.5.4 Superdense Coding

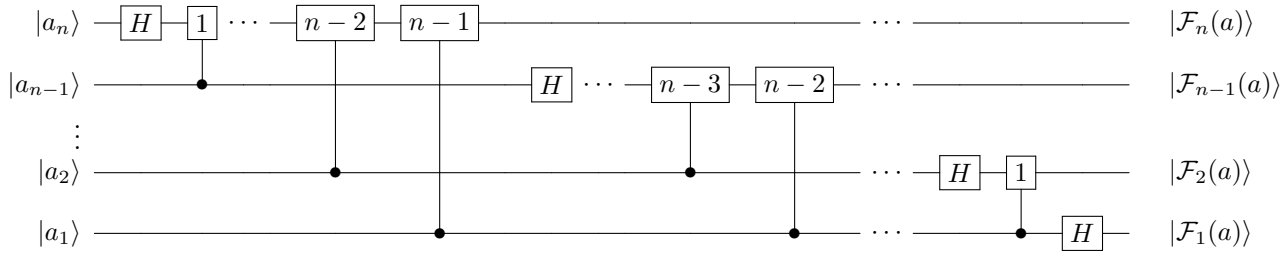
Απλό κύκλωμα [3] που παρουσιάζει την υπεροχή των κβαντικών υπολογιστών έναντι των κλασικών. Με τις πύλες X και Z μπορώ να μεταφέρω δύο bits πληροφορίας κωδικοποιώντας τα σε μόνο ένα Qubit.



2.5.5 Κβαντικός Μετασχηματισμός Fourier

Κύκλωμα που εφαρμόζει τον μετασχηματισμό **Fourier** [4] στην υπολογιστική βάση $|0\rangle, |1\rangle, \cdots, |n-1\rangle$:

$$Q\hat{F}T\,|x\rangle = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} e^{i\frac{2\pi xk}{n}}\,|k\rangle$$

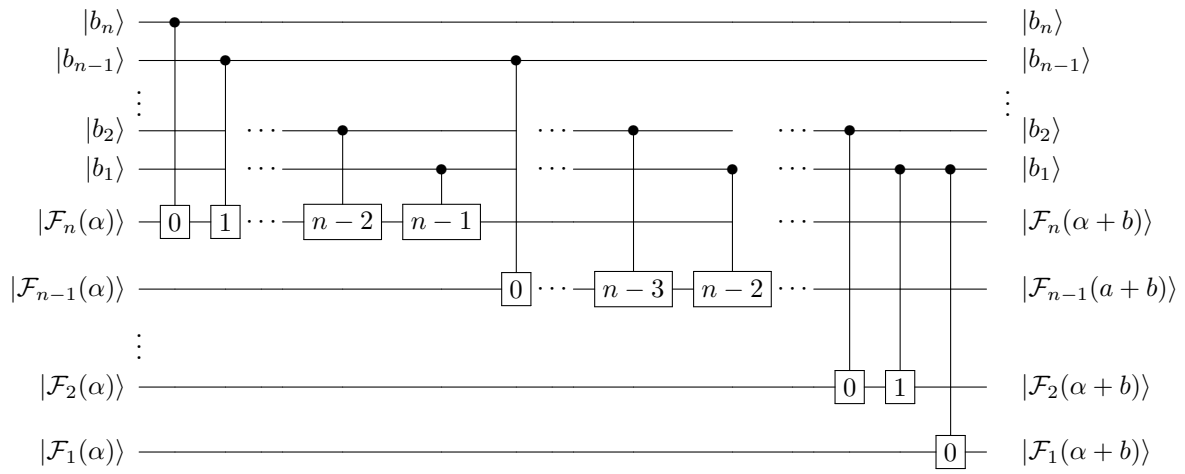


Όπου η πύλη $\boxed{k}$ αντιστοιχεί στον πίνακα στροφής $P(\frac{\pi}{2^k}) = \begin{pmatrix} 1 & 0 \\ 0 & e^{\frac{i\pi}{2^k}} \end{pmatrix}$.

Ο κβαντικός μετασχηματισμός Fourier για ένα διάνυσμα μήκους n μπορεί να εφαρμοστεί σε $\Theta(n^2)$ πράξεις, ενώ ο κλασικός σε $\Theta(n2^n)$. Είναι προφανές οτι ο κβαντικός είναι εκθετικά πιο γρήγορος. Συγκεκριμένα είναι τόσο γρήγορος που επιτρέπει να κάνουμε αποδοτικά πολλούς αλγορίθμους, ακόμα και την απλή άθροιση.

2.5.6 Κβαντικός Αθροιστής Fourier

Μπορούμε να κατασκευάσουμε **δυαδικό αθροιστή** [5], όπου η άθροιση συμβαίνει στο πεδίο των φάσεων. Συγκεκριμένα αν έχουμε 2 προσθετέους A και B στα Qubits $|A\rangle$, $|B\rangle$, τότε η άθροιση γίνεται με το κύκλωμα:

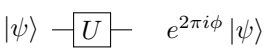


Σχήμα 10: Άθροιση στο πεδίο των φάσεων

Με έναν αντίστροφο μετασχηματισμό λαμβάνουμε το αποτέλεσμα.

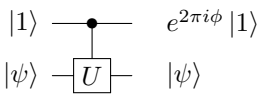
2.5.7 Μεταφορά Φάσης (Phase KickBack)

Κάθε κβαντική πύλη εφαρμόζεται σε μια κανονικοποιημένη κατάσταση και δημιουργεί μια επίσης κανονικοποιημένη κατάσταση. Αυτό σημαίνει οτι η εξίσωση ιδιοτιμών της θα είναι της μορφής: $\hat{U}|\psi\rangle = \lambda|\psi\rangle$, όπου: $\lambda^*\lambda\langle\psi|\psi\rangle = 1 \implies |\lambda|^2 = 1 \implies \boxed{\lambda = e^{2\pi i\phi}}$.



Σχήμα 11: Ολική φάση

Αυτή την ολική φάση $e^{2\pi i\phi}$ μπορούμε να την εξάγουμε από μια πύλη χρησιμοποιώντας την ελεγχόμενη εκδοχή της $(C\hat{U} = \hat{U} \oplus \hat{I})$.



Σχήμα 12: Μεταφορά Ολικής Φάσης

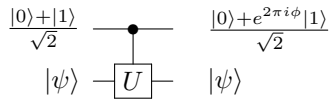
Πράγματι:

$$CU\left(|1\rangle|\psi\rangle\right) = \left(e^{2\pi i\phi}|1\rangle\right)|\psi\rangle$$

Παρ'όλα αυτά και πάλι είναι ολική φάση, οπότε δεν μετριέται με κανέναν τρόπο. Υπάρχει ένα τέχνασμα με το οποίο μπορούμε να κάνουμε την ολική φάση τοπική.

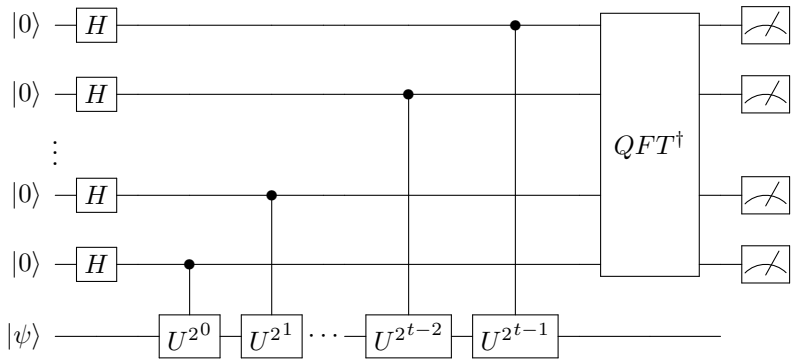
2.5.8 Κβαντική Εκτίμηση Φάσης - QPE

Συνδυάζοντας τις δύο προηγούμενες ιδέες μπορούμε να «διαβάσουμε» την φάση που εφαρμόζει μια πύλη σε μια κατάσταση. Η γωνία $0 \leq \phi \leq 1$ καλύπτει το διάστημα μοιρών $\{0, 360^o\}$. Στο δυαδικό σύστημα ένας τέτοιος αριθμός αναπαρίσταται με t ψηφία ως $0, \phi_0\phi_1...\phi_t$ και ακρίβεια της τάξης $\frac{1}{2^t}$. Ο κβαντικός μετασχηματισμός Fourier μετασχη-



Σχήμα 13: Μεταφορά Φάσης

ματίζει την κατάσταση $|x\rangle$ της υπολογιστικής βάσης στην κατάσταση: $Q\hat{F}T\,|x\rangle = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} e^{i\frac{2\pi xk}{n}} |k\rangle$. Εμείς θέλουμε το αντίστροφο: μια φάση ϕ να εμφανιστεί ως κατάσταση βάσης. Οπότε την εξάγουμε στο πεδίο των φάσεων με τις ελεγχόμενες πύλες U και με αντίστροφο μετασχηματισμό Fourier εμφανίζεται ως $|\phi_0\phi_1...\phi_t\rangle$. [2]



Σχήμα 14: Κβαντική Εκτίμηση Φάσης

2.5.9 Αλγόριθμος Εύρεσης Τάξης (Order Finding Algorithm)

Σε μια αριθμητική modulo N, ορίζουμε ως **τάξη** του αριθμού α τον μικρότερο θετικό αριθμό r (πλήν του μηδενός), για τον οποίον ισχύει:

$$\alpha^r \equiv 1 \pmod{N}$$

Η εύρεση της τάξης είναι δύσκολη διαδικασία. Ωστόσο, η συνάρτηση $f(r) = \alpha^r \pmod{N}$ είναι περιοδική και μάλιστα με περίοδο r. Οπότε αντί της τάξης, αρκεί να βρούμε την **περίοδο** αυτής της περιοδικής συνάρτησης. Ένα μαθηματικό εργαλείο που βρίσκει περιόδους είναι ο μετασχηματισμός Fourier. Εφαρμόζοντας λοιπόν τον κβαντικό αλγόριθμο εύρεσης φάσης στην πύλη $U_x\,|y\rangle = |xy \pmod{N}\rangle$ και με λίγη επεξεργασία του αποτελέσματος βρίσκουμε την τάξη r αρκετά γρήγορα.

2.5.10 Αλγόριθμος του Shor

Ο αλγόριθμος του Shor [6] χρησιμοποιείται για παραγοντοποίηση αριθμών. Δοθέντος ενός αριθμού N ζητάμε τους πρώτους διαιρέτες του. Ο αλγόριθμος είναι ο εξής:

1. Επιλέγουμε τυχαία έναν αριθμό α στο διάστημα $\{2...N-1\}$.
2. Εκτός κιάν επιλέξαμε τυχαία έναν από τους διαιρέτες, ισχύει: $gcd(\alpha, N) = 1$
3. Βρίσκουμε την τάξη r: $\alpha^r \equiv 1 \pmod{N}$
4. Έλεγχος α) Αν ο r είναι περιττός, πήγαινε στο βήμα 1

[2] Μια πύλη U λέμε οτι υψώνεται στην δύναμη k, όταν επαναλαβάνεται k φορές

5. Έλεγχος β) Αν $\alpha^{\frac{r}{2}} \equiv -1 \pmod{N}$, πήγαινε στο βήμα 1

6. Αν περάσαμε τους ελέγχους, βρήκαμε δύο διαιρέτες του N: $p, q = gcd(\alpha^{\frac{r}{2}} \pm 1, N)$

Η πιθανότητα να περάσουμε τους 2 ελέγχους είναι μεγαλύτερη του 50%. Το δύσκολο βήμα είναι η εύρεση της τάξης, όμως σε κβαντικό υπολογιστή γίνεται πάρα πολύ γρήγορα, επειδή ο κβαντικός μετασχηματισμός Fourier είναι **εκθετικά** πιο γρήγορος.

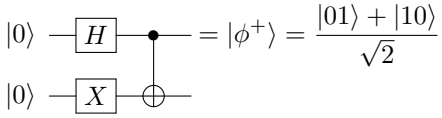
Ο αλγόριθμος του Shor θεωρείται πλέον κλασικός και είναι από τους πρώτους μεγάλους αλγορίθμους που δείξαν σαφές και εφαρμόσιμο πλεονέκτημα στους κβαντικούς υπολογιστές. Όλη μας η κρυπτογραφία εξαρτάται από την δυσκολία παραγοντοποίησης αριθμών.

2.5.11 Τελεστής Πυκνότητας

Τελεστής (ή Μήτρα) πυκνότητας ορίζεται η συμπαγής αναπαράσταση μιας κβαντικής κατάστασης ως άθροισμα τελεστών **προβολής**:

$$\rho = \sum_j p_j \left|\psi_j\right\rangle \left\langle\psi_j\right|$$

Για παράδειγμα, για την κατάσταση μέγιστης συμπλοκής Bell:



$$= |\phi^+\rangle = \frac{|01\rangle + |10\rangle}{\sqrt{2}}$$

έχουμε

$$\rho = \left(\frac{1}{\sqrt{2}}\begin{bmatrix}1\\0\\0\\1\end{bmatrix}\right)\left(\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 0 & 0 & 1\end{bmatrix}\right) = \frac{1}{2}\begin{bmatrix}1 & 0 & 0 & 1\\0 & 0 & 0 & 0\\0 & 0 & 0 & 0\\1 & 0 & 0 & 1\end{bmatrix}$$

Παρατηρούμε πως η κατάσταση δεν είναι καθαρή, αλλά μικτή, μιας και έχει μη-διαγώνια στοιχεία.

2.5.12 Αλγόριθμος του Grover

Ο αλγόριθμος του Grover [7] είναι ένας κβαντικός αλγόριθμος, ο οποίος χρησιμοποιείται για την αναζήτηση ενός μοναδικού στοιχείου του πεδίου ορισμού μιας συνάρτησης $f(x)$ το οποίο δίνει διαφορετική τιμή από όλα τα άλλα. Αν τα στοιχεία του πεδίου ορισμού δεν έχουν κάποια ταξινόμηση, τότε κλασικά θα θέλαμε χρόνο $O(N)$ για N στοιχεία· όμως με τον κβαντικό αλγόριθμο θέλουμε $\Omega(\sqrt{N})$. Ο αλγόριθμος αυτός δείχνει καθαρά το πλεονέκτημα που έχουν οι κβαντικοί υπολογιστές να ψάχνουν ταυτόχρονα πολλά δεδομένα σε έναν ευρύτερο χώρο.

Συγκεκριμένα αν έχουμε μια συνάρτηση $f : \{0, 1, .., N-1\} \to \{0, 1\}$, με μόνο μια τιμή ω για την οποία $f(x = \omega) = 1$. Σκοπός μας είναι να βρούμε το ω. Χρησιμοποιώντας τον τελεστή:

$$\begin{cases} U_\omega |x\rangle = -|x\rangle, & \text{για } x = \omega, \text{ δηλαδή } f(x) = 1 \\ U_\omega |x\rangle = |x\rangle, & \text{για } x \neq \omega, \text{ δηλαδή } f(x) = 0 \end{cases}$$

ο οποίος δρα πάνω σε $n = \lceil log_2 N \rceil$ Qubits. Μπορεί να γραφτεί και ως $U_\omega |x\rangle = (-1)^{f(x)} |x\rangle$

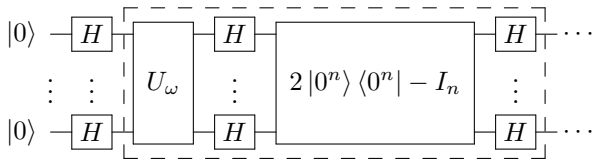
Algorithm 1 Grover's Algorithm

INPUT: An oracle U_ω
RESULT: ω

$|s\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle$
for iterations **do**
  Apply U_ω
  $U_s = 2|s\rangle\langle s| - I$
end for

Μετά από $r(N) \leq \lceil \frac{\pi}{4}\sqrt{N} \rceil$ επαναλήψεις έχουμε το ω.

Ο τελεστής $U_s = 2|s\rangle\langle s| - I$, ονομάζεται τελεστής διάχυσης και στην ουσία αυξάνει τις πιθανότητες της κατάστασης $|\omega\rangle$ και μειώνει τις υπόλοιπες, μετά από κάθε εφαρμογή του μαντείου U_ω.



Σχήμα 15: Αλγόριθμος του Grover

2.5.13 Αλγόριθμος Deutsch–Jozsa

Άλλος ένας αλγόριθμος που δείχνει την υπεροχή των κβαντικών υπολογιστών είναι ο αλγόριθμος του Deutsch–Jozsa [2]. Δοθείσης μιας συνάρτησης $f : \{0,1\}^n \to \{0,1\}$ η οποία είναι είτε σταθερή είτε ισόζυγη (δηλαδή επιστρέφει στο μισό πεδίο ορισμού 0 και στο άλλο μισό 1), μπορούμε να αποφανθούμε για το είδος της με μόνο μια εκτέλεση. Κατασκευάζουμε τον τελεστή U_f ο οποίος δρα πάνω σε έναν κβαντικό καταχωρητή x και ένα Qubit y και κάνει την απεικόνιση $|x\rangle|y\rangle \to |x\rangle|y \oplus f(x)\rangle$, όπου $\oplus$ η πρόσθεση modulo 2.

Ο αλγόριθμος είναι ο εξής:

Algorithm 2 Deutsch–Jozsa Algorithm

INPUT: An oracle U_f
RESULT: Type of f: constant / balanced

$|\psi_0\rangle = |0\rangle^{\otimes n} |1\rangle$ ▷ Prepare

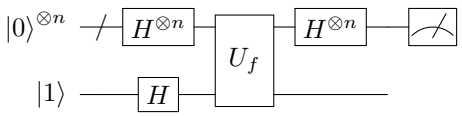
$|\psi_1\rangle = \frac{1}{\sqrt{2^{n+1}}} \sum_{x=0}^{2^n-1} |x\rangle (|0\rangle - |1\rangle)$ ▷ Broadcast H

$|\psi_2\rangle = \frac{1}{\sqrt{2^{n+1}}} \sum_{x=0}^{2^n-1} (-1)^{f(x)} |x\rangle (|0\rangle - |1\rangle)$ ▷ Apply U

$|\psi_3\rangle = \frac{1}{2^n} \sum_{y=0}^{2^n-1} \left[\sum_{x=0}^{2^n-1} (-1)^{f(x)} (-1)^{x*y} \right] |y\rangle$ ▷ Broadcast H to 1st register

$P(|0\rangle^{\otimes n}) = \left| \frac{1}{2^n} \sum_{x=0}^{2^n-1} (-1)^{f(x)} \right|^2 = 0, 1$ ▷ Measure

όπου $x * y = x_0 y_0 \oplus x_1 y_1 \oplus ... \oplus x_{n-1} y_{n-1}$ το άθροισμα των γινομένων κατά ψηφίο. Στην τελική μέτρηση αν ο πρώτος καταχωρητής είναι γεμάτος μηδενικά, τότε η f είναι σταθερή, αλλιώς είναι ισόζυγη.

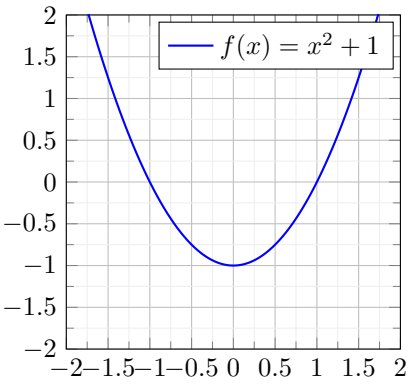


Σχήμα 16: Αλγόριθμος Deutsch–Jozsa

3 Βελτιστοποίηση

3.1 Σκοπός

Βελτιστοποίηση είναι ο κλάδος των μαθηματικών ο οποίος ασχολείται με την εύρεση της βέλτισης λύσης για ένα συγκεκριμένο πρόβλημα. Το πιο σύνηθες και σημαντικό πρόβλημα που εμφανίζεται στην πραγματικότητα είναι η **ελαχιστοποίηση** μιας συνάρτησης. Μερικές φορές η λύση στο πρόβλημα μπορεί να υπολογιστεί αναλυτικά. Για παράδειγμα, αν ζητάμε το ελάχιστο σημείο της συνάρτησης $f(x) = x^2 - 1$ μπορούμε μέσω της παραγώγου $f'(x) = 2x$ να υπολογίσουμε τα σημεία αλλαγής της κλίσης $f'(x) = 0 \implies x = 0$ και να υπολογίσουμε το ελάχιστο σημείο $f(x_0) = -1$ στην θέση $x_0 = 0$.



3.2 Βελτιστοποιητές

3.2.1 Εισαγωγή

Στην γενική περίπτωση όμως, ο αναλυτικός υπολογισμός της παραγώγου δεν είναι δυνατός· και ακόμα κι αν έχουμε έναν κλειστό τύπο για την παράγωγο, είναι δύσκολο να λύσουμε την εξίσωση $f'(x) = 0$. Χρειαζόμαστε λοιπόν έναν αλγόριθμο με τον οποίο να βρίσκουμε το ελάχιστο μιας συνάρτησης έχοντας μια προσέγγιση της **παραγώγου** σε κάθε σημείο. Ο αλγόριθμος αυτός είναι γνωστός ως Gradient Descent (Κάθοδος βασισμένη στην κλίση):

Algorithm 3 Gradient Descent Prototype

INPUT: A random value x, Number of iterations N
OUTPUT: Minimum of f(x)

```
for i in (1, N) do
    x ← x − f'(x)
end for
```

Επειδή η κλίση $f'(x)$ δείχνει προς την κατεύθυνση που η συνάρτηση μηδενίζεται, τότε προς τα εκεί η συνάρτηση φθίνει.

3.2.2 Gradient Descent

Βέβαια εμφανίζονται δύο προβλήματα: Πρώτον, στο τυχαίο σημείο που επιλέξαμε το x μπορεί η f να εμφανίζει τοπικό, και όχι ολικό, ελάχιστο. Δεύτερον, η κλίση μπορεί να είναι πολύ μεγάλη και να καθυστερεί η σύγκλιση. Για αυτό εισάγουμε την παράμετρο η που ονομάζεται **ρυθμός μάθησης** και συνήθως έχει μικρή τιμή $\eta \in (0.01, 0.1)$ κάνοντας μικρότερο το βήμα των επαναλήψεων. Οπότε ο

πλήρης αλγόριθμος γράφεται ως:

Algorithm 4 Gradient Descent

INPUT: A random value x, Number of iterations N
OUTPUT: Minimum of f(x)

$\eta = 0.01$
for i in $(1, N)$ **do**
    $x \leftarrow x - \eta * f'(x)$
end for

3.2.3 Βελτιστοποιητής Gradient Descent με ορμή

Ένας πολύ μικρός ρυθμός μάθησης βεβαιώνει πως δεν θα ξεφύγουμε από το ελάχιστο, αλλά καθυστερεί πολύ την σύγκλιση. Μια λύση για αυτό είναι να ξεκινάμε με μικρό ρυθμό μάθησης και να τον αυξάνουμε σταδιακά. Αυτή η μεταβλητή που ρυθμίζει τον ρυθμό μάθησης ονομάζεται **ορμή**. Ένας απλός αλγόριθμος βελτιστοποίησης με ορμή είναι ο εξής:

Algorithm 5 Gradient Descent with Momentum

INPUT: A random value x, Number of iterations N
OUTPUT: Minimum of f(x)

$\eta = 0.01$
$m = 0$
$\alpha = 0$
for i in $(1, N)$ **do**
    $\alpha \leftarrow m\alpha + \eta f'(x)$
    $x \leftarrow x - \eta * f'(x)$
end for

Οι υπολογισμοί δυσκολεύουν όταν οι συναρτήσεις που θέλουμε να ελαχιστοποιήσουμε είναι συναρτήσεις πολλών μεταβλητών. Η ιδέα όμως είναι η ίδια. Η μόνη διαφορά είναι οτι αντί της παραγώγου, χρησιμοποιούμε την κλίση $\nabla f(x)$ στους αλγορίθμους.

3.2.4 Βελτιστοποιητής Adam

Ίσως ο πιο ικανός βελτιστοποιητής για γενική χρήση είναι ο Adam [8]. Και αυτός είναι μια παραλλαγή του Gradient Descent με δύο ορμές.

Algorithm 6 Adam Optimizer

INPUT: A random initialization x, Number of iterations N
OUTPUT: Minimum of f(x)

$\eta = 0.01$
$\beta_1 = 0.9$
$\beta_2 = 0.99$
$\epsilon = 1e-08$
$a, b = 0$
for i in $(1, N)$ **do**
  Prepare
  $a \leftarrow \beta_1 a + (1-\beta_1)\nabla f(x)$
  $b \leftarrow \beta_2 b + (1-\beta_2)(\nabla f(x))^{\odot 2}$
  $\eta \leftarrow \eta \frac{\sqrt{(1-\beta_2)}}{(1-\beta_1)}$

  Update
  $x \leftarrow x - \eta \frac{a}{\sqrt{b}+\epsilon}$
end for

3.3 Στοχαστικότητα

Όσο καλός και να είναι ο βελτιστοποιητής, πάντα υπάρχει το ενδεχόμενο να εγκλωβιστούμε σε τοπικό ελάχιστο. Μια λύση σε αυτό το πρόβλημα είναι να χρησιμοποιούμε ένα άλλο διάνυσμα g αντί της κλίσης για την ανανέωση των μεταβλητών. Το διάνυσμα αυτό έχει έναν βαθμό αυθαιρεσίας, που μας επιτρέπει να απεγκλωβιστούμε από τοπικά ελάχιστα, αλλά στην στατιστική του εικόνα είναι ισοδύναμο με την κλίση· δηλαδή επιλέγεται τυχαία από μια κατανομή, αλλά η μέση τιμή του είναι η κλίση : $\mathbb{E}(g) = \nabla f$.

4 Νευρωνικά Δίκτυα

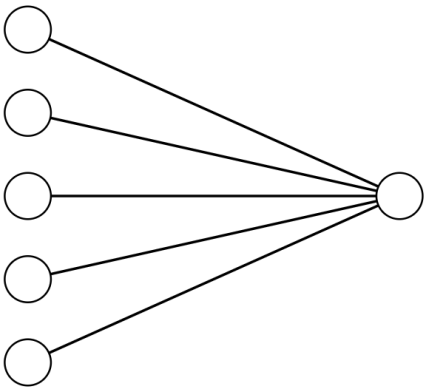
4.1 Νευρώνας

Ως **νευρώνας** ορίζεται μια μονάδα επεξεργασίας πληροφορίας. Κάθε νευρώνας έχει κάποιες εισόδους και μια έξοδο. Η πιο απλή και προφανής επιλογή εξόδου είναι ένας **γραμμικός συνδυασμός** των εισόδων. Αν συμβολίσουμε τις εισόδους ως x_i και τους συντελεστές του γραμμικού συνδυασμού ως w_i, τότε η έξοδος θα είναι:

$$y = \sum_{i=1}^{m} w_i x_i$$

Συνηθίζεται να χρησιμοποιείται και μια πόλωση, δηλαδή μια σταθερή τιμή μαζί με τον γραμμικό συνδυασμό. Στην γλώσσα της γραμμικής άλγεβρας:

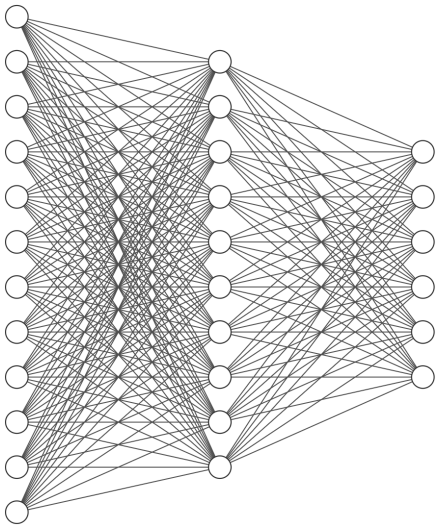
$$y = \begin{bmatrix} b & w_1 & \cdots & w_m \end{bmatrix} \cdot \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_m \end{bmatrix}$$



Σχήμα 17: Νευρώνας

4.2 Νευρωνικό Δίκτυο

Το επόμενο λογικό βήμα είναι να ενώσουμε πολλούς νευρώνες σε ένα επίπεδο.

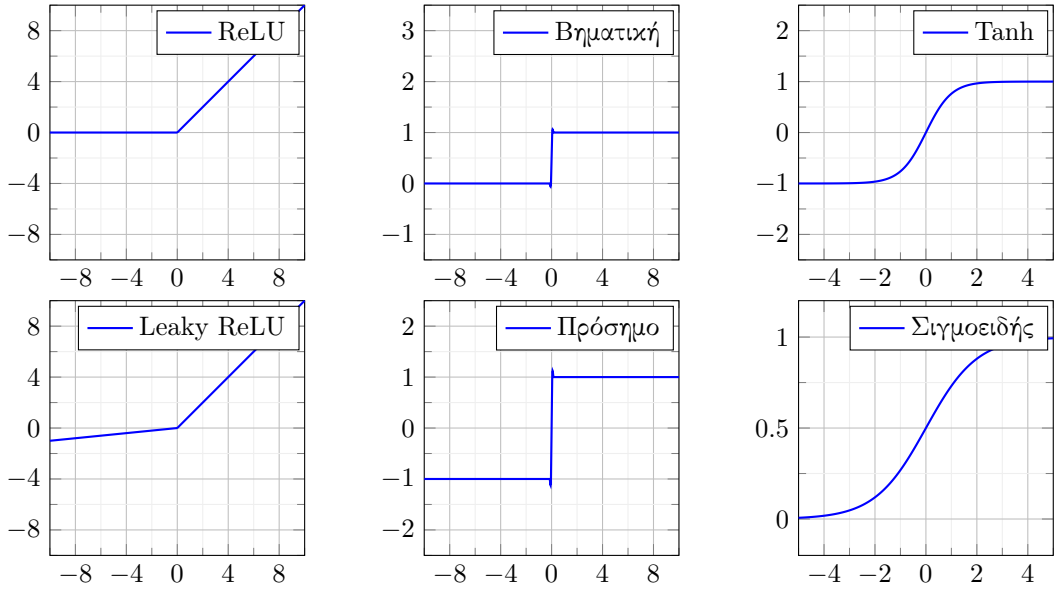


Σχήμα 18: Νευρωνικό Δίκτυο

$$\vec{y} = \begin{bmatrix} b_1 & w_{11} & \cdots & w_{1m} \\ b_2 & w_{21} & \cdots & w_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_n & w_{n1} & \cdots & w_{nm} \end{bmatrix} \cdot \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_m \end{bmatrix} = \begin{bmatrix} b_1 + \sum_{i=1}^{m} w_{1i} x_i \\ b_2 + \sum_{i=1}^{m} w_{2i} x_i \\ \vdots \\ b_n + \sum_{i=1}^{m} w_{ni} x_i \end{bmatrix}$$

Πάντα προτιμάμε τις γραμμικές πράξεις· ωστόσο, δυστυχώς δεν φτάνουν. Πράγματι, αν τοποθετήσουμε ένα δεύτερο επίπεδο γραμμικών συνδυασμών δίπλα στο πρώτο δεν θα έχουμε κερδίσει τίποτα· και αυτό γιατί γραμμικοί συνδυασμοί γραμμικών συνδυασμών είναι επίσης γραμμικοί συνδυασμοί. Άρα, θα έχουμε τις διπλάσιες μεταβλητές για το ίδιο αποτέλεσμα. Για κακή μας τύχη, πρέπει να εφαρμόζουμε στις εξόδους των νευρώνων μη-γραμμικές συναρτήσεις. Αυτή η διαδικασία κάνει «χρήσιμο» τον νευρώνα, γιάυτό και ονομάζεται «**συνάρτηση ενεργοποίησης**». Μερικά παραδείγματα τέτοιων

συναρτήσεων είναι τα:

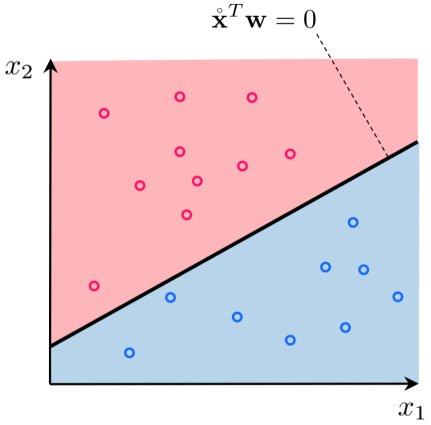


Συνήθως χρησιμοποιείται μια συνάρτηση με εύχρηστη παράγωγο και φραγμένο σύνολο τιμών, όπως η σιγμοειδής ή η υπερβολική εφαπτομένη.

4.3 Γραμμικός διαχωριστής - Perceptron

Ένα απλό, αλλά αρκετά ισχυρό, πρότυπο είναι το Perceptron. Αποτελείται από έναν νευρώνα ο οποίος δέχεται $\{x_i\}$ εισόδους σταθμισμένες με βάρη $\{w_i\}$ συν μια πόλωση και επιστρέφει το πρόσημο του υπολογισμού:

$$u = b + \sum_{i=1}^{m} w_i x_i$$
$$y = \text{sign}(u) = \pm 1$$

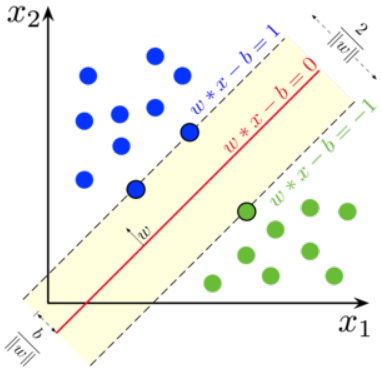


Σχήμα 19: Perceptron

Ο Perceptron χρησιμοποιείται για **ταξινόμηση** δεδομένων σε 2 κατηγορίες. Σκοπός του είναι απλά να πετύχει την σωστή ταξινόμηση και προϋπόθεση της σωστής λειτουργίας είναι τα δεδομένα να είναι γραμμικώς διαχωρίσιμα. Με χρήση πολλών Perceptron μπορούμε να κατηγοριοποιούμε δεδομένα σε πολλές κατηγορίες.

4.4 Μηχανή Διανυσμάτων Υποστήριξης (ΜΔΥ - SVM)

Το πρότυπο SVM είναι μια ειδικότερη περίπτωση του Perceptron. Σκοπός του δεν είναι απλώς να κατηγοριοποιήσει σωστά τα δεδομένα σε 2 κατηγορίες, αλλά να επιτύχει τον αυστηρότερο δυνατό διαχωρισμό των δεδομένων, μεγιστοποιώντας την ελάχιστη απόσταση των δειγμάτων από το επίπεδο διαχωρισμού.



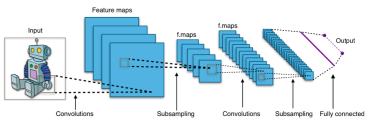
Σχήμα 20: SVM

4.5 Μη-γραμμικά διαχωρίσιμα δεδομένα

Στην γενικότερη περίπτωση τα δεδομένα μπορεί να μην είναι γραμμικά διαχωρίσιμα, οπότε αυξάνουμε τεχνητά τις διαστάσεις των δειγμάτων μέχρι να είναι γραμμικά διαχωρίσιμα. Το κόλπο αυτό ονομάζεται Kernel Trick. Με μια συνάρτηση $\phi(x)$ απεικονίζουμε τα δεδομένα σε έναν χώρο περισσοτέρων διαστάσεων, όπου είναι γραμμικά διαχωρίσιμα. Ωστόσο, δεν υπάρχει μια τέτοια απεικόνιση που να δουλεύει για όλα τα δείγματα, και κάθε φορά πρέπει να βρίσκουμε έναν νέο μετασχηματισμό.

4.6 Συνελικτικό Νευρωνικό Δίκτυο

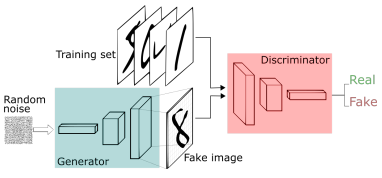
Ένα αρκετά ικανό πρότυπο νευρωνικών δικτύων για ανάλυση **εικόνων** είναι τα Συνελικτικά Νευρωνικά Δίκτυα [9]. Προτού τροφοδοτηθεί η εικόνα ως είσοδος στον νευρωνικό, την προ-επεξεργάζεται εξάγοντας αντιπροσωπευτικά δεδομένα. Τα χαρακτηριστικά αυτά υπόκεινται σε περαιτέρω επεξεργασία εως ότου τροφοδοτηθούν στο νευρωνικό δίκτυο προς κατηγοριοποίηση. Το νευρωνικό αυτό εκμεταλλεύεται την δισδιάστατη μορφή των δεδομένων (που συνήθως είναι εικόνες) και την τοπικότητα που υπάρχει στην είσοδο (γειτονικά pixels αναμένεται να είναι λίγο-πολύ όμοια) και μειώνει σημαντικά την διάσταση των δεδομένων εισόδου, βελτιώνοντας κατά πολύ την ταχύτητα της επεξεργασίας.



Σχήμα 21: CNN

4.7 Παραγωγικό Ανταγωνιστικό Δίκτυο

Ένα άλλο πρότυπο νευρωνικών δικτύων είναι τα Παραγωγικά Ανταγωνιστικά Δίκτυα [10]. Αυτού του είδους τα νευρωνικά αποτελούνται από δύο δομές: έναν παραγωγό και έναν ελεγκτή. Εκπαιδεύονται και τα δύο πάνω στο ίδιο σύνολο δεδομένων (πχ εικόνες από γάτες) με την διαφορά ότι το ένα εκπαιδεύεται στο να παράγει δεδομένα που μιμούνται τα πραγματικά και το άλλο ελέγχει αν τα δεδομένα που λαμβάνει είναι αυθεντικά ή τεχνητά.



Σχήμα 22: GAN

Σκοπός του παραγωγού είναι να εξαπατήσει τον ελεγκτή πως τα δικά του δεδομένα είναι αυθεντικά και σκοπός του ελεγκτή είναι να μην εξαπατηθεί από τα πλαστά δεδομένα του παραγωγού. Οπότε κατά κάποιο τρόπο η συνάρτηση την οποία χειρίζονται και οι δύο είναι κοινή:

$$\underset{D}{max}V(D) = \mathbb{E}_{x\sim p_{data}(x)}[logD(x)] + \mathbb{E}_{z\sim p_z(z)}[log(1 - D(G(z)))]$$

$$\underset{G}{min}V(G) = \mathbb{E}_{z\sim p_z(z)}[log(1 - D(G(z)))] \quad \text{ή} \quad \mathbb{E}_{z\sim p_z(z)} - [logD(G(z))]$$

Όπως φαίνεται ο ελεγκτής D προσπαθεί να διαχωρίζει σωστά τα πραγματικά δεδομένα που δέχεται από την κατανομή x (πρώτος όρος) και να εντοπίζει τα ψεύτικα δεδομένα που κατασκευάζει ο παραγωγός G (δεύτερος όρος). Αντίστοιχα, ο παραγωγός προσπαθεί να ελαχιστοποιήσει την πιθανότητα να θεωρήσει ο ελεγκτής δικά του δεδομένα ως μη-αυθεντικά. Είναι ένα από τα πιο εντυπωσιακά είδη νευρωνικών δικτύων, πετυχαίνει εκπληκτικά αληθοφανή αποτελέσματα και βασίζεται σε μία πολύ απλή και κομψή ιδέα.

4.8 Εκπαίδευση Νευρωνικών Δικτύων

Για να εκπαιδεύσουμε ένα νευρωνικό δίκτυο, δηλαδή για να υπολογίσουμε τις βέλτιστες παραμέτρους του με βάση την αντικειμενική συνάρτηση, χρειαζόμαστε την κλίση της σε κάθε σημείο. Ο υπολογισμός της κλίσης με αριθμητικές μεθόδους, πχ με τον ορισμό της παραγώγου, είναι πολύ χρονοβόρος. Ένας αναλυτικός υπολογισμός της κλίσης του εκάστοτε νευρωνικού είναι πρακτικά αδύνατος μιας και έχουμε πολλές μη γραμμικές συναρτήσεις εμφωλευμένες.

Υπάρχει ένας αποδοτικός αλγόριθμος υπολογισμού της κλίσης και ονομάζεται **Οπισθοδιάδοση** (BackPropagation). Με βάση αυτόν τον αλγόριθμο, υπολογίζουμε την έξοδο του νευρωνικού εκτελώντας το προς τα μπρος. Μετά διασχίζουμε το νευρωνικό προς τα πίσω, υπολογίζοντας το κάθε στοιχείο του ∇f την στιγμή που συναντάμε τον αντίστοιχο κόμβο.

5 Βελτιστοποίηση Κβαντικών Κυκλωμάτων

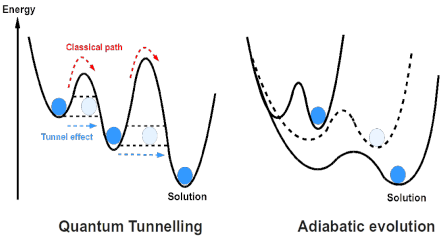
Οι κβαντικοί υπολογιστές είναι ιδιαίτερα καλοί στο να αναζητούν λύσεις σε **χώρους μεγάλων διαστάσεων**· για αυτό λοιπόν, μοιάζουν ιδανικοί για Νευρωνικά Δίκτυα και για βελτιστοποίηση εν γένει. Ήδη έχουν δείξει την σπουδαιότητά τους σε προβλήματα κβαντικής χημείας (πχ [11]).

5.1 Αμιγώς Κβαντική Βελτιστοποίηση

Μια μέθοδος είναι να εισαγάγουμε τα δεδομένα του προβλήματος στο κύκλωμα και να το αφήσουμε να κάνει όλες τις βελτιστοποιήσεις μόνο του (με μια εκτέλεση του κυκλώματος και χωρίς επίβλεψη).

5.1.1 Αδιαβατικός Υπολογισμός

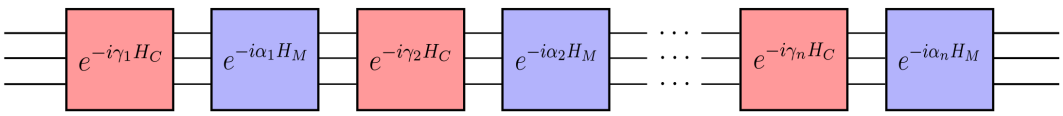
Ένας παρεμφερής τρόπος υπολογισμού (όχι με την μορφή κυκλώματος) είναι ο **αδιαβατικός υπολογισμός** [12]. Στην κβαντομηχανική ο τελεστής που μας δίνει την ενέργεια του συστήματος ονομάζεται Χαμιλτονιανή και συμβολίζεται με ένα $\hat{H}$. Το **κβαντικό αδιαβατικό θεώρημα** μας επιβεβαιώνει πως ένα φυσικό σύστημα προσαρμόζεται στο περιβάλλον του, όταν οδηγείται με μικρές μεταβολές. Πράγματι, αν καταφέρουμε να κατασκευάσουμε την χαμιλτονιανή $\hat{H} = (1-t)\hat{H}_i + t\hat{H}_f, t \in (0,1)$, τότε όσο ο χρόνος t μεγαλώνει η Χαμιλτονιανή μετακινείται από την κατάσταση $\hat{H}_i$ στην $\hat{H}_f$. Το μόνο που μένει είναι να κωδικοποιήσουμε την συνάρτηση ελαχιστοποίησης στον τελεστή $\hat{H}_f$ και να περιμένουμε το σύστημα να πάει από μόνο του στην κατάσταση αυτή. Η εταιρία D-Wave ασχολείται με αδιαβατικό κβαντικό υπολογισμό.



Σχήμα 23: Αδιαβατική Μέθοδος

5.1.2 QAOA

Ένας άλλος αμιγώς κβαντικός αλγόριθμος βελτιστοποίησης, που πραγματώνεται με κύκλωμα, είναι ο αλγόριθμος Quantum Approximate Optimization Algorithm - QAOA [13]:



Σχήμα 24: Ο αλγόριθμος QAOA

Algorithm 7 QAOA

INPUT: f(z) , β_i, γ_i
GOAL: $z \approx \underset{z\in\{0,1\}^n}{argmin} f(z)$

Compute $\hat{C}$ such that: $\hat{C}\,|z\rangle = f(z)\,|z\rangle$ ▷ Cost Hamiltonian
Compute: $B = \bigotimes_n X$ ▷ Mixer Hamiltonian
Construct: $U(C,\gamma) = e^{-i\gamma C}$ ▷ Cost Gate
Construct: $U(B,\beta) = e^{-i\beta B}$ ▷ Mixer Gate

$|\psi\rangle = |+\rangle^{\otimes n}$ ▷ Broadcast Hadamard
for i in {0,p} **do**
  $|\psi\rangle \leftarrow U(B,\beta_i)U(C,\gamma_i)\,|\psi\rangle$
end for

Measure $|\psi\rangle$

Αρχικά κατασκευάζουμε μια ισοβαρή υπέρθεση όλων πιθανών καταστάσεων και ύστερα χρησιμοποιούμε δύο πύλες: η μία υπολογίζει το κόστος που θέλουμε να ελαχιστοποιήσουμε και η άλλη ανακατεύει τις καταστάσεις. Με σωστή επιλογή των παραμέτρων β και γ πετυχαίνουμε καλή σύγκλιση.

Για προβλήματα συνδυαστικής βελτιστοποίησης ο αλγόριθμος QAOA είναι αρκετά αποδοτικός και αρκετά εύκολος στην υλοποίηση: πχ Max-Cut [14], Knapsack [15].

5.2 Υβριδικοί Αλγόριθμοι Βελτιστοποίησης

Σε αντίθεση με τις προηγούμενες μεθόδους, μπορούμε να εκπαιδεύουμε κλασικά ένα κβαντικό κύκλωμα.

5.2.1 Κβαντικό SVM (QSVM)

Κατ αναλογία με τις κλασικές Μηχανές Διανυσμάτων Υποστήριξης (SVM), μπορούμε να ορίσουμε Κβαντικές Μηχανές Διανυσμάτων Υποστήριξης (QSVM) [16]. Η ιδέα είναι ακριβώς η ίδια μόνο που εκμεταλλευόμαστε τον πολύ μεγαλύτερο χώρο στον οποίο έχουν πρόσβαση οι κβαντικές καταστάσεις.

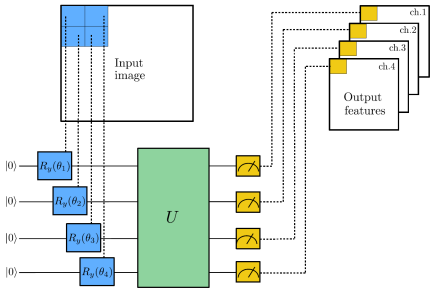
5.2.2 Κβαντικό Συνελικτικό Νευρωνικό Δίκτυο (QNN)

Με το ίδιο σκεπτικό μπορούμε να κατασκευάσουμε ένα Κβαντικό Συνελικτικό Νευρωνικό Δίκτυο (Quonvolutional NN) για κατηγοριοποίηση εικόνων [17]. Παράδειγμα

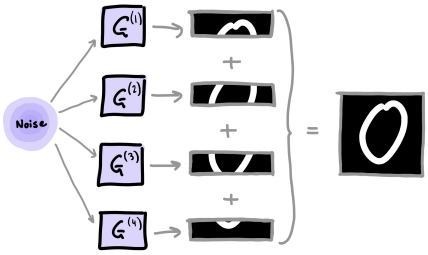
5.2.3 Κβαντικό GAN

Επίσης μπορούμε να κατασκευάσουμε Κβαντικά Παραγωγικά Ανταγωνιστικά Δίκτυα, με κβαντικό παραγωγό [18] και κλασικό ελεγκτή. Μάλιστα για καλύτερο έλεγχο, λιγότερο θόρυβο (και δυνατότητα

προσομοίωσης) χρησιμοποιούνται πολλοί μικροί παραγωγοί οι οποίοι κατασκευάζουν κομμάτια της εικόνας, τα οποία μετά ενώνουμε.



Σχήμα 25: QNN



Σχήμα 26: QGAN

6 Προσομοιωτής

6.1 Σχεδιαστικές Αρχές

Σε αντίθεση με τους γνωστούς προσομοιωτές (πχ. Qiskit [19], Cirq, PennyLane [20]) που έχουν φτιαχτεί για διδακτικούς σκοπούς και με έμφαση στην ευκολία χρήσης, αυτός ο προσομοιωτής σκοπό έχει την μεγάλη **ταχύτητα** εκτέλεσης· για αυτόν τον σκοπό, ιδανική είναι η γλώσσα προγραμματισμού C++, μιας και έχει την ταχύτητα της C, αλλά και πολλές έτοιμες και αποδοτικές δομές δεδομένων (πχ vector, map).

Σχεδιαστικές Αρχές του προσομοιωτή είναι:

- **Απλότητα**: Διαθέτει τις απλές κβαντικές πύλες και βασικές λειτουργίες, όπως: εκτέλεση πειράματος, υπολογισμός διανύσματος κατάστασης, υπολογισμός πιθανοτήτων, υπολογισμός αναμενόμενης τιμής τελεστή και είναι πολύ απλός στην χρήση του.
- **Συμβατότητα**: Δεν εξαρτάται από καμία βιβλιοθήκη, εκτός της βασικής βιβλιοθήκης της C++.
- **Ταχύτητα**: Όλες οι δομές δεδομένων και οι λειτουργίες έχουν κατασκευαστεί από το μηδέν, με κύριο σκοπό την ταχύτητα.
- **Ευελιξία**: Αξιοποιούνται οι ομαδοποιήσεις των δεδομένων σε κλάσεις που προσφέρει η C++, έτσι ώστε να οργανώνονται αυτόνομα σε μικρές ομάδες· αυτό επιτρέπει την αλλαγή της υλοποίησης πολλών λειτουργιών ή δομών δεδομένων, χωρίς να χρειαστεί τροποποίηση του υπόλοιπου κώδικα. Επίσης, όλα τα αρχεία του κώδικα κληρονομούν μερικές σταθερές παραμέτρους (όπως η ακρίβεια των δεκαδικών αριθμών) από το αρχείο "parameters.h", οπότε πολύ εύκολα μπορεί να γίνει εξαγωγή μιας προτιμητέας υλοποίησης με την αλλαγή μόνο μιας γραμμής.
- **Ανοιχτό λογισμικό**: Ο πηγαίος κώδικας είναι διαθέσιμος για όλους προς κάθε χρήση.

6.2 Μαθηματικά εργαλεία

Η υλοποίηση αυτή δεν χρησιμοποιεί καμία έτοιμη βιβλιοθήκη για μαθηματικά. Τα μαθηματικά εργαλεία που χρειάζονται για τα κβαντικά κυκλώματα είναι:

- Μιγαδικοί αριθμοί
- Τετραγωνικοί πίνακες και μάλιστα με διαστάσεις δυνάμεις του 2 (επιτρέπει σημαντικές βελτιστοποιήσεις στο δυαδικό σύστημα)
- Διανύσματα (ή πίνακες-στήλες)

- Πράξεις με πίνακες και διανύσματα. Συγκεκριμένα: Τανυστικό γινόμενο, Γινόμενο πίνακα-διανύσματος, εσωτερικό γινόμενο δύο διανυσμάτων, κατασκευή ανάστροφου-συζυγή πίνακα
- Εκτύπωση των πινάκων και των διανυσμάτων

6.2.1 Μιγαδικοί Αριθμοί

Για τους μιγαδικούς αριθμούς χρησιμοποιούμε τον τύπο **c_type** που ορίζεται στο αρχείο "*parameters.h*" και είναι είτε double είτε float, ανάλογα με την ακρίβεια που επιθυμούμε για τους υπολογισμούς μας. Επίσης, υπερφορτώνουμε τους απλούς τελεστές για να επιτρέπεται μίξη πραγματικών και μιγαδικών αριθμών (πχ comp a = 3.0*z;)

6.2.2 Τετραγωνικοί πίνακες

Επειδή οι πίνακες δεν είναι γνωστοί την ώρα της μεταγλώττισης (άρα δεν μπορούν να είναι στατικοί), πρέπει να δεσμεύεται η απαιτούμενη μνήμη την ώρα της εκτέλεσης. Αν πραγματώσουμε λοιπόν τους πίνακες ως λίστα από λίστες (πχ std::vector< std::vector<T> > Array) θα χρειάζεται να κάνουμε πολλές κλήσεις για **δεύσμευση μνήμης**. Αντί αυτού χρησιμοποιούμε έναν **μονοδιάστατο πίνακα** τον οποίο χειριζόμαστε σαν δισδιάστατο.

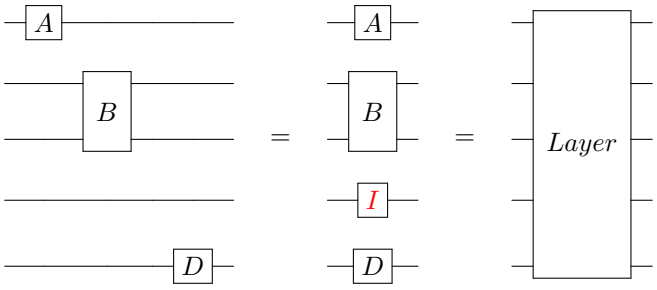
Συγκεκριμένα, ορίζεται η κλάση **SqMatrix** για τετραγωνικούς πίνακες ($M_{n\times n}$), η οποία αποτελείται από έναν μονοδιάστατο πίνακα n^2 στοιχείων και μιας τιμής n για να γνωρίζουμε τις διαστάσεις του τετραγωνικού πίνακα (χωρίς να χρειάζεται να τις υπολογίζουμε κάθε φορά). Πάνω σε αυτήν την υλοποίηση, χτίζεται μια απλή διεπαφή για τετραγωνικούς πίνακες με τις μεθόδους **get** και **set**, με τις οποίες διαβάζουμε από, και γράφουμε σε, πίνακες. Έτσι αποφεύγουμε τις πολλαπλές κλήσεις για δέσμευση μνήμης καθώς και αντικαθιστούμε την διπλή αναφορά (de-reference) με μονή, το οποίο βελτιώνει κι'άλλο την απόδοση των πινάκων, που είναι ο πυρήνας των υπολογισμών μας.

Δηλαδή, για έναν πίνακα διαστάσεων $A_{n\times n}$ έχουμε μια λίστα $K[n^2]$ στοιχείων και το στοιχείο $a[i][j]$ είναι στην ουσία το στοιχείο $K[i*n+j]$.

6.3 Πύλη - Επίπεδο πυλών

Η πύλη είναι στην ουσία ένας τετραγωνικός πίνακας, συν μια λίστα των Qubits πάνω στα οποία θα εφαρμοστεί. Φυσιολογικά λοιπόν, ορίζεται η κλάση Application η οποία περιέχει τον πίνακα, την προαναφερθείσα λίστα, καθώς και την παράγωγο της πύλης αν αυτή είναι παραμετρική (η παράγωγος χρειάζεται για την βελτιστοποίηση των κυκλωμάτων).

Πύλες που εφαρμόζονται σε διαφορετικά και μη-επάλληλα Qubits, μπορούν να εφαρμοστούν ταυτόχρονα σε ένα κύκλωμα. Συγκεκριμένα μπορούν να συντεθούν σε μια μεγάλη πύλη η οποία υπολογίζεται ως το τανυστικό γινόμενο όλων των επιμέρους πυλών:



Σχήμα 27: Επίπεδο ως πύλη

$$Layer = D \otimes I \otimes B \otimes A$$

Παρ'όλαυτα, το επίπεδο ως πύλη έχει $2^n \times 2^n = 2^{2n}$ στοιχεία, ενώ οι μικροί πίνακες (αν είναι του ενός Qubit) έχουν όλοι μαζί $2 \times 2 \times n = 4n$ στοιχεία. Οπότε ο μεγάλος πίνακας υπολογίζεται μόνο την ώρα της εκτέλεσης και όχι της εισαγωγής στο κύκλωμα.

6.4 Κύκλωμα

Η κλάση Circuit μοντελοποιεί κβαντικά κυκλώματα. Εφαρμογή μιας πύλης γίνεται με την μέθοδο app, ενώ οι πολύ γνωστές πύλες υπάρχουν ήδη και μπορούν να εφαρμοστούν αμέσως από τις αντίστοιχες μεθόδους: πχ RX(2, 0.672) για την πύλη Rotation X(0.672) στο Qubit 2.

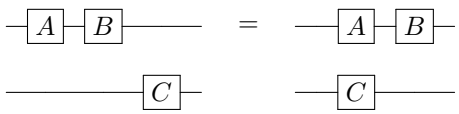
Ο χρήστης μπορεί να κατασκευάσει το κύκλωμα και κατόπιν:

- Να εκτελέσει **πείραμα** (μέτρηση) με την μέθοδο measure
- Να μετρήσει τις **πιθανότητες** της κάθε κατάστασης βάσης με την μέθοδο print_probabilities
- Να τυπώσει το **διάνυσμα της κατάστασης** με την μέθοδο print_state_vector
- Να μετρήσει την **αναμενόμενη τιμή** $\langle E \rangle$ για ένα μέγεθος με την μέθοδο exp_value
- Να υπολογίσει την **κλίση** $\nabla \langle E \rangle$ με βάση τις παραμέτρους του κυκλώματος για κάθε παραμετρική πύλη (με όποια σειρά της έδωσε ο χρήστης) με την μέθοδο exp_value_grad.

6.4.1 Συμπύκνωση σε Επίπεδα

Κάθε πύλη, που εφαρμόζεται σε ένα υποσύνολο των Qubits, επηρεάζει το όλο διάνυσμα καταστάσεων. Οπότε πρέπει να κατασκευάζεται κάθε φορά το όλο επίπεδο πυλών, συμπληρώνοντας με μοναδιαίους

πίνακες τις θέσεις, όπου δεν υπάρχει πραγματική πύλη. Αυτή η διαδικασία είναι πολύ χρονοβόρα, ειδικά όταν ο χρήστης δίνει πολλές μικρές πύλες. Επιλέγουμε λοιπόν να **συμπυκνώνουμε** όσο περισσότερο μπορούμε το κύκλωμα, έτσι ώστε να έχει τον μικρότερο δυνατό αριθμό επιπέδων.



Σχήμα 28: Συμπύκνωση επιπέδων

Αυτό που χρειαζόμαστε είναι να εφαρμόζουμε τις πύλες όσο πιο αριστερά γίνεται. Για να το πετύχουμε αυτό, πρέπει να γνωρίζουμε την πρώτη διαθέσιμη θέση για κάθε Qubit στο κύκλωμα. Οπότε, αν η πύλη χρησιμοποιεί πολλά Qubits, η εφαρμογή της θα γίνει στο πρώτο κοινό διαθέσιμο επίπεδο.

Ο αλγόριθμος που κάνει αυτή την συμπύκνωση (σε ψευδογλώσσα) είναι ο εξής:

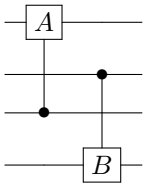
Algorithm 8 Gate application

INPUT:
available: index of the first available layer for every qubit
layers: Circuit Layout. Stores gate applications layer-by-layer
Application a: {Gate: Square Matrix, qubits: [int] }
RESULT:
Puts gate in circuit

```
layer_idx = 0

for qubit in a->qubits do
    layer_idx = max(layer_idx, available[qubit])        ▷ Find first common index
end for

if len(layers) < layer_idx + 1 then                     ▷ Out of Circuit
    layers.append([a])                                  ▷ New Layer of applications
else
    layers[layer_idx].append(a)                         ▷ Push this application in existing layer
end if

for qubit in a->qubits do                               ▷ Update available positions for these qubits
    available[qubit] = layer_idx + 1
end for
```

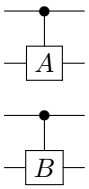
6.4.2 Μεταθέσεις

Έχοντας κατασκευάσει το κύκλωμα μένει να το εκτελέσουμε. Για να το εκτελέσουμε, ξεκινάμε με ένα αρχικοποιημένο διάνυσμα καταστάσεων $|0\rangle$ και εφαρμόζουμε τα επίπεδα-πύλες ένα-ένα μέχρι να φτάσουμε στο τέλος. Όμως υπάρχει περίπτωση οι πύλες που υπάρχουν σε ένα επίπεδο να μην εφαρμόζονται σε γειτονικά Qubits ή να εφαρμόζονται σε γειτονικά αλλά με αντίστροφη σειρά. Παραδείγματος χάριν:



Σχήμα 29: Ανακατεμένα Qubits

Αυτές οι δύο πύλες ανήκουν στο ίδιο επίπεδο (αφού τα σύνολα των Qubits τους είναι μεταξύ τους ξένα), παρόλο που δεν μπορούμε να τις σχεδιάσουμε την μια πάνω στην άλλη. Δεν μπορούμε να κατασκευάσουμε μια μεγάλη πύλη ως το τανυστικό τους γινόμενο, γιατί οι πύλες δεν είναι στην σειρά. Αν είχαμε αυτό όμως το επίπεδο, τώρα τα πράγματα θα ήταν πολύ εύκολα:



Σχήμα 30: Πύλες στην σειρά

Μια αφελής λύση θα ήταν να κατασκευάσουμε τον μεγάλο πίνακα που αντιστοιχεί στο επίπεδο, αντιστοιχόντας τα στοιχεία του με βάση την μετάθεση που χρειάζεται να κάνουμε. Όμως πιο αποδοτικό είναι να κατασκευάσουμε το επίπεδο με τις πύλες στην σειρά και απλώς να **μεταθέσουμε** τα Qubits στο διάνυσμα κατάστασης που περιέχει πολύ λιγότερα στοιχεία.

Χρειαζόμαστε λοιπόν έναν αλγόριθμο που να φτιάχνει την μετάθεση των Qubits για κάθε επίπεδο με βάση τις πύλες που συναντάμε και να μας δίνει και τον αριθμό των Qubits που αντιστοιχούν σε

πραγματικές πύλες, για να συμπληρώσουμε με μοναδιαίους πίνακες στα υπόλοιπα.

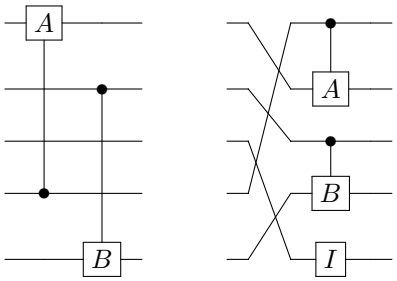
Algorithm 9 Make Permutation

INPUT:
layer: A layer of Gate applications
qubits: The number of qubits in the circuit
RESULT:
Permutation and number of qubits affected in this layer

```
allocated = [false]*qubits                          ▷ Keep track of allocated qubits
gates = len(layer)                                  ▷ Gates in this layer
last_used_qubit = 0

for gate in range(gates) do                         ▷ Iterate over Gates
    gate_size = len(layer[gate].qubits)             ▷ Size of particular Gate

    for qubit in range(gate_size) do                ▷ permutation[fake_qubit] = real_qubit
        permutation[qubit+last_used_qubit] = layer[gate].qubits[qubit]
        allocated[layer[gate].qubit] = true         ▷ allocated[real_qubit] = true
    end for
last_qubit_used += gate_size
end for
qubits_used = last_used_qubit                       ▷ Keep a copy to return

for iterator in range(qubits) do                    ▷ Fill permutation with unallocated qubits
    if allocated[iterator] == false then
        permutation[last_used_qubit++] = iterator
    end if
end for

Return (permutation, qubits_used)
```

6.4.3 Εφαρμογή σε μετάθεση

Έχοντας την μετάθεση των Qubits χρειάζεται να φτιάξουμε τον μεγάλο πίνακα του επιπέδου συμπληρώνοντας με μοναδιαίους πίνακες στα αχρησιμοποίητα Qubits:



Σχήμα 31: Πριν και μετά την μετάθεση

Το τελικό βήμα είναι:

1. Να φτιάξουμε το νέο διάνυσμα κατάστασης με βάση την μετάθεση
2. Να εφαρμόσουμε σε αυτό το επίπεδο-πύλη

3. Να αντιστρέψουμε την μετάθεση

Στην ουσία κατασκευάζουμε έναν μετασχηματισμό ομοιότητας $S = PDP^{-1}$, όπου ο πίνακας P^{-1} κάνει μετάθεση, ο P την αντίστροφη μετάθεση και ο D την εφαρμογή του επιπέδου πυλών:

Algorithm 10 Apply Layer Gate on permuted qubits

INPUT:
StateVector: The State-Vector
Permutation: Qubit permutation
qubits: The number of qubits in the circuit
Gate: Layer-Gate
RESULT:
The new State-Vector after this layer is applied

```
mapped_state_vector = StateVector
SV_size = 1 « qubits                                   ▷ State Vector size

for pos in range(len(StateVector)) do                  ▷ Iterate Basis States
    mapped_pos = 0                                     ▷ Find index of mapped State Vector
    for fake in range(qubits) do
        real = permutation[fake]
        real_bit = (pos » real) & 1
        mapped_pos |= (real_bit « fake)                ▷ replace fake qubit with real one
    end for
    mapped_state_vector[mapped_pos] = StateVector[pos]
end for

mapped_state_vector = gate * mapped_state_vector       ▷ Matrix Mul.

for pos in range(len(StateVector)) do                  ▷ Undo mapping
    mapped_pos = 0
    for fake in range(qubits) do
        real = permutation[fake]
        real_bit = (pos » real) & 1
        mapped_pos |= (real_bit « fake)
    end for
    StateVector[pos] = mapped_state_vector[mapped_pos] ▷ Reverse
end for
```

6.5 Μετρήσεις

Μετά την εκτέλεση της προσομοίωσης, έχοντας το διάνυσμα καταστάσεως υπολογισμένο, μένει να μετρήσουμε τα αποτελέσματα.

6.5.1 Διάνυσμα Κατάστασης

Μιας και πρόκειται για προσομοίωση, ο χρήστης μπορεί να δεί το ακριβές διάνυσμα κατάστασης, κάτι που δεν γίνεται με τα πραγματικά κυκλώματα. Το κύκλωμα προσφέρει αυτή τη λειτουργία με την μέθοδο print_state_vector.

6.5.2 Πιθανότητες

Επίσης ο χρήστης μπορεί να μετρήσει τις πιθανότητες της κάθε κατάστασης βάσης με την μέθοδο print_probabilities. Οι πιθανότητες προκύπτουν εύκολα από το διάνυσμα κατάστασης: παίρνοντας το μέτρο του τετραγώνου για κάθε κατάσταση.

6.5.3 Τυχαίο Πείραμα

Αν και ο προσομοιωτής δουλεύει με διανύσματα καταστάσεως, δίνεται η δυνατότητα να εκτελεστεί πείραμα σαν να ήταν πραγματικός κβαντικός υπολογιστής. Για να γίνει αυτό πρέπει με βάση της πιθανότητες της κάθε κατάστασης να φτιαχτεί μια κατανομή πιθανοτήτων. Με χρήση της βιβλιοθήκης <random> που προσφέρει η C++, μπορούμε να κατασκευάσουμε μια τέτοια κατανομή. Χρησιμοποιούμε την τυχαία γεννήτρια αριθμών **Mersenne Twister** για τυχαιότητα και την κλάση std::discrete_distribution η οποία μοντελοποιεί την διακριτή κατανομή μας. Τα αποτελέσματα μπαίνουν σε ένα λεξικό χρησιμοποιόντας την δομή δεδομένων std::map της C++.

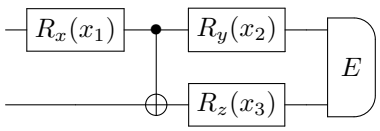
Ο χρήστης μπορεί να εκτελέσει ένα πείραμα με την μέθοδο measure(shots), δίνοντας το πλήθος των μετρήσεων που επιθυμεί να λάβουν μέρος.

6.5.4 Παρατηρούμενα μεγέθη

Πέρα από την απλή μέτρηση των καταστάσεων βάσης, ο χρήστης πολύ συχνά θέλει να μετρήσει την αναμενόμενη τιμή ενός μεγέθους με την αντίστοιχη Χαμιλτονιανή. Και αυτή η λειτουργία υποστηρίζεται από τον προσομοιωτή με την μέθοδο exp_value(Observable, qubits).

6.6 Διαφορικός Προγραμματισμός

Μέσα στις λειτουργίες του προσομοιωτή είναι να επιστρέφει την κλίση της αναμενόμενης τιμής ενός τελεστή με βάση τις παραμέτρους του κυκλώματος $\nabla\langle E\rangle$. Για παράδειγμα αν έχουμε το παρακάτω κύκλωμα:



Σχήμα 32: Παραμετρικό κύκλωμα

στο οποίο μετράμε την τιμή του τελεστή E πάνω στην υπολογιστική βάση, τότε σε βάθος πολλών επαναλήψεων η τιμή αυτή θα προσεγγίσει την αναμενόμενη τιμή $\langle E\rangle$, η οποία μπορεί να υπολογιστεί με αναλυτικούς, πλήν χρονοβόρους, υπολογισμούς.

6.6.1 Βελτιστοποίηση του κυκλώματος

Στο κομμάτι της βελτιστοποίησης ζητείται να ελαχιστοποιήσουμε έναν τέτοιο τελεστή και ο πιο γνωστός τρόπος είναι ο αλγόριθμος Gradient Descent (ή κάποια παραλαγή του):

$$x^{(t+1)} \leftarrow x^{(t)} - \eta * \nabla\langle E(x^{(t)})\rangle$$

Το υπολογιστικά δύσκολο βήμα είναι ο υπολογισμός της κλήσης $\nabla\langle E\rangle$ για ένα παραμετρικό κύκλωμα. Ο προφανής, αλλά αφελής, τρόπος είναι να χρησιμοποιήσουμε τον ορισμό της παραγώγου και να εκτελέσουμε το κύκλωμα $O(p)$ φορές, όπου p ο αριθμός των παραμέτρων:

$$\nabla\langle E\rangle_i = \frac{\partial\langle E\rangle}{\partial x_i} = \lim_{\epsilon\to 0}\frac{\langle E\rangle_{x_i+\epsilon} - \langle E\rangle}{\epsilon}$$

Η λύση αυτή δουλεύει, αλλά είναι αρκετά δαπανηρή, γιατί χρειάζεται να εκτελούμε το κύκλωμα ξεχωριστά κάθε φορά για όλες τις παραμέτρους.

6.6.2 Κανόνας Μετατόπισης Παραμέτρων (Parameter-Shift Rule)

Ένας αριθμητικός υπολογισμός της κλίσης που χρησιμοποιεί τον ορισμό της παραγώγου, δεν είναι ιδανικός, γιατί βασίζεται στην ακρίβεια που προσφέρει η τιμή ϵ. Αντί αυτού, συνήθως χρησιμοποιείται ο Κανόνας Μετατόπισης των Παραμέτρων [21], [22]· για πολλές απλές πύλες (πχ πύλες Pauli) ο γενικός κανόνας που ισχύει είναι:

$$\nabla_{\theta_i}\langle\hat{E}\rangle(\boldsymbol{\theta}) = \frac{1}{2}\left[\langle\hat{E}\rangle\left(\boldsymbol{\theta} + \frac{\pi}{2}\hat{\mathbf{e}}_i\right) - \langle\hat{E}\rangle\left(\boldsymbol{\theta} - \frac{\pi}{2}\hat{\mathbf{e}}_i\right)\right].$$

Αυτός ο τρόπος υπολογισμού της παραγώγου εγγυάται ακρίβεια.

6.6.3 Συζυγής Παραγώγιση (Adjoint Differentiation)

Όπως και στα κλασικά νευρωνικά δίκτυα, έτσι και εδώ, χρειαζόμαστε έναν αλγόριθμο για τον γρήγορο υπολογισμό της κλίσης· κάτι ανάλογο προς τον αλγόριθμο οπισθοδιάδοσης (BackPropagation).

Στα κβαντικά κυκλώματα, λοιπόν, υπάρχει ένας τέτοιος αλγόριθμος και ονομάζεται **Συζυγής Παραγώγιση** (Adjoint Differentiation) [23]. Στην πραγματικότητα το μόνο που κάνουμε στα κβαντικά κυκλώματα είναι να:

1. Ξεκινάμε από την αρχική κατάσταση $|0\rangle$
2. Εκτελούμε το κύκλωμα και λαμβάνουμε την κατάσταση $|\Psi\rangle = U_n U_{n-1}...U_0\,|0\rangle$
3. Μετράμε έναν τελεστή $\langle M\rangle = \langle\Psi|M|\Psi\rangle$

Ωστόσο, οι πύλες-τελεστές U_i είναι μοναδιαίοι, άρα ισχύει: $U^\dagger U\,|\phi\rangle = |\phi\rangle$

Ξαναγράφοντας την αναμενόμενη τιμή του τελεστή ως:

$$\langle M\rangle = \langle b|k\rangle = \langle\Psi|M|\Psi\rangle,$$

όπου

$$\langle b| = \langle\Psi|M = \langle 0|U_1^\dagger \dots U_n^\dagger M$$

$$|k\rangle = |\Psi\rangle = U_n U_{n-1}\dots U_1|0\rangle$$

παρατηρούμε πως τα διανύσματα $|b\rangle$ και $|k\rangle$ θα μπορούσαν να είχαν επιλεγεί λίγο διαφορετικά:

$$\langle b_n| = \langle 0|U_1^\dagger \dots U_n^\dagger M U_n$$

$$|k_n\rangle = U_{n-1}\dots U_1|0\rangle$$

Για να αλλάξουμε θέση στο σημείο διαχωρισμού της έκφρασης, απλώς αφαιρούμε τον τελεστή από το διάνυσμα $|k\rangle$ και το προσθέτουμε στο $\langle b|$:

$$\langle b_n| = \langle b|U_n$$

$$|k_n\rangle = U_n^\dagger|k\rangle$$

Για την παράγωγο τώρα:

$$\begin{aligned}\frac{\partial\langle M\rangle}{\partial\theta_i} &= \langle 0|U_1^\dagger \dots \frac{dU_i^\dagger}{d\theta_i} \dots M \dots U_i \dots U_1|0\rangle \\ &\quad + \langle 0|U_1^\dagger \dots U_i^\dagger \dots M \dots \frac{dU_i}{d\theta_i} \dots U_1|0\rangle \\ &= 2\cdot\Re\left(\langle 0|U_1^\dagger \dots U_i^\dagger \dots M \dots \frac{dU_i}{d\theta_i} \dots U_1|0\rangle\right)\end{aligned}$$

Ή χρησιμοποιώντας τα βοηθητικά μας διανύσματα:

$$\frac{\partial\langle M\rangle}{\partial\theta_i} = 2\Re\left(\langle b_i|\frac{dU_i}{d\theta_i}|k_i\rangle\right)$$

με

$$\langle b_i| = \langle 0|U_1^\dagger \dots U_n^\dagger M U_n \dots U_{i+1}$$

$$|k_i\rangle = U_{i-1}\dots U_1|0\rangle$$

Όταν ο τελεστής που μεταφέρεται είναι η παράγωγος, τότε θα συμβολίζουμε:

$$\langle \tilde{b}_i| = \langle b_i| \frac{dU_i}{d\theta_i}$$

και

$$\frac{\partial \langle M \rangle}{\partial \theta_i} = 2\Re\left(\langle \tilde{b}_i | k_i \rangle\right)$$

Το μόνο που χρειάζεται να κάνουμε είναι να υπολογίζουμε τα

$$|b_i\rangle = U_{i+1}^\dagger |b_{i+1}\rangle$$

$$|k_i\rangle = U_i^\dagger |k_{i+1}\rangle$$

αναδρομικά, ξεκινώντας από το τέλος του κυκλώματος (από το τελικό διάνυσμα κατάστασης μετά την εκτέλεση).

Ο πλήρης αλγόριθμος είναι ο εξής:

Algorithm 11 Adjoint Differentiation

INPUT: The output StateVector of a Circuit, Every gate U_i of the circuit (in order of insertion)
OUTPUT: Vector $\nabla\langle E \rangle$

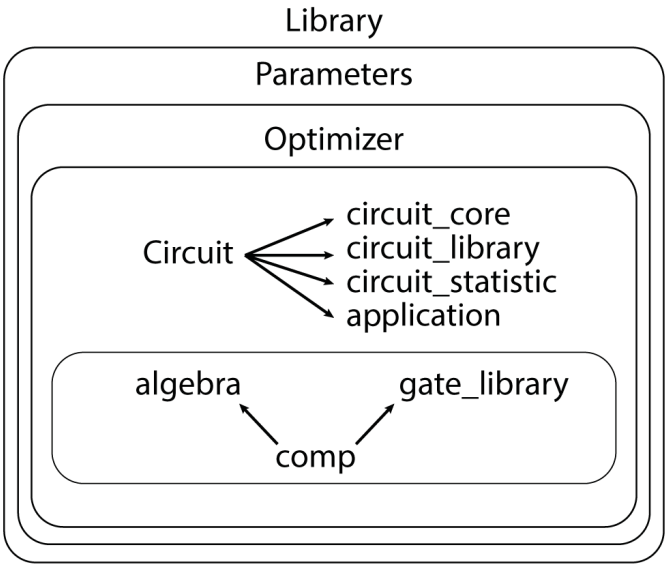
$|\lambda\rangle :=$ Circuit Output StateVector ▷ Execute Circuit
$|\phi\rangle := |\lambda\rangle$
$|\lambda\rangle \leftarrow \hat{E}\,|\lambda\rangle$
for $i \in \{P, ..., 1\}$ **do** ▷ Back-Propagate
  $|\phi\rangle \leftarrow \hat{U}_i^\dagger\,|\phi\rangle$
  $|\mu\rangle := |\phi\rangle$
  $|\mu\rangle \leftarrow (d\hat{U}_i/d\theta_i)\,|\mu\rangle$
  $\nabla\langle E\rangle_i = 2\Re\,\langle\lambda|\mu\rangle$
  if $i > 1$ **then**
    $|\lambda\rangle \leftarrow \hat{U}_i^\dagger\,|\lambda\rangle$
  end if
end for

Ο προσομοιωτής υποστηρίζει αυτή τη λειτουργία με την μέθοδο exp_value_grad, η οποία επιστρέφει την κλίση του κυκλώματος ως προς δοθέντα τελεστή. Για τον αλγόριθμο αυτόν το κύκλωμα χρησιμοποιεί ένα λεξικό με κλειδιά τον αύξων αριθμό κάθε πύλης και τιμές την θέση που η κάθε πύλη βρίσκεται μέσα στο κύκλωμα. Το λεξικό υλοποιείται με χρήση της βιβλιοθήκης map της C++.

6.6.4 Βελτιστοποιητές

Ο προσομοιωτής υποστηρίζει 3 βελτιστοποιητές προς το παρον. Τον απλό Gradient Descent, τον AdaGrad με μεταβλητό ρυθμό μάθησης και τον πολύ γνωστό Adam [8].

6.7 Σύνοψη κώδικα



Σχήμα 33: Σχέδιο Προσομοιωτή

6.7.1 library

Το αρχείο library.h καλεί όλα τα αρχεία του κώδικα και δημιουργεί την κβαντική μας βιβλιοθήκη.

6.7.2 parameters

Στο αρχείο parameters.h ορίζουμε γενικές παραμέτρους που αφορούν στην ολική λειτουργία του προσοιωτή, όπως ο μέγιστος αριθμός των Qubits, την ακρίβεια των πράξεών μας, την σχολαστικότητα στους ελέγχους κλπ.

6.7.3 comp

Το αρχείο comp.h είναι η βιβλιοθήκη μιγαδικών αριθμών. Ορίζει την δομή των μιγαδικών αριθμών, ορίζει βασικές μονομελείς και διμελείς πράξεις.

6.7.4 algebra

Το αρχείο algebra.h αναλαμβάνει την γραμμική άλγεβρα. Ορίζει τον δισδιάστατο τετραγωνικό πίνακα και πάνω σε αυτόν όλα τα μαθηματικά εργαλεία που χρειαζόμαστε: τανυστικό γινόμενο, εσωτερικό γινόμενο, αναμενόμενη τιμή τελεστή, ανάστροφο συζυγή πίνακα, εσωτερικό γινόμενο πίνακα με διάνυσμα.

6.7.5 gate_library

Το αρχείο gate_library.h περιέχει τους ορισμούς βασικών κβαντικών πυλών.

6.7.6 application

Το αρχείο application.h ορίζει την κλάση Application η οποία αντιστοιχεί στην εφαρμογή μιας κβαντικής πύλες πάνω σε ορισμένα Qubits.

6.7.7 circuit

Το αρχείο circuit.h ορίζει ένα κβαντικό κύκλωμα. Είναι η βασική κλάση την οποία χρησιμοποιεί ο χρήστης για την προσομοίωση.

6.7.8 circuit_core

Το αρχείο circuit_core.h περιέχει τους αλγορίθμους που χρειάζονται για τις λειτουργίες του κυκλώματος: εισαγωγή πύλης, μεταθέσεις, εφαρμογή επιπέδου πυλών, εκτέλεση κυκλώματος, αναμενόμενη τιμή, κλίση παραμέτρων ως προς αναμενόμενη τιμή.

6.7.9 circuit_library

Το αρχείο circuit_library.h περιέχει την βιβλιοθήκη κβαντικών πυλών που υποστηρίζει εκ κατασκευής ο προσομοιωτής χρησιμοποιώντας τους ορισμούς πινάκων που βρίσκονται στο αρχείο gate_library.h.

6.7.10 circuit_statistics

Το αρχείο circuit_statistics περιέχει εργαλεία μετρήσεων που χρησιμοποιεί ο χρήστης μετά την εκτέλεση του κυκλώματος.

6.7.11 optimizers

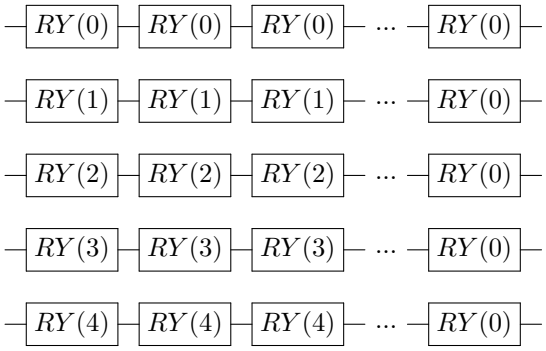
Το αρχείο optimizers.h ορίζει βελτιστοποιητές κβαντικών κυκλωμάτων.

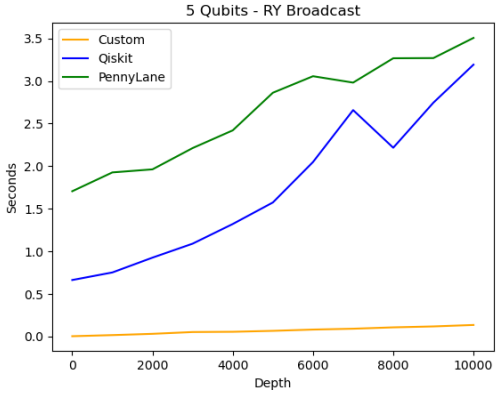
7 Μετρήσεις

Για τις μετρήσεις μας, συγκρίνουμε τον χρόνο 3 δύσκολων προσομοιώσεων στον προσομοιωτή μας, στην Qiskit της IBM (python) και στο lightning.qubit της PennyLane (C++).

7.1 5 Qubits - Πύλες στροφής RY

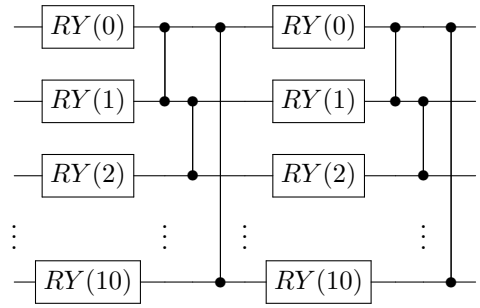
Ένα κύκλωμα 5 Qubit γεμάτο με πύλες RY και βάθος οριζόμενο από τον χρήστη. Επιλέγονται παραμετρικές πύλες RY του ενός Qubit, γιατί είναι οι πιο υπολογιστικά δαπανηρές.

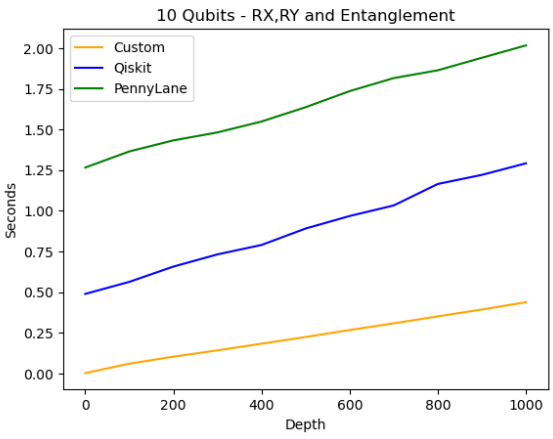




7.2 10 Qubits - Πύλες στροφής RY, RX και συμπλοκή

Ένα κύκλωμα 10 Qubit με παραμετρικά επίπεδα RY και επίπεδα συμπλοκής CZ. Είναι ένα πρότυπο νευρωνικού δικτύου αρκετά δύσκολο στην προσομοίωσή του (μιας και έχει μεγάλη συμπλοκή).

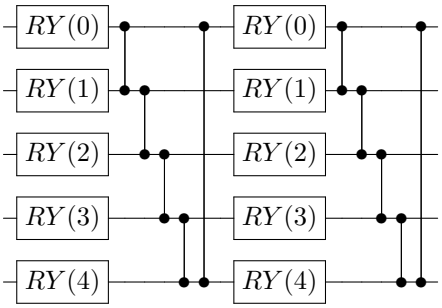


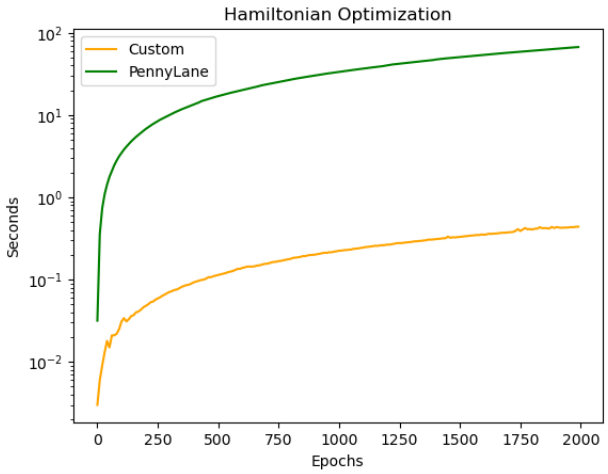


7.3 5 Qubits - Βελτιστοποίηση Χαμιλτονιανής

Ένα κύκλωμα σαν αυτό της δεύτερης δοκιμής με 5 Qubits, αλλά αυτή την φορά το βελτιστοποιούμε με βάση τον χαμιλτονιανό τελεστή του ατόμου H_2. Σκοπός μας είναι το κύκλωμα να παραγάγει μια κατάσταση $|\psi\rangle$, τέτοια ώστε η αναμενόμενη τιμή $\langle\psi| H_2 |\psi\rangle$ να είναι ελάχιστη. Δημιουργούμε τεχνητά

μη-γραμμικό μετασχηματισμό, συμπλέκοντας τα 4 Qubits που μας ενδιαφέρουν με ένα ακόμα Qubit.





8 Συμπεράσματα

Ο προσομοιωτής μας είναι αρκετά πιο γρήγορος από τους εμπορικούς, όπως ήταν αναμενόμενο (μόνο και μόνο λόγω γλώσσας). Παρατηρούμε, ωστόσο, οτι στο κομμάτι της βελτιστοποίησης ο προσομοιωτής μας είναι 3 τάξεις μεγέθους πιο γρήγορος από το C++ backend της PennyLane· αυτό μας επιτρέπει να βελτιστοποιούμε πολύ μεγαλύτερα κυκλώματα στον ίδιο χρόνο.

9 Προεκτάσεις του έργου

Ο προσομοιωτής είναι θεωρητικά πλήρης, εφόσον μπορεί να προσομοιώσει κάθε δυνατό κβαντικό κύκλωμα. Ωστόσο, μπορούμε να προσθέσουμε λειτουργίες καθώς και να βελτιστοποιήσουμε την απόδοσή του.

9.1 Βιβλιοθήκες

Στην βάση του προσομοιωτή μπορούν να προστεθούν βιβλιοθήκες ειδικού σκοπού.

9.1.1 Βελτιστοποίηση

Μια προσθήκη είναι περισσότεροι βελτιστοποιητές:

1. Βελτιστοποιητής άλγεβρας Lie [24] [25]. Οι αλγόριθμοι καθόδου με κλίση (Gradient Descent) μπορούν να γίνουν κατευθείαν σε μια ομάδα Lie, αντί σε έναν ευκλείδιο χώρο. Η μέθοδος αυτή υπόσχεται σύγκλιση στο ολικό ελάχιστο, αλλά χρησιμοποιεί εκθετικά περισσότερες παραμέτρους.

2. Βελτιστοποιητής Κβαντικής Φυσικής Κλίσης (Quantum Natural Gradient Optimizer) [26]. Βασίζεται στον υπολογισμό της διαγωνίου προσέγγισης του μετρικού τανυστή Fubini-Study.

3. Ο βελτιστοποιητής RotoSelect. Είναι μια ειδική περίπτωση βελτιστοποιητή που στοχεύει σε πύλες στροφής. [27]

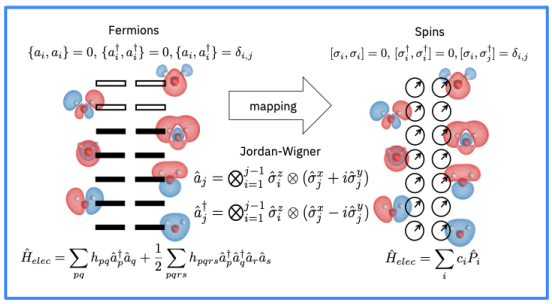
9.1.2 Κβαντική Χημεία

Οι κβαντικοί υπολογιστές είναι ιδιαίτερα αποδοτικοί σε προβλήματα κβαντικής χημείας. Θα μπορούσε να προστεθεί μια βιβλιοθήκη κβαντικής χημείας σαν αυτή της PennyLane. Στις βασικές της λειτουργίες συμπεριλαμβάνονται:

1. Κατασκευή Χαμιλτονιανής για μόρια

2. Απεικονίσεις τελεστών δημιουργίας / καταστροφής σε τελεστές spin (πχ Jordan-Wigner)

3. Η μέθοδος Hartree-Fock



Σχήμα 34: Απεικόνιση Jordan-Wigner

9.1.3 Κβαντική Βελτιστοποίηση

Θα ήταν χρήσιμη μια βιβλιοθήκη κβαντικής βελτιστοποίησης. Βασικά συστατικά που χρειαζόμαστε είναι:

1. Αλγόριθμοι, όπως ο QAOA [13].
2. Βιβλιοθήκες Νευρωνικών Δικτύων (πχ [28], [29], [30]).
3. Στοχαστικότητα στον υπολογισμό της κλίσης.

9.2 Απόδοση - Παραλληλία

Πάντα μπορούμε να βελτιστοποιήσουμε ένα πρόγραμμα. Μια εύκολη βελτίωση θα ήταν να χρησιμοποιήσουμε παραλληλία, είτε με πολλούς πυρήνες (OpenMP) είτε με χρήση επιταχυντή - κάρτας γραφικών (CUDA). Επίσης, θα μπορούσαμε να βελτιώσουμε τους βασικούς αλγορίθμους εκτέλεσης του κυκλώματος, όπως τις μεταθέσεις ή την κατασκευή του τανυστικού γινομένου, χρησιμοποιώντας αραιούς πίνακες για επίπεδα με λίγες πύλες.

10 Κώδικας

10.1 library

```
#include "parameters.h"

// MATHEMATICS CORE
#include "comp.h"
#include "algebra.h"


// ARRAY AND GATE HANDLERS
#include "gate_library.h"


// CIRCUIT
#include "circuit.h"
#include "circuit_core.h"
#include "circuit_statistics.h"
#include "circuit_library.h"


#include "optimizers.h"
```

10.2 parameters

```
// BUILDING PARAMETERS. CHANGE HERE AND ALL HEADER
// FILES WILL INHERIT THEM
#ifndef __PARAMETERS__
#define __PARAMETERS__ 420


// REMOVE THIS LINE TO EXCLUDE ALL CHECKS
#define __SAFE_LIBRARY__ 420
#define MAX_QUBITS 10


// COMPLEX NUMBER TYPE
typedef double c_type;
//typedef float c_type; // No difference really... keep double



#endif
```

10.3 comp

```
#ifndef __MY_COMPLEX_LIBRARY
#define __MY_COMPLEX_LIBRARY 420
#include <iostream> // cout
#include <math.h>         // sqrt
#include <stdlib.h>         // abs

#include "parameters.h"

// CORE LIBRARY FOR COMPLEX NUMBERS

// Constants
const c_type pi = 3.141592653589793238;


// CUSTOM COMPLEX NUMBER
typedef struct complex__{
        c_type real;
        c_type imag;
} comp;


// Complex Operations
```

```
inline comp         c                       (c_type real, c_type imag);
inline comp         conj                 (comp a);
inline comp         mul_conj          (comp a, comp b);
inline c_type         distance          (comp a, comp b);
inline void           print_comp          (comp a);


// Complex Number Constructor
inline comp c(c_type _real, c_type _imag) {
        return (comp) {_real, _imag};
}


inline comp conj (comp a) {
        return (comp) {a.real, -a.imag};
}

// COMPLEX CONJUGATE MULTIPLICATION a*b
inline comp mul_conj(comp a, comp b) {
        return (comp) {a.real*b.real + a.imag*b.imag,
                a.real*b.imag - a.imag*b.real};
}


// print as: a + ib
inline void print_comp (comp a) {
        if(a.real == 0.0 and a.imag == 0.0) {
                std::cout << "0  ";
                return;
        }

        if(a.real != 0.0)
                std::cout << a.real;

        if(a.imag != 0.0) {
                if(a.imag > 0.0) {
                        std::cout << "+i";
                        if(a.imag != 1.0)
                                std::cout <<  a.imag;
                }
                else {
                        std::cout << "-i";
                        if(a.imag != -1.0)
                                std::cout << -a.imag;
                }
        }

        std::cout << "  ";
}


// euclidean distance of two complex numbers
inline c_type distance (comp a, comp b) {
        // manhattan distance
        return abs(a.real - b.real) + abs(a.imag - b.imag);
}




// Complex Arithmetics

// COMPLEX OPERATOR COMPLEX

inline comp operator + (comp c1, comp c2) {
    return comp {c1.real + c2.real, c1.imag + c2.imag};
}
```

```
inline comp operator - (comp c1, comp c2) {
    return comp {c1.real - c2.real, c1.imag - c2.imag};
}

inline comp operator * (comp c1, comp c2) {
    return (comp) {c1.real*c2.real - c1.imag*c2.imag,
                c1.real*c2.imag + c1.imag*c2.real};
}

inline comp operator / (comp c1, comp c2) {
        c_type den = c2.real * c2.real + c2.imag * c2.imag;

    return (comp) {
            (c1.real * c2.real + c1.imag * c2.imag) / den,
                (c1.imag * c2.real - c1.real * c2.imag) / den
        };
}



// COMPLEX OPERATOR REAL & REAL OPERATOR COMPLEX

// position independant
inline comp operator + (comp c, c_type r) { return (comp) {r + c.real, c.imag}; }
inline comp operator + (c_type r, comp c) { return (comp) {r + c.real, c.imag}; }
inline comp operator * (comp c, c_type r) { return (comp) {r * c.real, r * c.imag}; }
inline comp operator * (c_type r, comp c) { return (comp) {r * c.real, r * c.imag}; }

// position dependant
inline comp operator - (comp c, c_type r) { return (comp) {c.real - r, c.imag}; }
inline comp operator - (c_type r, comp c) { return (comp) {r - c.real, -c.imag}; }
inline comp operator / (comp c, c_type r) { return (comp) {c.real / r, c.imag / r}; }
inline comp operator / (c_type r, comp c) {
        c_type den = c.real * c.real + c.imag * c.imag;
    return (comp) { (r * r + c.imag * c.imag) / den, (- r * c.imag) / den };
}



#endif
```

10.4 algebra

```
#ifndef __ALGEBRA__
#define __ALGEBRA__ 420

#include <vector>
#include "parameters.h"
#include "comp.h"



// Define Custom 2D Square Matrix
// Implement with 1D-flattened array
class SqMatrix {
        private:
                unsigned int _dims;         // rows-columns
                std::vector<comp> _elems;
        public:
                inline SqMatrix(unsigned int d);
                inline SqMatrix(std::vector<std::vector <comp>> e);

                inline comp get(unsigned int r, unsigned int c) const;
                inline void set(unsigned int r, unsigned int c, comp e);
                inline unsigned int dims() const { return _dims; }
};
```

```
// Init
inline SqMatrix::SqMatrix(unsigned int d) {
        _dims  = d;
        _elems = std::vector<comp> ( d*d );
}

// Init from vector of vector
inline SqMatrix::SqMatrix(std::vector<std::vector <comp>> e) {
        unsigned int d = e.size();

        #ifdef __SAFE_LIBRARY__
                for(unsigned int i = 0; i < d; ++i){
                        if(e[i].size() != d) {
                                std::cout << "Matrix Constructor Error: "
                                        << "2D vector is not square\n";
                                exit(1);
                        }
                }
        #endif

        _elems = std::vector<comp> (d*d);
        _dims  = d;
        for(unsigned int i = 0; i < d; ++i)
                for(unsigned int j = 0; j < d; ++j)
                        _elems[i * d + j] = e[i][j];
}


// GET HANDLER
inline comp SqMatrix::get(unsigned int r, unsigned int c) const {
        // you can ignore this. vector will crash anyway
        #ifdef __SAFE_LIBRARY__
                if(r > _dims-1){
                        std::cout << "SqMatrix Get_Element Error:"
                                "Element row index " << r << " out of range\n";
                        exit(1);
                }
                if(c > _dims-1){
                        std::cout << "SqMatrix Get_Element Error:"
                                "Element column index " << c << " out of range\n";
                        exit(1);
                }
        #endif

        return _elems[r * _dims + c];
}

// SET HANDLER
inline void SqMatrix::set(unsigned int r, unsigned int c, comp e){
        // you can ignore this. vector will crash anyway
        #ifdef __SAFE_LIBRARY__
                if(r > _dims-1){
                        std::cout << "SqMatrix Set_Element Error:"
                                "Element row index " << r << " out of range\n";
                        exit(1);
                }
                if(c > _dims-1){
                        std::cout << "SqMatrix Set_Element Error:"
                                "Element column index " << c << " out of range\n";
                        exit(1);
                }
        #endif

        _elems[r * _dims + c] = e;
}



// LINEAR ALGEBRA
```

```
// Tensor product for square matrices
SqMatrix square_tensor_prod(
        const SqMatrix &arr1,
        const SqMatrix &arr2)
{
        // tensor product dimension
        unsigned int dim = arr1.dims() * arr2.dims();
        unsigned int p = arr2.dims(); // power of two

        // find which power of two is p
        unsigned int power;
        for(power = 0; p >> power != 1; ++power) ;

        // =========== FILL ARRAY =========== //
        SqMatrix elements (dim);


        for(unsigned int i = 0; i < dim; ++i) {
                for(unsigned int j = 0; j < dim; ++j)
                        // C_ij = a[i/p][j/p] * b[i%p][j%p];
                        // dimensions are power of two!!! optimize / and %
                        elements.set(i, j,
                                arr1.get(i>>power,j>>power) * arr2.get(i&(p-1),j&(p-1)) );
        }

        return elements;
}


// return an I array with dimensions dim x dim
SqMatrix Identity_matrix(unsigned int dim) {

        SqMatrix result(dim);

        for(unsigned int i = 0; i < dim; ++i)
                for(unsigned int j = 0; j < dim; ++j)
                        result.set(i, j, c((i == j) ? 1.0 : 0.0, 0.0) );

        return result;
}




/*
        / S 0 0 0 \ / psi1 \
        | 0 S 0 0 | | psi2 |
        | 0 0 S 0 | | .... |
        \ 0 0 0 S / \ psin /
*/
// Given S and Statevector only!!!
std::vector<comp> Sparse_Array_dot_state(
        const SqMatrix &array,
        const std::vector<comp> &state)
{
        // how many S's in the diagonal?
        unsigned int Id_dim = state.size() / array.dims();


#ifdef __SAFE_LIBRARY__
        // =========== CHECK DIMENSIONS! =========== //
        if(state.size() != Id_dim * array.dims()) {
                std::cout << "Array Dot State Error: "
                        "Array dimensions(" << array.dims() <<
                        ") and State dimensions(" << state.size() <<
                        ") don't match\n";
                        exit(1);
        }
```

```
#endif


        unsigned int dim = state.size();
        std::vector<comp> res(dim);
        unsigned int s_dim = array.dims();


        // for every sub-matrix S in the diagonal
        for(unsigned int s = 0; s < Id_dim; ++s) {
                // Iterate S
                for(unsigned int i = 0; i < s_dim; ++i) {
                        comp c_i = c(0.0, 0.0);
                        // c_i = SUM a_ik b_k
                        for(unsigned int k = 0; k < s_dim; ++k)
                                // offset for statevector --> some S arrays
                                c_i = c_i + array.get(i,k) * state[s_dim * s + k];

                        // result row in statevector
                        res[s_dim * s + i] = c_i;
                }
        }
        return res;
}



bool is_hermitian( const SqMatrix &Obs ) {
        // ============ CHECK ELEMENTS ============ //
        // Hermitian means: a_ij = a_ji*
        for(unsigned int i = 0; i < Obs.dims(); ++i) {
                for(unsigned int j = 0; j < Obs.dims(); ++j)
                        // allow a small error --> their distance
                        if( distance(Obs.get(i,j) , conj(Obs.get(j,i))) > 0.0001 )
                                return false;
        }

        return true;
}



// MUST BE DONE WITH SPARSE ARRAYS, TOO!
// Returns:   <A> = <Psi|A|Psi>
c_type expectation_value (
        const SqMatrix &Obs,
        const std::vector<comp> &state )
{
        #ifdef __SAFE_LIBRARY__
                // =========== ARRAY CHECKS =========== //
                if(!is_hermitian(Obs)) {
                        std::cout << "Expectation Value Error: "
                                "Observable matrix is not Hermitian!\n";
                        exit(1);
                }
        #endif

        // compute
        unsigned int dim = state.size();
        comp tmp_res = c(0.0, 0.0);
        for(unsigned int i = 0; i < dim; ++i) {
                comp tmp = c(0.0, 0.0);
                for(unsigned int j = 0; j < dim; ++j)
                        tmp = tmp + Obs.get(i,j) * state[j];
                tmp_res = tmp_res + mul_conj(state[i], tmp);
        }

        // check imaginary part. Should be zero or very small
        #ifdef __SAFE_LIBRARY__
```

```
                // this complex should be real.
                if( abs(tmp_res.imag) > 0.0001 ) {
                        std::cout << "Expectation Value Error: "
                                "Result has big imaginary part\n";
                        exit(1);
                }
        #endif


        return tmp_res.real;
}



// Return transpose conjugate of a matrix
inline SqMatrix dagger(const SqMatrix & m) {
        unsigned int d = m.dims();

        SqMatrix res(d);

        // b_ij = a_ji*
        // RE-ORDER FOR CACHE EFFICIENCY
        for(unsigned int i = 0; i < d; ++i)
                for(unsigned int j = 0; j < d; ++j)
                        res.set(i,j, conj(m.get(j,i)) );

        return res;
}



// inner product of two complex vectors
comp inner_product(
        const std::vector<comp> &v1,
        const std::vector<comp> &v2)
{
        unsigned int s1 = v1.size();
        #ifdef __SAFE_LIBRARY__
                if(s1 != v2.size()) {
                        std::cout << "Inner Product Error: "
                                "vector sizes are different\n";
                        exit(1);
                }
        #endif

        comp res = c(0.0, 0.0);
        for(unsigned int i = 0; i < s1; ++i)
                res = res + mul_conj(v1[i], v2[i]);

        return res;
}


////////////////////////////////

// DEBUG INFO
void print_vector(std::vector<c_type> vec) {
        std::cout << "Printing Vector\n";
        for(unsigned int i = 0; i < vec.size(); ++i)
                std::cout << vec[i] << " ";
        std::cout << "\n";
}


void print_matrix(const SqMatrix &m) {
        int r = m.dims();

        for(int i = 0; i < r; ++i) {
                for(int j = 0; j < r; ++j)
                        print_comp( m.get(i, j) );
```

```
                std::cout << "\n";
        }
}

////////////////////////////////
// SqMatrix operators



// Make tensor product as operator %
inline SqMatrix operator % (
        const SqMatrix &arr1,
        const SqMatrix &arr2)
{
        return square_tensor_prod(arr1, arr2);
}


// Add arrays
inline SqMatrix operator + (
        const SqMatrix &arr1,
        const SqMatrix &arr2)
{
        #ifdef __SAFE_LIBRARY__
                if(arr1.dims() != arr2.dims()) {
                        std::cout << "SqMatrix operator + error: "
                                "Different sizes\n";
                                exit(1);
                }
        #endif

        unsigned int n = arr1.dims();

        SqMatrix res(n);

        for(unsigned int i = 0; i < n; ++i)
                for(unsigned int j = 0; j < n; ++j)
                        res.set(i,j, arr1.get(i,j) + arr2.get(i,j));

        return res;
}


// Add arrays
inline SqMatrix operator - (
        const SqMatrix &arr1,
        const SqMatrix &arr2)
{
        #ifdef __SAFE_LIBRARY__
                if(arr1.dims() != arr2.dims()) {
                        std::cout << "SqMatrix operator + error: "
                                "Different sizes\n";
                                exit(1);
                }
        #endif

        unsigned int n = arr1.dims();

        SqMatrix res(n);

        for(unsigned int i = 0; i < n; ++i)
                for(unsigned int j = 0; j < n; ++j)
                        res.set(i,j, arr1.get(i,j) - arr2.get(i,j));

        return res;
}
```

```
// Make number * array operator
inline SqMatrix operator * (
        const c_type number,
        const SqMatrix & arr)
{
        unsigned int n = arr.dims();

        SqMatrix res(n);

        for(unsigned int i = 0; i < n; ++i)
                for(unsigned int j = 0; j < n; ++j)
                        res.set(i,j, arr.get(i,j) * number);

        return res;
}
#endif
```

10.5 gate_library

```
#ifndef __ARRAY_LIBRARY__
#define __ARRAY_LIBRARY__ 420

#include <vector>
#include <math.h>
#include "parameters.h"
#include "comp.h"
#include "algebra.h"


// contains arrays and
// matrices built on top of those arrays


// Standard Gate ARRAYS!!!!


// NON- PARAMETRIC GATES

// ===== ONE QUBIT OPEARATIONS =====
// 1 0
// 0 1
const std::vector<std::vector<comp>> arr_I {
        { c(1.0,0.0), c(0.0,0.0) },
        { c(0.0,0.0), c(1.0,0.0) }
};

// 1                1  1
// sqrt(2)        1 -1
const c_type sqrt1_2 = sqrt(0.5);
std::vector<std::vector<comp>> arr_H {
        { c(sqrt1_2,0.0), c( sqrt1_2,0.0)},
        { c(sqrt1_2,0.0), c(-sqrt1_2,0.0)}
};

// Pauli Gates
// 0 1
// 1 0
const std::vector<std::vector<comp>> arr_X {
        { c(0.0,0.0), c(1.0,0.0) },
        { c(1.0,0.0), c(0.0,0.0) }
};

// 0 -i
// i  0
const std::vector<std::vector<comp>> arr_Y {
        { c(0.0,0.0), c(0.0,-1.0) },
        { c(0.0,1.0), c(0.0, 0.0) }
};

// 1  0
```

```
// 0 -1
const std::vector<std::vector<comp>> arr_Z {
        { c(1.0,0.0), c( 0.0,0.0) },
        { c(0.0,0.0), c(-1.0,0.0) }
};

// 1  0                        1 0
// 0  e^{i pi/2} =        0 i
const std::vector<std::vector<comp>> arr_S {
        { c(1.0,0.0), c(0.0,0.0) },
        { c(0.0,0.0), c(0.0,1.0) }
};

// 1  0
// 0  e^{i pi/4}
const std::vector<std::vector<comp>> arr_T {
        { c(1.0,0.0), c(0.0,0.0) },
        { c(0.0,0.0), c(cos(pi/4),sin(pi/4)) }
};



// ===== TWO QUBIT OPERATIONS =====
// 1 0 0 0
// 0 1 0 0
// 0 0 0 1
// 0 0 1 0
const std::vector<std::vector<comp>> arr_CNOT {
        { c(1.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0) },
        { c(0.0,0.0), c(1.0,0.0), c(0.0,0.0), c(0.0,0.0) },
        { c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(1.0,0.0) },
        { c(0.0,0.0), c(0.0,0.0), c(1.0,0.0), c(0.0,0.0) }
};

// 1 0 0  0
// 0 1 0  0
// 0 0 0 -i
// 0 0 i  0
const std::vector<std::vector<comp>> arr_CY {
        { c(1.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0) },
        { c(0.0,0.0), c(1.0,0.0), c(0.0,0.0), c(0.0,0.0) },
        { c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,-1.0) },
        { c(0.0,0.0), c(0.0,0.0), c(0.0,1.0), c(0.0,0.0) }
};

// 1 0 0  0
// 0 1 0  0
// 0 0 1  0
// 0 0 0 -1
const std::vector<std::vector<comp>> arr_CZ {
        { c(1.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0) },
        { c(0.0,0.0), c(1.0,0.0), c(0.0,0.0), c(0.0,0.0) },
        { c(0.0,0.0), c(0.0,0.0), c(1.0,0.0), c(0.0,0.0) },
        { c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(-1.0,0.0) }
};

// 1 0 0 0
// 0 0 1 0
// 0 1 0 0
// 0 0 0 1
const std::vector<std::vector<comp>> arr_SWAP {
        { c(1.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0) },
        { c(0.0,0.0), c(0.0,0.0), c(1.0,0.0), c(0.0,0.0) },
        { c(0.0,0.0), c(1.0,0.0), c(0.0,0.0), c(0.0,0.0) },
        { c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(1.0,0.0) }
};

// ===== THREE QUBIT OPERATIONS =====
const std::vector<std::vector<comp>> arr_TOFFOLI {
        { c(1.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0) },
```

```
        { c(0.0,0.0), c(1.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0) },
        { c(0.0,0.0), c(0.0,0.0), c(1.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0) },
        { c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(1.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0) },
        { c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(1.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0) },
        { c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(1.0,0.0), c(0.0,0.0), c(0.0,0.0) },
        { c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(1.0,0.0) },
        { c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(1.0,0.0), c(0.0,0.0) }
};



/*
 * PARAMETRIC OPERATIONS
 */


// ===== ONE QUBIT OPEARATIONS =====

// Rotation gates
std::vector<std::vector<comp>> arr_RX (c_type angle) {
        return {
                { c(cos(angle/2.0),0.0), c(0.0,-sin(angle/2.0)) },
                { c(0.0,-sin(angle/2.0)), c(cos(angle/2.0),0.0) }
        };
}


std::vector<std::vector<comp>> arr_RY (c_type angle) {
        return {
                { c(cos(angle/2.0),0.0), c(-sin(angle/2.0),0.0) },
                { c(sin(angle/2.0),0.0), c(cos(angle/2.0),0.0) }
        };
}


std::vector<std::vector<comp>> arr_RZ (c_type angle) {
        return {
                { c(cos(angle/2.0),-sin(angle/2.0)), c(0.0,0.0) },
                { c(0.0,0.0), c(cos(angle/2.0),sin(angle/2.0)) }
        };
}


// 1 0
// 0 e^{i phi}
std::vector<std::vector<comp>> arr_P (c_type angle) {
        return {
                { c(1.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(cos(angle),sin(angle)) }
        };
}


// ===== TWO QUBIT OPEARATIONS =====
std::vector<std::vector<comp>> arr_CRX (c_type angle) {
        c_type phi = angle/2.0;

        return {
                { c(1.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(1.0,0.0), c(0.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(0.0,0.0), c(cos(phi),0.0), c(0.0,-sin(phi)) },
                { c(0.0,0.0), c(0.0,0.0), c(0.0,-sin(phi)), c(cos(phi),0.0) }
        };
}

std::vector<std::vector<comp>> arr_CRY (c_type angle) {
        c_type phi = angle/2.0;

        return {
```

```
                { c(1.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(1.0,0.0), c(0.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(0.0,0.0), c(cos(phi),0.0), c(-sin(phi),0.0) },
                { c(0.0,0.0), c(0.0,0.0), c(sin(phi),0.0), c(cos(phi),0.0) }
        };
}

std::vector<std::vector<comp>> arr_CRZ (c_type angle) {
        c_type phi = angle/2.0;

        return {
                { c(1.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(1.0,0.0), c(0.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(0.0,0.0), c(cos(phi),-sin(phi)), c(0.0,0.0) },
                { c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(cos(phi), sin(phi)) }
        };
}


std::vector<std::vector<comp>> arr_CP (c_type angle) {
        return {
                { c(1.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(1.0,0.0), c(0.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(0.0,0.0), c(1.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(0.0,0.0), c(0.0,0.0),c(cos(angle),sin(angle)) }
        };
}



/*
 * GRADIENTS OF PARAMETRIC GATES
 */
std::vector<std::vector<comp>> arr_RX_dif (c_type angle) {
        return {
                { c(-sin(angle/2.0)/2.0,0.0), c(0.0,-cos(angle/2.0)/2.0) },
                { c(0.0,-cos(angle/2.0)/2.0), c(-sin(angle/2.0)/2.0,0.0) }
        };
}



std::vector<std::vector<comp>> arr_RY_dif (c_type angle) {
        return {
                { c(-sin(angle/2.0)/2.0,0.0), c(-cos(angle/2.0)/2.0,0.0) },
                { c( cos(angle/2.0)/2.0,0.0), c(-sin(angle/2.0)/2.0,0.0) }
        };
}



std::vector<std::vector<comp>> arr_RZ_dif (c_type angle) {
        return {
                { c(-sin(angle/2.0)/2.0,-cos(angle/2.0)/2.0), c(0.0,0.0) },
                { c(0.0,0.0), c(-sin(angle/2.0)/2.0,cos(angle/2.0)/2.0) }
        };
}


// 1 0
// 0 e^{i phi}
std::vector<std::vector<comp>> arr_P_dif (c_type angle) {
        return {
                { c(1.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(-sin(angle),cos(angle)) }
        };
}


// ===== TWO QUBIT OPEARATIONS =====
```

```
std::vector<std::vector<comp>> arr_CRX_dif (c_type angle) {
        c_type phi = angle/2.0;

        return {
                { c(1.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(1.0,0.0), c(0.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(0.0,0.0), c(-sin(phi)/2.0,0.0), c(0.0,-cos(phi)/2.0) },
                { c(0.0,0.0), c(0.0,0.0), c(0.0,-cos(phi)/2.0), c(-sin(phi)/2.0,0.0) }
        };
}

std::vector<std::vector<comp>> arr_CRY_dif (c_type angle) {
        c_type phi = angle/2.0;

        return {
                { c(1.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(1.0,0.0), c(0.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(0.0,0.0), c(-sin(phi)/2.0,0.0), c(-cos(phi)/2.0,0.0) },
                { c(0.0,0.0), c(0.0,0.0), c(cos(phi)/2.0,0.0), c(-sin(phi)/2.0,0.0) }
        };
}

std::vector<std::vector<comp>> arr_CRZ_dif (c_type angle) {
        c_type phi = angle/2.0;

        return {
                { c(1.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(1.0,0.0), c(0.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(0.0,0.0), c(-sin(phi)/2.0,-cos(phi)/2.0), c(0.0,0.0) },
                { c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(-sin(phi)/2.0,cos(phi)/2.0) }
        };
}


std::vector<std::vector<comp>> arr_CP_dif (c_type angle) {
        return {
                { c(1.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(1.0,0.0), c(0.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(0.0,0.0), c(1.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(0.0,0.0), c(0.0,0.0),c(-sin(angle),cos(angle)) }
        };
}



// Debug info
void print_array(std::vector<std::vector<comp>> arr) {
        unsigned int dim1 = arr.size();

        for(unsigned int i = 0; i < dim1; ++i) {
                unsigned int dim2 = arr[i].size();

                for(unsigned int j = 0; j < dim2; ++j)
                        print_comp(arr[i][j]);

                std::cout << "\n";
        }

}



const SqMatrix empty(0);



// NON-PARAMETRIC MATRICES
// ===== ONE QUBIT OPERATIONS =====
const SqMatrix matrix_I(arr_I);
```

```
const SqMatrix matrix_H(arr_H);
const SqMatrix matrix_X(arr_X);
const SqMatrix matrix_Y(arr_Y);
const SqMatrix matrix_Z(arr_Z);
const SqMatrix matrix_S(arr_S);
const SqMatrix matrix_T(arr_T);

// ===== TWO QUBIT OPERATIONS =====
const SqMatrix matrix_CNOT(arr_CNOT);
const SqMatrix matrix_CY(arr_CY);
const SqMatrix matrix_CZ(arr_CZ);
const SqMatrix matrix_SWAP(arr_SWAP);

// ===== THREE QUBIT OPERATIONS =====
const SqMatrix matrix_TOFFOLI(arr_TOFFOLI);


// PARAMETRIC MATRICES
// ===== ONE QUBIT OPERATIONS =====
SqMatrix matrix_RX(c_type angle) { return SqMatrix(arr_RX(angle)); }
SqMatrix matrix_RY(c_type angle) { return SqMatrix(arr_RY(angle)); }
SqMatrix matrix_RZ(c_type angle) { return SqMatrix(arr_RZ(angle)); }
SqMatrix matrix_P (c_type angle) { return SqMatrix(arr_P (angle)); }
// SqMatrix matrix_ROT(c_type phi, c_type theta, c_type omega) { return SqMatrix(arr_ROT(phi, theta, omega)); }
// ===== TWO QUBIT OPERATIONS =====
SqMatrix matrix_CRX(c_type angle) { return SqMatrix(arr_CRX(angle)); }
SqMatrix matrix_CRY(c_type angle) { return SqMatrix(arr_CRY(angle)); }
SqMatrix matrix_CRZ(c_type angle) { return SqMatrix(arr_CRZ(angle)); }
SqMatrix matrix_CP(c_type angle) { return SqMatrix(arr_CP(angle)); }



// DERIVATIVES OF GATES
// ===== ONE QUBIT OPERATIONS =====
SqMatrix matrix_RX_dif(c_type angle) { return SqMatrix(arr_RX_dif(angle)); }
SqMatrix matrix_RY_dif(c_type angle) { return SqMatrix(arr_RY_dif(angle)); }
SqMatrix matrix_RZ_dif(c_type angle) { return SqMatrix(arr_RZ_dif(angle)); }
SqMatrix matrix_P_dif (c_type angle) { return SqMatrix(arr_P_dif (angle)); }
// SqMatrix matrix_ROT(c_type phi, c_type theta, c_type omega) { return SqMatrix(arr_ROT(phi, theta, omega)); }
// ===== TWO QUBIT OPERATIONS =====
SqMatrix matrix_CRX_dif(c_type angle) { return SqMatrix(arr_CRX_dif(angle)); }
SqMatrix matrix_CRY_dif(c_type angle) { return SqMatrix(arr_CRY_dif(angle)); }
SqMatrix matrix_CRZ_dif(c_type angle) { return SqMatrix(arr_CRZ_dif(angle)); }
SqMatrix matrix_CP_dif(c_type angle) { return SqMatrix(arr_CP_dif(angle)); }
#endif
```

10.6 application

```
#ifndef __APPLICATION__
#define __APPLICATION__ 420

#include <iostream>
#include <vector>

#include "parameters.h"
#include "comp.h"
#include "algebra.h"



// GATE APPLICATION
class Application {
        public:
                // gate
                SqMatrix application_gate;
                // gate derivative
                SqMatrix application_gate_dif;
                // qubits this gate is applied unto
                std::vector<unsigned int> qubits;
```

```
                // Application Constructor
                Application(SqMatrix g, SqMatrix g_dif, std::vector<unsigned int> q )
                : application_gate(g), application_gate_dif(g_dif), qubits(q)

                {
#ifdef __SAFE_LIBRARY__
                        // Check qubits-gate dimensions!
                        unsigned int qubs = q.size();
                        if (g.dims() != (unsigned int) (1<<qubs)) {
                                std::cout << "Application Constructor Error:"
                                        "qubits don't match array size\n";
                                        exit(1);
                        }

                        // Check for duplicates in qubit list
                        std::vector<bool> qubit_used(MAX_QUBITS, false);
                        for(unsigned int i = 0; i < qubs; ++i) {
                                if(qubit_used[q[i]] == true) {
                                        std::cout << "Application Constructor Error: "
                                                "Dulpicate Qubit Detected!\n";
                                        exit(1);
                                }
                                qubit_used[q[i]] = true;
                        }
#endif
                };
};
#endif
```

10.7 circuit

```
#ifndef __QUANTUM_CIRCUIT__
#define __QUANTUM_CIRCUIT__ 420


#include <iostream>         // input/output
#include <vector>         // all my arrays
#include <map>                  // dictionary for measure();
#include <bitset>    // convert integer to bin_string
#include <random>         // discrete_distribution
#include <math.h>         // floor
#include <unordered_map>          // application SN --> actual gate
#include <tuple>         // (Layer, Layer_number)

#include "parameters.h"
#include "algebra.h"
#include "comp.h"
#include "application.h"
#include "gate_library.h"



#define max(a,b) ((a)>(b)?(a):(b))


class Circuit {
        private:
                // Core Utilities
                unsigned int qubits;
                std::vector<unsigned int> available_layer;
                std::vector< std::vector<Application> > layers;
                std::vector<comp> state_vector;
                bool executed;


                // Adjoint Differentiation
                // a map that keeps track of application Serial Numbers
                std::unordered_map<unsigned int,
                        std::tuple<unsigned int, unsigned int>> app_id;
```

```
                unsigned int applications;
                unsigned int parametric_gates;


        public:
                // Constructor
                Circuit(unsigned int q);



                // =========== circuit_core.h =========== //
                void app (Application &a);
                void run();
                c_type exp_value (
                        const SqMatrix &Obs,
                        const std::vector<unsigned int> &qubs);

                // returns Nabla <E> for an observable E
                std::vector<c_type> exp_value_grad(
                        const SqMatrix &Obs,
                        const std::vector<unsigned int> &qubs);



                // ======== circuit_statistics.h ======== //
                void print_circuit();
                void print_state_vector(bool integer);
                void print_probabilities(bool integer, bool cent);
                void measure(unsigned int shots);




                // ========== circuit_library.h ========= //
                // NON-PARAMETRIC GATES
                void I(unsigned int qubit);
                void H(unsigned int qubit);
                void X(unsigned int qubit);
                void Y(unsigned int qubit);
                void Z(unsigned int qubit);
                void S(unsigned int qubit);
                void T(unsigned int qubit);
                void CNOT(unsigned int control, unsigned int target);
                void CY(unsigned int control, unsigned int target);
                void CZ(unsigned int control, unsigned int target);
                void SWAP(unsigned int qubit1, unsigned int qubit2);

                // PARAMETRIC GATES
                void RX(unsigned int qubit, c_type angle);
                void RY(unsigned int qubit, c_type angle);
                void RZ(unsigned int qubit, c_type angle);
                void P (unsigned int qubit, c_type angle);
                // void ROT(unsigned int qubit, c_type phi, c_type theta, c_type omega);
                void CRX(unsigned int control, unsigned int target, c_type angle);
                void CRY(unsigned int control, unsigned int target, c_type angle);
                void CRZ(unsigned int control, unsigned int target, c_type angle);
                void CP(unsigned int control, unsigned int target, c_type angle);
                void TOFFOLI(unsigned int control1, unsigned int control2, unsigned int target);
};


// Constructor
Circuit::Circuit(unsigned int q) {
        #ifdef __SAFE_LIBRARY__
                // check qubit range
                if(q == 0) {
                        std::cout << "Zero Qubits Given!\n";
                        exit(1);
                }
```

```
                if(q > MAX_QUBITS) {
                        std::cout << "More than " << MAX_QUBITS
                                << " Qubits! You are a mad man....\n";
                        exit(1);
                }
        #endif


        // Initialize data
        state_vector = std::vector<comp> ( 1<<q, c(0.0, 0.0) );
        state_vector[0] = c(1.0, 0.0);
        qubits = q;
        available_layer = std::vector<unsigned int> (q,0);
        applications = 0;
        parametric_gates = 0;
        executed = false;
};
#endif
```

10.8 circuit_core

```
#include "circuit.h"


// Circuit Core utilities


// ========== APPLY GATE ON CIRCUIT ========== //
void Circuit::app(Application &a) {

        unsigned int pos_in_layer;

        // ========== FIND THE RIGHT LAYER ========== //
        unsigned int layer_index = 0;
        unsigned int max_qubit_used = 0;
        for(unsigned int i = 0; i < a.qubits.size(); ++i) {
                layer_index = max(layer_index, available_layer[a.qubits[i]]);
                max_qubit_used = max(max_qubit_used, a.qubits[i]);
        }


#ifdef __SAFE_LIBRARY__
        // ========== QUBIT OUT OF RANGE ========== //
        if(max_qubit_used > qubits - 1) {
                std::cout << "Application Error: "
                        "Qubit " << max_qubit_used << " out of Range! "
                        "(0, " << qubits - 1 << ")\n";
                exit(1);
        }
#endif


        // ========== NEW CIRCUIT LAYER? ========== //
        if( layers.size() < layer_index + 1) {
                layers.push_back( std::vector<Application> {a} );
                pos_in_layer = 0;
        }
        else {
                layers[layer_index].push_back(a);
                pos_in_layer = layers[layer_index].size() - 1;
        }


        // ========== UPDATE AVAILABLE QUBIT POSITIONS ========== //
        for(unsigned int i = 0; i < a.qubits.size(); ++i)
                available_layer[a.qubits[i]] = layer_index + 1;


        // map application's Serial Number to its place on the circuit
        app_id[applications++] = std::make_tuple(layer_index, pos_in_layer);
```

```
        executed = false;
}



// apply gate on State_Vector
// given a permutation map for the qubits
void apply_on_map(
                std::vector<comp> &SV,
                std::vector<unsigned int> &permutation,
                unsigned int qubits,
                SqMatrix &gate)
{
        // =========== MAP STATEVECTOR =========== //
        std::vector<comp> mapped_state_vector = SV;

        // iterate state vector and map fake qubits to real qubits
        unsigned int SV_size = 1 << qubits;
        for(unsigned int pos = 0; pos < SV_size; ++pos) {

                unsigned int mapped_pos = 0;
                for(unsigned int fake = 0; fake < qubits; ++fake) {
                        unsigned int real = permutation[fake];
                        // replace fake qubit with the real one
                        unsigned int real_bit = (pos >> real) & 1;
                        mapped_pos |= (real_bit << fake);
                }
                mapped_state_vector[mapped_pos] = SV[pos];
        }

        // =========== APPLY LAYER =========== //
        mapped_state_vector =
                Sparse_Array_dot_state(gate, mapped_state_vector);

        // =========== UNDO MAPPING OF STATEVECTOR =========== //
        for(unsigned int pos = 0; pos < SV_size; ++pos) {

                unsigned int mapped_pos = 0;
                for (unsigned int fake = 0; fake < qubits; ++fake) {
                        unsigned int real = permutation[fake];
                        unsigned int real_bit = (pos >> real) & 1;
                        mapped_pos |= (real_bit << fake);
                }
                // reverse map
                SV[pos] = mapped_state_vector[mapped_pos];
        }
}


// returns qubits used for gates
void make_permutation(
                std::vector<unsigned int> & permutation,
                unsigned int qubits,
                const std::vector<Application> & layer)
{
        std::vector<bool> allocated(qubits, false);

        // PREMARE MAP AND PARAMETERS
        int gates = layer.size();
        unsigned int last_used_qubit = 0;

        // ITERATE LAYER OF GATES
        for(int i = 0; i < gates; ++i) {

                unsigned int gate_size = layer[i].qubits.size();

                // ITERATE GATE
                for(unsigned int qubit = 0; qubit < gate_size; ++qubit) {
```

```
                        permutation[qubit + last_used_qubit] = layer[i].qubits[qubit];
                        allocated[layer[i].qubits[qubit]] = true;
                }
                last_used_qubit += gate_size; // qubits used on this gate
        }

        // =========== FILL PERMUTATION WITH UNALLOCATED WIRES =========== //
        for(unsigned int un_it = 0; un_it < qubits; ++un_it) {
                if(allocated[un_it] == false)
                        permutation[last_used_qubit++] = un_it;
        }
}



// Tensor product of many gates
void fill_gate(
                SqMatrix & layer_array,
                const std::vector<Application> & layer)
{
        // Layer (by definition) is never empty!
        layer_array = layer[0].application_gate;

        // fill with gates given
        int gates = layer.size();
        for(int i = 1; i < gates; ++i) {
                layer_array = layer[i].application_gate % layer_array;
        }
}


// Tensor product of many gates + fill with Identities
void full_fill_gate(
                SqMatrix & layer_array,
                const std::vector<Application> & layer,
                unsigned int qubits )
{
        // Layer (by definition) is never empty!
        layer_array = layer[0].application_gate;

        // fill with gates given
        int gates = layer.size();
        for(int i = 1; i < gates; ++i) {
                layer_array = layer[i].application_gate % layer_array;
        }

        unsigned int rest_wires_dim = (1<<qubits) / layer_array.dims();

        SqMatrix big_id = Identity_matrix(rest_wires_dim);

        layer_array = big_id % layer_array;
}



// =========== RUN AND UPDATE STATE VECTOR =========== //
void Circuit::run(){

        // don't run the same circuit twice
        if(executed) return;

        // =========== CLEAR STATE VECTOR =========== //
        unsigned int SV_size = 1 << qubits;
        state_vector[0] = c(1.0, 0.0);
        for(unsigned int i = 1; i < SV_size; ++i)
                state_vector[i] = c(0.0, 0.0);


        // =========== LAYER BY LAYER =========== //
        int depth = layers.size();
```

```
        for(int i = 0; i < depth; ++i) {


                // IDEA: Split full layers to half-half
                // Maybe it is a good optimization for full layers
                // Needs proof

                std::vector<unsigned int> permutation(qubits);

                make_permutation(permutation, qubits, layers[i]);
                SqMatrix layer_array(0);        // init as emptys
                fill_gate(layer_array, layers[i]);
                apply_on_map(state_vector, permutation, qubits, layer_array);

        }

        executed = true;
}



// Takes A and list of qubits A is applied onto
// Returns:   <A> = <Psi|A|Psi>
c_type Circuit::exp_value (
                const SqMatrix &Obs,
                const std::vector<unsigned int> &qubs)
{
#ifdef __SAFE_LIBRARY__
        // =========== COMPATIBILITY CHECKS =========== //
        unsigned int qub_len = 1 << qubs.size();
        if(Obs.dims() != qub_len) {
                std::cout << "Expectation Value Error: "
                        "Observable matrix size doesn't match qubits given\n";
                exit(1);
        }
        // =========== QUBIT CHECKS =========== //
        std::vector<bool> qubit_used(MAX_QUBITS, false);
        for(unsigned int i = 0; i < qubs.size(); ++i) {
                // Check range
                if(qubs[i] > qubits - 1) {
                        std::cout << "Expectation Value Error: "
                                "Qubit " << qubs[i] << " out of Range "
                                "(0," << qubits-1 << ")\n";
                        exit(1);
                }
                // Check duplicates
                if(qubit_used[qubs[i]] == true) {
                        std::cout << "Expectation Value Error: "
                                "Dulpicate Qubit Detected!\n";
                        exit(1);
                }
                qubit_used[qubs[i]] = true;
        }
#endif

        run();

        // Like run(); only one layer containing Obs gate
        std::vector<Application> layer = {Application(Obs, empty, qubs)};

        std::vector<unsigned int> permutation(qubits);

        make_permutation(permutation, qubits, layer);

        SqMatrix layer_array(0);        // empty
        full_fill_gate(layer_array, layer, qubits);

        // Don't call apply_on_map since we don't need to unmap
        // =========== MAP STATEVECTOR =========== //
        std::vector<comp> mapped_state_vector = state_vector;
```

```
        // iterate state vector and map real qubits to fake qubits
        unsigned int SV_size = 1 << qubits;
        for(unsigned int pos = 0; pos < SV_size; ++pos) {

                unsigned int mapped_pos = 0;
                for(unsigned int fake = 0; fake < qubits; ++fake) {
                        unsigned int real = permutation[fake];
                        // replace fake qubit with the real one
                        unsigned int real_bit = (pos >> real) & 1;
                        mapped_pos |= (real_bit << fake);
                }
                mapped_state_vector[mapped_pos] = state_vector[pos];
        }

        c_type exp_val = expectation_value(layer_array, mapped_state_vector);

        return exp_val;
}



// https://arxiv.org/pdf/2009.02823.pdf
// returns Nabla <E> for an observable E
std::vector<c_type> Circuit::exp_value_grad(
                const SqMatrix &Obs,
                const std::vector<unsigned int> &qubs)
{

#ifdef __SAFE_LIBRARY__
        // =========== COMPATIBILITY CHECKS =========== //
        unsigned int qub_len = 1 << qubs.size();
        if(Obs.dims() != qub_len) {
                std::cout << "Expectation Value Error: "
                        "Observable matrix size doesn't match qubits given\n";
                exit(1);
        }
        // =========== QUBIT CHECKS =========== //
        std::vector<bool> qubit_used(MAX_QUBITS, false);
        for(unsigned int i = 0; i < qubs.size(); ++i) {
                // Check range
                if(qubs[i] > qubits - 1) {
                        std::cout << "Expectation Value Error: "
                                "Qubit " << qubs[i] << " out of Range "
                                "(0," << qubits-1 << ")\n";
                        exit(1);
                }
                // Check duplicates
                if(qubit_used[qubs[i]] == true) {
                        std::cout << "Expectation Value Error: "
                                "Dulpicate Qubit Detected!\n";
                        exit(1);
                }
                qubit_used[qubs[i]] = true;
        }
#endif


        run();


        // result vector --> Nabla <E>
        std::vector<c_type> grad (parametric_gates);
        unsigned int param = parametric_gates - 1;

        // Find <E>
        // YOU MUST RUN THE CIRCUIT FIRST!
        run();
        std::vector<comp> phi    = state_vector;
```

```
std::vector<comp> lambda = state_vector;


// ============================================== //
// ======= |Lambda> = Observable |Lambda> ======= //
// ============================================== //

// A layer containing only Obs array
std::vector<Application> layer = {Application(Obs, empty, qubs)};
std::vector<unsigned int> perm(qubits);

// PERMUTE
make_permutation(perm, qubits, layer);
// MAKE GATE
SqMatrix filled_gate(0);        // init as empty
fill_gate(filled_gate, layer);
// APPLY GATE ON GIVEN STATEVECTOR WITH GIVEN MAP
apply_on_map(lambda, perm, qubits, filled_gate);




// ============================================== //
// ========== ITERATE GATES IN REVERSE ========== //
// ============================================== //


// iterate backwards all gates
for(int app = applications-1; app >= 0; --app) {

        // look up in hash --> find gate in circuit
        std::tuple coordinates = app_id[app];
        Application gate =
                layers[std::get<0>(coordinates)][std::get<1>(coordinates)];


        // ================================================ //
        // ============ |Phi> = Ui^dag |Phi> ============ //
        // ================================================ //

        SqMatrix arr = dagger(gate.application_gate);
        std::vector<unsigned int> gate_qubs = gate.qubits;

        // Prepare Layer and permutation
        std::vector<Application> layer = {Application(arr, empty, gate_qubs)};
        std::vector<unsigned int> permutation(qubits);


        make_permutation(permutation, qubits, layer);
        SqMatrix filled_gate(0);        // init as empty
        fill_gate(filled_gate, layer);
        apply_on_map(phi, permutation, qubits, filled_gate);


        // ================================================ //
        // ======= |Mi> = (dUi^dag / dthetai) |Mi> ====== //
        // ================================================ //

        arr = gate.application_gate_dif;

        // PARAMETRIC GATE!
        if(arr.dims() > 1)
        {
                std::vector<comp> mi = phi;

                layer = {Application(arr, empty, gate_qubs)};
                make_permutation(permutation, qubits, layer);

                filled_gate = SqMatrix(0);        // init as empty
                fill_gate(filled_gate, layer);
```

```
                        apply_on_map(mi, permutation, qubits, filled_gate);

                        // Nabla <E>i
                        grad[param--] = 2.0 * inner_product(lambda, mi).real;
                }


                // UPDATE LAMBDA
                if(param > 0) {

                // ============================================== //
                // ========= |Lambda> = Ui^dag |Lambda> ========= //
                // ============================================== //

                arr = dagger(gate.application_gate);

                // Prepare Layer and permutation
                layer = {Application(arr, empty, gate_qubs)};
                make_permutation(permutation, qubits, layer);

                filled_gate = SqMatrix(0);        // init as empty
                fill_gate(filled_gate, layer);
                apply_on_map(lambda, permutation, qubits, filled_gate);
                }


        } // applications loop

        return grad;
}
```

10.9 circuit_library

```
#include "circuit.h"


// ============================================== //
// ============ STANDARD GATE LIBRARY ============ //
// ============================================== //


void Circuit::I(unsigned int qubit){
        Application a(matrix_I, empty, {qubit});
        app(a);
}
void Circuit::H(unsigned int qubit){
        Application a(matrix_H, empty, {qubit});
        app(a);
}
void Circuit::X(unsigned int qubit){
        Application a(matrix_X, empty, {qubit});
        app(a);
}
void Circuit::Y(unsigned int qubit){
        Application a(matrix_Y, empty, {qubit});
        app(a);
}
void Circuit::Z(unsigned int qubit){
        Application a(matrix_Z, empty, {qubit});
        app(a);
}
void Circuit::S(unsigned int qubit){
        Application a(matrix_S, empty, {qubit});
        app(a);
}
void Circuit::T(unsigned int qubit){
        Application a(matrix_T, empty, {qubit});
        app(a);
}
```

```
void Circuit::CNOT(unsigned int control, unsigned int target){
        Application a(matrix_CNOT, empty, {target, control});
        app(a);
}
void Circuit::CY(unsigned int control, unsigned int target){
        Application a(matrix_CY, empty, {target, control});
        app(a);
}
void Circuit::CZ(unsigned int control, unsigned int target){
        Application a(matrix_CZ, empty, {target, control});
        app(a);
}
void Circuit::SWAP(unsigned int qubit1, unsigned int qubit2){
        Application a(matrix_SWAP,  empty,{qubit1, qubit2});
        app(a);
}

void Circuit::TOFFOLI(unsigned int control1, unsigned int control2, unsigned int target){
        Application a(matrix_TOFFOLI, empty, {target, control1, control2});
        app(a);
}



// include derivative for these

void Circuit::RX(unsigned int qubit, c_type angle){
        Application a(matrix_RX(angle), matrix_RX_dif(angle), {qubit});
        app(a);
        parametric_gates++;
}
void Circuit::RY(unsigned int qubit, c_type angle){
        Application a(matrix_RY(angle), matrix_RY_dif(angle), {qubit});
        app(a);
        parametric_gates++;
}
void Circuit::RZ(unsigned int qubit, c_type angle){
        Application a(matrix_RZ(angle), matrix_RZ_dif(angle), {qubit});
        app(a);
        parametric_gates++;
}
void Circuit::P (unsigned int qubit, c_type angle){
        Application a(matrix_P(angle), matrix_P_dif(angle), {qubit});
        app(a);
        parametric_gates++;
}
void Circuit::CRX(unsigned int control, unsigned int target, c_type angle){
        Application a(matrix_CRX(angle), matrix_CRX_dif(angle), {target, control});
        app(a);
        parametric_gates++;
}
void Circuit::CRY(unsigned int control, unsigned int target, c_type angle){
        Application a(matrix_CRY(angle), matrix_CRY_dif(angle), {target, control});
        app(a);
        parametric_gates++;
}
void Circuit::CRZ(unsigned int control, unsigned int target, c_type angle){
        Application a(matrix_CRZ(angle), matrix_CRZ_dif(angle), {target, control});
        app(a);
        parametric_gates++;
}
void Circuit::CP(unsigned int control, unsigned int target, c_type angle){
        Application a(matrix_CP(angle), matrix_CP_dif(angle), {target, control});
        app(a);
        parametric_gates++;
}
```

10.10 circuit_statistics

```
#include "circuit.h"

// ================================================ //
// ============== CIRCUIT STATISTICS ============== //
// ================================================ //



// Print StateVector in binary - integer
void Circuit::print_state_vector(bool integer = false) {

    std::cout << "============ Printing State Vector ============ \n";

        unsigned int dim = state_vector.size();
        for(unsigned int base_state = 0; base_state < dim; ++base_state){

                // Print base state in binary
                if(integer == false) {
                std::string filled_base =
                    std::bitset< MAX_QUBITS >(base_state).to_string();
                std::string base_string(filled_base, MAX_QUBITS - qubits, qubits);

                std::cout << "|" << base_string << ">\t" << "= ";
            }
            // Print base state as integer
            else {
                    std::cout << "|" << base_state << ">\t" << "= ";
            }
                print_comp(state_vector[base_state]);
                std::cout << "\n";
        }
}



// Debug info --> Print layer
void Circuit::print_circuit() {
        std::cout << "============= Printing Circuit ============= \n";

        for(unsigned int i = 0; i < layers.size(); ++i) {
                std::cout << "===== Printing Layer: " << i << " =====\n";

                for(unsigned int j = 0; j < layers[i].size(); ++j){
                        std::cout << "Gate:\n";
                        print_matrix(layers[i][j].application_gate);

                        std::cout << "Applied on Qubit(s): ";
                        for(unsigned int k = 0; k < layers[i][j].qubits.size(); ++k)
                                std::cout << layers[i][j].qubits[k] << " ";
                        std::cout << "\n";
                }
        }
}




// Print probabilities of base_states
void Circuit::print_probabilities(bool integer = false, bool cent = false) {
    std::cout << "============ Printing probabilities ============ \n";

        unsigned int dim = state_vector.size();
        for(unsigned int base_state = 0; base_state < dim; ++base_state){

                // Print base state in binary
                if(integer == false) {
                std::string filled_base =
```

```
                    std::bitset< MAX_QUBITS >(base_state).to_string();
                std::string base_string(filled_base, MAX_QUBITS - qubits, qubits);

                    std::cout << "P(|" << base_string << ">)\t" << "= ";
                }
                // Print base state as integer
                else {
                        std::cout << "P(|" << base_state << ">)\t" << "= ";
                }
                comp z = state_vector[base_state];
                // |z|^2 = a^2 + b^2
                    c_type prob = z.real * z.real + z.imag * z.imag;
                    if(cent == false)
                            std::cout << prob << "\n";
                    else
                            std::cout << prob*100 << "%\n";
        }

}



// measure
void Circuit::measure(unsigned int shots){

        std::cout << "============ Running Experiment ============ \n";
        std::cout << "Shots: " << shots << "\n";

    // Setup the random bits
    std::random_device rd;         // Random Device
    std::mt19937_64 gen(rd());         // Mersenne Twister pseudo-random generator

    // Set Distribution Weights
        unsigned int SPACE_SIZE = 1 << qubits;
        std::vector<int> weights (SPACE_SIZE);
        for(unsigned int i = 0; i < SPACE_SIZE; ++i) {
                comp z = state_vector[i];
                c_type prob = z.real * z.real + z.imag * z.imag;
                weights[i] = floor(prob*1000000);         // make integers
        }

    // Create the distribution with those weights
    std::discrete_distribution<> d(weights.begin(), weights.end());

    // Run experiment
    std::map<int, int> counts;
    for(unsigned int shot = 0; shot < shots; ++shot) {
            ++counts[d(gen)];
    }

    // Print measurements
    for(auto p : counts) {
            // Make base_state in binary
                std::string filled_base =
                        std::bitset< MAX_QUBITS >(p.first).to_string();
                std::string base_string(filled_base, MAX_QUBITS - qubits, qubits);
                // Print result
                std::cout << "#(|" << base_string << ">)\t" << "= "
                        << p.second << "\n";
    }
}
```

10.11 optimizers

```
#ifndef __OPTIMIZERS__
#define __OPTIMIZERS__ 420

#include <math.h>
#include <random>
#include <time.h>
```

```
#include "parameters.h"
#include "comp.h"


// DEFINE OPTIMIZER PARAMETERS
#define EPOCHS 500
#define STEP_SIZE 0.01
#define VERBOSE true




// random weights in range (-2pi, 2pi)
std::vector<c_type> random_weights (unsigned int weights_len) {

        srand((unsigned)time(0));
        std::vector<c_type> weights (weights_len, 0.0);

        for(unsigned int i = 0; i < weights_len; ++i) {
                int random_num = (rand()%100000); // in range (0, 999)
                // map (0,999) to (-2pi, 2pi)
                c_type resize = 2*pi/100000;// shrink to (0, 4pi)
                c_type shift  = pi;                         // offset --> 2 pi
                weights[i] = (resize*random_num) - shift;
        }

        return weights;
}




// Adagrad Optimizer
// Take cost, weights --> returns optimized weights
std::vector<c_type> ADAGRAD (
        unsigned int qubits,
        void make_circuit (Circuit &, std::vector<c_type> &),
        SqMatrix Obs,
        std::vector<unsigned int> Obs_qubs,
        std::vector<c_type> init_weights,

        unsigned int epochs = EPOCHS,
        c_type step_size = STEP_SIZE,
        c_type eps = 1e-08,
        bool verbose = VERBOSE
)
{
        // PARAMETERS
        std::vector<c_type> weights = init_weights;
        unsigned int weights_len = weights.size();
        std::vector<c_type> learning_rate (weights_len, step_size);
        std::vector<c_type> alpha (weights_len, 0.0);


        if(verbose)
                std::cout << "========== ADAGRAD =========\n"
                        "Epochs:\t" << epochs << "\t Learning Rate:\t" <<
                        step_size << "\n====================================\n";



        // iterations
        for(unsigned int epoch = 1; epoch <= epochs; ++epoch) {

                // Make Circuit
                Circuit c(qubits);
                make_circuit(c, weights);
```

```
                // Compute Grad for observable
                std::vector<c_type> grad = c.exp_value_grad(Obs, Obs_qubs);


                // UPDATE
                for(unsigned int grad_i = 0; grad_i < weights_len; ++grad_i) {
                        alpha[grad_i]  += grad[grad_i] * grad[grad_i];
                        learning_rate[grad_i] = step_size / sqrt(alpha[grad_i] + eps);
                        weights[grad_i] = weights[grad_i] - learning_rate[grad_i] * grad[grad_i];
                }

                // DEBUG PRINTING
                if(verbose && epoch % 10 == 0) {
                        std::cout << "Epoch:\t" << epoch << "\t<E>: ";
                        std::cout << c.exp_value(Obs, Obs_qubs) << "\n";
                }
        }

        return weights;
}




// Adam Optimizer
// Take cost, weights --> returns optimized weights
std::vector<c_type> ADAM(
        unsigned int qubits,
        void make_circuit (Circuit &, std::vector<c_type> &),
        SqMatrix Obs,
        std::vector<unsigned int> Obs_qubs,
        std::vector<c_type> init_weights,

        unsigned int epochs = EPOCHS,
        c_type step_size = STEP_SIZE,
        c_type beta1_init = 0.9,
        c_type beta2_init = 0.99,
        c_type eps = 1e-08,
        bool verbose = VERBOSE
)
{
        // WEIGHTS
        std::vector<c_type> weights = init_weights;
        unsigned int weights_len = weights.size();
        // PARAMETERS
        c_type learning_rate = step_size;
        c_type alpha = 0.0, beta = 0.0;
        c_type beta1 = beta1_init;
        c_type beta2 = beta2_init;


        if(verbose)
                std::cout << "=========== ADAM OPTIMIZER ==========\n"
                        "Epochs:\t" << epochs << "\t Learning Rate:\t" <<
                        learning_rate << "\n====================================\n";



        // iterations
        for(unsigned int epoch = 1; epoch <= epochs+1; ++epoch) {

                // Make Circuit
                Circuit c(qubits);
                make_circuit(c, weights);

                // Compute Grad for observable
                std::vector<c_type> grad = c.exp_value_grad(Obs, Obs_qubs);
```

```
                // UPDATE
                for(unsigned int grad_i = 0; grad_i < weights_len; ++grad_i) {
                        alpha = beta1 * alpha + (1.0 - beta1) * grad[grad_i];
                        beta  = beta2 * beta  + (1.0 - beta2) * grad[grad_i] * grad[grad_i];
                        learning_rate = step_size * sqrt(1.0 - pow(beta2, epoch))
                                / (1.0 - pow(beta1, epoch));
                        weights[grad_i] = weights[grad_i] - learning_rate * alpha / (sqrt(beta) + eps);
                }

                // DEBUG PRINTING
                if(verbose && epoch % 10 == 0) {
                        std::cout << "Epoch:\t" << epoch << "\t<E>: ";
                        std::cout << c.exp_value(Obs, Obs_qubs) << "\n";
                }
        }

        return weights;
}




// Gradient Descent Optimizer
// Take cost, weights --> returns optimized weights
std::vector<c_type> GD(
        unsigned int qubits,
        void make_circuit (Circuit &, std::vector<c_type> &),
        SqMatrix Obs,
        std::vector<unsigned int> Obs_qubs,
        std::vector<c_type> init_weights,

        unsigned int epochs = EPOCHS,
        c_type learning_rate = STEP_SIZE,
        bool verbose = VERBOSE
)
{
        if(verbose)
                std::cout << "========== GRADIENT DESCENT =========\n"
                        "Epochs:\t" << epochs << "\t Learning Rate:\t" <<
                        learning_rate << "\n====================================\n";

        // Parameters
        std::vector<c_type> weights = init_weights;
        unsigned int weights_len = weights.size();


        // iterations
        for(unsigned int epoch = 1; epoch <= epochs; ++epoch) {

                // Make Circuit
                Circuit c(qubits);
                make_circuit(c, weights);

                // Compute Grad for observable
                std::vector<c_type> grad = c.exp_value_grad(Obs, Obs_qubs);

                // Update weights
                for(unsigned int grad_i = 0; grad_i < weights_len; ++grad_i) {
                        weights[grad_i] = weights[grad_i] - learning_rate * grad[grad_i];
                }

                // Print Results
                if(verbose && epoch % 10 == 0) {
                        std::cout << "Epoch:\t" << epoch << "\t<E>: ";
                        std::cout << c.exp_value(Obs, Obs_qubs) << "\n";
                }
        }
```

```
        return weights;
}

#endif
```

11 Παράρτημα Α - Γραμμική Άλγεβρα

11.1 Βασικοί Ορισμοί

Ως **πίνακας** ορίζεται μια ορθογώνια διάταξη αντικειμένων (συνήθως αριθμών). πχ:

$$A = \begin{pmatrix} 1 & \pi & -2 \\ e & 0 & 7 \\ 3i & 9 & -\pi \end{pmatrix}$$

- Το **στοιχείο** στην γραμμή i και στήλη j συμβολίζεται α_{ij}. πχ: $\alpha_{12} = \pi$
- Ένας πίνακας διαστάσεων $n \times 1$ ονομάζεται **διάνυσμα**. πχ: $\vec{v} = \begin{pmatrix} 1.0 \\ 0 \\ e \end{pmatrix}$
- Ο πίνακας B φτιαγμένος από τον A με $\beta_{ij} = \alpha_{ji}$ ονομάζεται **ανάστροφος** και συμβολίζεται $B = A^T$
- Ο πίνακας B φτιαγμένος από τον A με $\beta_{ij} = \alpha^*_{ji}$ ονομάζεται **ερμιτιανός ή αυτοσυζυγής** και συμβολίζεται $B = A^\dagger$

Η πρόσθεση και η αφαίρεση πινάκων γίνεται κατά στοιχείο όταν έχουν ίδιες διαστάσεις. Αν A πίνακας διαστάσεων $(m \times n)$ και B πίνακας διαστάσεων $(n \times p)$ ορίζεται το **εσωτερικό γινόμενο** $\Gamma = A \cdot B$ ως εξής:

$$\gamma_{ij} = \sum_{k=1}^{n} \alpha_{ik}\beta_{kj}$$

11.2 Διανυσματικοί Χώροι

11.2.1 Μετρική - Βάσεις

Ένα μέτρο για τα στοιχεία X ενός συνόλου είναι η **Ευκλείδεια νόρμα** (ή αλλιώς μετρική) που ορίζεται ως εξής (δοθέντος εσωτερικού γινομένου):

$$\|X\| \equiv \sqrt{X^\dagger X}$$

Συνάρτηση απόστασης d μεταξύ δύο στοιχείων ορίζεται η (θετική πραγματική) τιμή

$$d(X, Y) \equiv \|X - Y\|$$

Γραμμικός συνδυασμός των διανυσμάτων $\{\alpha_i\}$ ονομάζεται κάθε παραγόμενο διάνυσμα της μορφής:

$$\beta_1\vec{\alpha_1} + \cdots + \beta_k\vec{\alpha_k}$$

Μια συλλογή k διανυσμάτων ονομάζεται **γραμμικά ανεξάρτητη** αν

$$\beta_1\vec{\alpha_1} + \cdots + \beta_k\vec{\alpha_k} = \vec{0} \iff \forall i : \beta_i = 0$$

Μια συλλογή γραμμικά ανεξάρτητων διανυσμάτων αποκαλείται **βάση**. Μια βάση ονομάζεται **ορθοκανονική** αν $\forall ij, \quad \alpha_i^T \cdot \alpha_j = \delta_{ij}$.

Το σύνολο των γραμμικών συνδυασμών μιας βάσης ονομάζεται **διανυσματικός χώρος**. Συνηθίζεται για βάση των χώρων να επιλέγουμε τα διανύσματα που έχουν μόνο έναν άσσο ως στοιχεία:

$$\vec{e_1} = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ 0 \end{pmatrix} \qquad \vec{e_2} = \begin{pmatrix} 0 \\ 1 \\ \vdots \\ 0 \\ 0 \end{pmatrix} \qquad \cdots \qquad \vec{e_{n-1}} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ 0 \end{pmatrix} \qquad \vec{e_n} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}$$

11.3 Τανυστικό γινόμενο

Το τανυστικό γινόμενο (ή αλλιώς γινόμενο Kronecker) είναι μια πράξη μεταξύ πινάκων η οποία ως αποτέλεσμα έχει έναν πίνακα μεγαλύτερων διαστάσεων. Συγκεκριμένα, για δύο πίνακες $A_{m\times n}$ και $B_{p\times q}$, το αποτέλεσμα είναι ένας πίνακας $A \otimes B_{pm\times qn}$.

Η πράξη ορίζεται ως:

$$A \otimes B = \begin{bmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{bmatrix}$$

ή πιο αναλυτικά:

$$A \otimes B = \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12} & \cdots & a_{11}b_{1q} & \cdots & \cdots & a_{1n}b_{11} & a_{1n}b_{12} & \cdots & a_{1n}b_{1q} \\ a_{11}b_{21} & a_{11}b_{22} & \cdots & a_{11}b_{2q} & \cdots & \cdots & a_{1n}b_{21} & a_{1n}b_{22} & \cdots & a_{1n}b_{2q} \\ \vdots & \vdots & \ddots & \vdots & & & \vdots & \vdots & \ddots & \vdots \\ a_{11}b_{p1} & a_{11}b_{p2} & \cdots & a_{11}b_{pq} & \cdots & \cdots & a_{1n}b_{p1} & a_{1n}b_{p2} & \cdots & a_{1n}b_{pq} \\ \vdots & \vdots & & \vdots & \ddots & & \vdots & \vdots & & \vdots \\ \vdots & \vdots & & \vdots & & \ddots & \vdots & \vdots & & \vdots \\ a_{m1}b_{11} & a_{m1}b_{12} & \cdots & a_{m1}b_{1q} & \cdots & \cdots & a_{mn}b_{11} & a_{mn}b_{12} & \cdots & a_{mn}b_{1q} \\ a_{m1}b_{21} & a_{m1}b_{22} & \cdots & a_{m1}b_{2q} & \cdots & \cdots & a_{mn}b_{21} & a_{mn}b_{22} & \cdots & a_{mn}b_{2q} \\ \vdots & \vdots & \ddots & \vdots & & & \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{p1} & a_{m1}b_{p2} & \cdots & a_{m1}b_{pq} & \cdots & \cdots & a_{mn}b_{p1} & a_{mn}b_{p2} & \cdots & a_{mn}b_{pq} \end{bmatrix}$$

Και αν συμβολίσουμε με // και % το πηλίκο και το υπόλοιπο της ακέραιας διαίρεσης, τότε έχουμε έναν κλειστό τύπο για τα στοιχεία του τελικού πίνακα:

$$(A \otimes B)_{i,j} = a_{i//p,j//q} b_{i\%p,j\%q}$$

(υποθέτοντας οτι η αρίθμηση ξεκινάει από το μηδέν)

11.3.1 Εξισώσεις Ιδιοτιμών

Στην γραμμική άλγεβρα, ιδιοδιάνυσμα ενός γραμμικού μετασχηματισμού (στην περίπτωσή μας πίνακα) είναι ένα μη-μηδενικό διάνυσμα το οποιό αλλάζει το πολύ κατά μια σταθερά όταν ο εν λόγω μετασχηματισμός εφαρμοστεί πάνω του.
Η σταθερά αυτή ονομάζεται ιδιοτιμή και η εξίσωση ιδιοτιμών είναι:

$$\hat{A}\vec{v} = \lambda\vec{v}$$

όπου $\hat{A}$ ο μετασχηματισμός, $\vec{v}$ το ιδιοδιάνυσμα και λ η ιδιοτιμή.

Για μοναδιαίους πίνακες, όπως αυτούς που χρησιμοποιούμε στους κβαντικούς υπολογιστές:

$$\hat{U}\left|\psi\right\rangle = e^{i2\pi\phi}\left|\psi\right\rangle$$

12 Παράρτημα Β - Πιθανότητες & Στατιστική

12.1 Πείραμα τύχης

Πείραμα τύχης ονομάζουμε μιαν φυσική διαδικασία της οποίας δεν μπορούμε να προβλέψουμε το αποτέλεσμα· για παράδειγμα μια ρίψη ζαριού. Τα πιθανά αποτελέσματα τα καλούμε ενδεχόμενα και μπορεί να είναι διακριτά (πχ ρίψη νομίσματος) ή συνεχή (η χρονική στιγμή που συμβαίνει ένας σεισμός).

12.2 Ορισμός Πιθανότητας

Όταν το πείραμα είναι πραγματικά τυχαίο και δεν μεροληπτεί, τότε περιμένουμε να είναι δίκαιο. Για πειράματα με διακριτά ενδεχόμενα, ορίζουμε ως πιθανότητα ενός συμβάντος τον λόγο των ευνοϊκών ενδεχομένων αυτού προς το συνολικό πλήθος των ενδεχομένων. Για παράδειγμα, η πιθανότητα να φέρουμε ζυγό αριθμό με ένα ζάρι είναι $3/6=1/2$.

12.3 Κατανομή Πιθανότητας

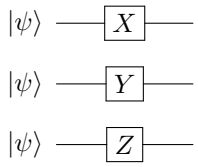
Ονομάζουμε κατανομή πιθανότητας την συνάρτηση η οποία υπολογίζει την πιθανότητα του κάθε ενδεχομένου. Φυσικά, οι τιμές της είναι μη-αρνητικές και το άθροισμα των τιμών της σε όλο το πεδίο ορισμού είναι ίσο με ένα.

13 Βασικές Κβαντικές Πύλες

13.1 Πύλες του ενός Qubit

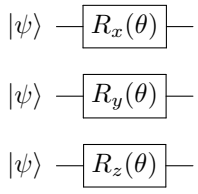
13.1.1 Πύλες Pauli

Οι πιο βασικές πύλες είναι οι πύλες του Pauli. Οι 3 πύλες αυτές περιστρέφουν το Qubit γύρω από τους 3 άξονες X,Y,Z στην σφαίρα του Bloch κατά 180 μοίρες.



13.1.2 Πύλες στροφής

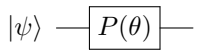
Πέραν από τις πύλες Pauli που στρέφουν τα σημεία κατά 180 μοίρες, υπάρχουν και οι πύλες Rx, Ry, Rz που κάνουν ελεγχόμενη στροφή.



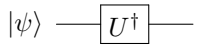
Επίσης, βασικη πύλη είναι η πύλη φάσης:

$$P(\lambda) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{pmatrix}$$

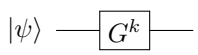
13.1.3 Λοιπές Πύλες



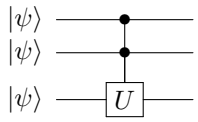
Πολύ συχνά χρησιμοποιούμε τις αντίστροφες πύλες για να αναιρέσουμε μια διαδικασία. Οι πύλες αυτές αντιστοιχούν στον ανάστροφο - συζυγή πίνακα της αρχικής πύλης.



Χρήσιμη είναι και η ύψωση μιας πύλης σε δύναμη. Εξ ορισμού: μια πύλη υψώνεται στην δύναμη k, όταν επαναλαμβάνεται k φορές.



Η πιο γενική πύλη που μπορεί να κατασκευαστεί είναι η $U = e^{ia} R_z(\beta) R_y(\gamma) R_z(\delta)$



13.2 Πύλες των πολλών Qubits

Όλες οι πύλες του ενός Qubit μπορούν να ελέγχονται από πολλά Qubits.

Αναφορές

[1] Frank Laloe Claude Cohen-Tannoudji Bernard Diu. *Quantum Mechanics. Volumne 1*. Wiley, 1991.

[2] I. Chuang M. Nielsen. *Quantum Computation and Quantum Information. 10th Anniversary Edition*. Cambridge University Press, 2010.

[3] Charles H. Bennett and Stephen J. Wiesner. "Communication via one- and two-particle operators on Einstein-Podolsky-Rosen states". In: *Phys. Rev. Lett.* 69 (20 Nov. 1992), pp. 2881–2884. DOI: 10.1103/PhysRevLett.69.2881. URL: https://link.aps.org/doi/10.1103/PhysRevLett.69.2881.

[4] D. Coppersmith. *An approximate Fourier transform useful in quantum factoring*. 2002. DOI: 10.48550/ARXIV.QUANT-PH/0201067. URL: https://arxiv.org/abs/quant-ph/0201067.

[5] Lidia Ruiz-Perez and Juan Carlos Garcia-Escartin. "Quantum arithmetic with the quantum Fourier transform". In: *Quantum Information Processing* 16.6 (Apr. 2017). DOI: 10.1007/s11128-017-1603-1. URL: https://doi.org/10.1007%2Fs11128-017-1603-1.

[6] Peter W. Shor. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer". In: *SIAM Journal on Computing* 26.5 (Oct. 1997), pp. 1484–1509. DOI: 10.1137/s0097539795293172. URL: https://doi.org/10.1137%2Fs0097539795293172.

[7] Lov K. Grover. *A fast quantum mechanical algorithm for database search*. 1996. DOI: 10.48550/ARXIV.QUANT-PH/9605043. URL: https://arxiv.org/abs/quant-ph/9605043.

[8] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. DOI: 10.48550/ARXIV.1412.6980. URL: https://arxiv.org/abs/1412.6980.

[9] Keiron O'Shea and Ryan Nash. *An Introduction to Convolutional Neural Networks*. 2015. DOI: 10.48550/ARXIV.1511.08458. URL: https://arxiv.org/abs/1511.08458.

[10] Ian J. Goodfellow et al. *Generative Adversarial Networks*. 2014. DOI: 10.48550/ARXIV.1406.2661. URL: https://arxiv.org/abs/1406.2661.

[11] Qi Gao et al. "Applications of quantum computing for investigations of electronic transitions in phenylsulfonyl-carbazole TADF emitters". In: *npj Computational Materials* 7.1 (May 2021). DOI: 10.1038/s41524-021-00540-6. URL: https://doi.org/10.1038%2Fs41524-021-00540-6.

[12] Tameem Albash and Daniel A. Lidar. "Adiabatic quantum computation". In: *Reviews of Modern Physics* 90.1 (Jan. 2018). DOI: 10.1103/revmodphys.90.015002. URL: https://doi.org/10.1103%2Frevmodphys.90.015002.

[13] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. *A Quantum Approximate Optimization Algorithm*. 2014. DOI: 10.48550/ARXIV.1411.4028. URL: https://arxiv.org/abs/1411.4028.

[14] Ritajit Majumdar et al. *Optimizing Ansatz Design in QAOA for Max-cut*. 2021. DOI: 10.48550/ARXIV.2106.02812. URL: https://arxiv.org/abs/2106.02812.

[15] Pierre Dupuy de la Grand'rive and Jean-Francois Hullo. *Knapsack Problem variants of QAOA for battery revenue optimisation*. 2019. DOI: 10.48550/ARXIV.1908.02210. URL: https://arxiv.org/abs/1908.02210.

[16] Vojtěch Havlíček et al. "Supervised learning with quantum-enhanced feature spaces". In: *Nature* 567.7747 (Mar. 2019), pp. 209–212. DOI: 10.1038/s41586-019-0980-2. URL: https://doi.org/10.1038%2Fs41586-019-0980-2.

[17] Maxwell Henderson et al. *Quanvolutional Neural Networks: Powering Image Recognition with Quantum Circuits*. 2019. DOI: 10.48550/ARXIV.1904.04767. URL: https://arxiv.org/abs/1904.04767.

[18] He-Liang Huang et al. "Experimental Quantum Generative Adversarial Networks for Image Generation". In: *Physical Review Applied* 16.2 (Aug. 2021). DOI: 10.1103/physrevapplied.16.024051. URL: https://doi.org/10.1103%2Fphysrevapplied.16.024051.

[19] Daniel Koch et al. *Fundamentals In Quantum Algorithms: A Tutorial Series Using Qiskit Continued*. 2020. DOI: 10.48550/ARXIV.2008.10647. URL: https://arxiv.org/abs/2008.10647.

[20] Ville Bergholm et al. *PennyLane: Automatic differentiation of hybrid quantum-classical computations*. 2018. DOI: 10.48550/ARXIV.1811.04968. URL: https://arxiv.org/abs/1811.04968.

[21] K. Mitarai et al. "Quantum circuit learning". In: *Physical Review A* 98.3 (Sept. 2018). DOI: 10.1103/physreva.98.032309. URL: https://doi.org/10.1103%2Fphysreva.98.032309.

[22] Maria Schuld et al. "Evaluating analytic gradients on quantum hardware". In: *Physical Review A* 99.3 (Mar. 2019). DOI: 10.1103/physreva.99.032331. URL: https://doi.org/10.1103%2Fphysreva.99.032331.

[23] Tyson Jones and Julien Gacon. *Efficient calculation of gradients in classical simulations of variational quantum algorithms*. 2020. DOI: 10.48550/ARXIV.2009.02823. URL: https://arxiv.org/abs/2009.02823.

[24] THOMAS SCHULTE-HERBRÜGGEN et al. "GRADIENT FLOWS FOR OPTIMIZATION IN QUANTUM INFORMATION AND QUANTUM DYNAMICS: FOUNDATIONS AND APPLICATIONS". In: *Reviews in Mathematical Physics* 22.06 (July 2010), pp. 597–667. DOI: 10.1142/s0129055x10004053. URL: https://doi.org/10.1142%2Fs0129055x10004053.

[25] Roeland Wiersema and Nathan Killoran. *Optimizing quantum circuits with Riemannian gradient flow*. 2022. DOI: 10.48550/ARXIV.2202.06976. URL: https://arxiv.org/abs/2202.06976.

[26] James Stokes et al. "Quantum Natural Gradient". In: *Quantum* 4 (May 2020), p. 269. DOI: 10.22331/q-2020-05-25-269. URL: https://doi.org/10.22331%2Fq-2020-05-25-269.

[27] Mateusz Ostaszewski, Edward Grant, and Marcello Benedetti. "Structure optimization for parameterized quantum circuits". In: *Quantum* 5 (Jan. 2021), p. 391. DOI: 10.22331/q-2021-01-28-391. URL: https://doi.org/10.22331%2Fq-2021-01-28-391.

[28] William Huggins et al. "Towards quantum machine learning with tensor networks". In: *Quantum Science and Technology* 4.2 (Jan. 2019), p. 024001. DOI: 10.1088/2058-9565/aaea94. URL: https://doi.org/10.1088%2F2058-9565%2Faaea94.

[29] Román Orús. "A practical introduction to tensor networks: Matrix product states and projected entangled pair states". In: *Annals of Physics* 349 (Oct. 2014), pp. 117–158. DOI: 10.1016/j.aop.2014.06.013. URL: https://doi.org/10.1016%2Fj.aop.2014.06.013.

[30] Guillaume Verdon et al. *Quantum Graph Neural Networks*. 2019. DOI: 10.48550/ARXIV.1909.12264. URL: https://arxiv.org/abs/1909.12264.

Ευρετήριο

NATIONAL TECHNICAL UNIVERSITY OF ATHENS

SCHOOL OF ELECTRICAL
AND COMPUTER ENGINEERING

COMPUTER SCIENCE DIVISION

Quantum Computer Simulator

with emphasis on Quantum Optimization
and Quantum Neural Networks

SENIOR THESIS

Pagonis A. Alexandros

Supervisor: Aris T. Pagourtzis
Professor at N.T.U.A.

Athens, June 2022

NATIONAL TECHNICAL UNIVERSITY OF ATHENS

SCHOOL OF ELECTRICAL
AND COMPUTER ENGINEERING

COMPUTER SCIENCE DIVISION

Quantum Computer Simulator

with emphasis on Quantum Optimization
and Quantum Neural Networks

SENIOR THESIS

Pagonis A. Alexandros

.....
Aris Pagourtzis
Professor at N.T.U.A.

.....
Dimitris Fotakis
Professor at N.T.U.A.

.....
Mesaritakis Charis
Professor at UAegean

.....

Pagonis A. Alexandros

Electrical and Computer Engineer

Quantum Circuit Simulator

Abstract

Quantum Computers are in an early stage of development and we cannot execute meaningful algorithms on them. That is why, in reality we work with Quantum Circuit Simulators.

In this thesis we construct a Quantum Circuit Simulator. This Simulator is written in the C++ programming language, it is complete and it aims for performance.

Its basic Design principles are: user-friendliness (a syntax similar to Pennylane's was used), compatibility (the simulator only uses the basic C++ library), easy and efficient circuit optimization, flexibility and open-source.

In its core there is a library for complex numbers and linear algebra; on top of this core, there are algorithms needed for the circuit simulation and its efficient optimization. At last, many tools are provided to the user, giving him/her the ability to conduct experiments and optimize his/her circuits.

Keywords: Quantum Computing, Simulator, C++, Quantum Circuit, Optimization, Qiskit, PennyLane, Qubit

Acknowledgements

I would like to thank Ioannis Theodonis, who introduced me into Quantum Computing and helped me in all my progress. Moreover, I would like to thank my brother Konstantinos Pagonis for his help in the illustrations and his advices. Finally, I would like to thank my supervisor, professor Aris Pagourtzis, who trusted me in such a difficult endeavour (both in the subject choice and in its execution).

Contents

List of Figures

1 Quantum Physics

1.1 Introduction

1.1.1 Physics in the 19th century

At the end of the 19th century, Physics had made great progress. Classical mechanics had a powerful mathematical formalism and electromagnetism had an elegant, and seemingly, complete description from the Maxwell equations. Another branch of Physics, though, Thermodynamics, used a new mathematical tool: statistics. That was "the bridge" to the microcosm. And indeed, soon enough our understanding of the microcosm was proven wrong.

1.1.2 The necessity for a Quantum Theory

The two great physical theories of the 19th century, electromagnetism and thermodynamics, gave explanations on the, until then, most deceitful phenomena: light and warmth. In spite of their great success, crisis was about to come: The theories predicted that a body in thermal equilibrium (with its surrounding) should radiate limitless amounts of energy in high frequencies, which doesn't happen in reality.

Figure 1: Radiation due to warmth

1.1.3 Ultraviolet Catastrophe

The title **"Ultraviolet Catastrophe"** shows how serious was the crisis for these theories. The false acceptance was that matter on the smallest scale could vibrate over all the spectrum of frequencies. Trying to solve this problem of disagreement between theory and experiment, physicist Max Planck suggested a discrete spectrum of frequencies and his mathematical result matched with the measurements. That first idea, that energy is transferred in small packets ("quanta"), introduced us to Quantum Mechanics.

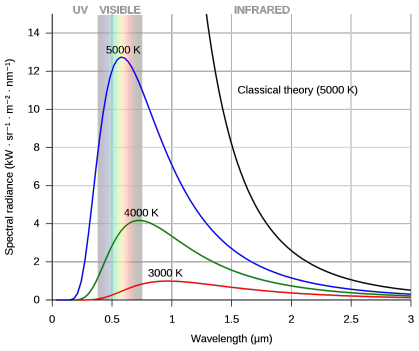


Figure 2: Ultraviolet Catastrophe

Of course, Physics made great progress from this point. Soon, Einstein's work on the photoelectric effect followed and in only 30 years Quantum Mechanics showed that the microcosm does not follow the "laws" of our everyday experience.

1.2 Causality

1.2.1 "God does not play dice"

The new experiments showed that in the microscopic world determinism doesn't hold (or holds in part). The first equation that accurately described particles (**Schrödinger**'s equation), relied upon that. Basic physical quantities, like the particle's position, are not fully defined, and not due to inadequacy of performing an accurate measurement, but due to the **innate probabilistic nature** of the particles. This interpretation conflicts with our everyday experience, since this probabilistic behaviour does not reflect in the big scale, hence it is not observed without tools.

1.2.2 Equation of Motion - Schrödinger Equation

$$i\hbar\frac{\partial\psi}{\partial t} = \left(-\frac{\hbar^2}{2m}\nabla^2 + V(\vec{r})\right)\psi \quad (1)$$

In classical mechanics all quantities depend on the body's position. Knowing its position $\vec{r}(t)$ every moment, we calculate its velocity $\vec{u}(t)$, its momentum, its net force, energies etc. In Quantum Mechanics, the first equation of motion (that Schrödinger [1] devised and bears his name), does not pin-point the particle's exact location. The solution ψ of the equation, also known as **Wavefunction**, needs a non-obvious interpretation. The interpretation given seemed (and still seems) entirely arbitrary, yet "reality" verifies it.

1.2.3 Wavefunction interpretation

$|\psi|^2 = \psi^*\psi(\vec{r}, t)$ is a Probability Distribution Function of finding the particle in the position $\vec{r}$ at time t. That means that the particle follows a distribution and, when measured, it will appear at the volume V with probability:

$$\iiint_V (\psi^*\psi)dV. \quad (2)$$

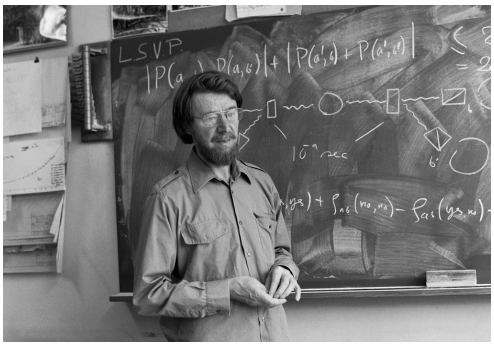


Figure 3: John Bell

Einstein himself didn't accept the statistical interpretation of nature. Distinctive is his phrase "God does not play dice". He -as any sceptical person would do- believed that the randomness in the measurements was not due to a real/innate randomness, but due to some mechanism that acts unbeknownst to us and generates those different results. The principle that the foundation of reality (as we experience it) behaves randomly seems completely irrational. One of the greatest achievements in Quantum Mechanics is that of **John Bell**'s work, who in 1964 proposed

an ingenious experiment in which it could be proven if the randomness was innate or not. And the experiments showed that Einstein was wrong.

1.3 Dirac's Notation

1.3.1 Superposition of States

If two functions ψ_1 and ψ_2 are solutions of the Schrödinger equation, then every linear combination $\alpha\psi_1 + \beta\psi_2$ is also a solution. This linear property reminds us of a Vector (or Linear) Space. Indeed, it would be very useful if we had a state space, where quantum state-functions are represented by vectors. Thus, our understanding of a system would be more organised and practical. For a vector space we need to define an inner product between vectors - that is: a measure of similarity between quantum states - functions.

1.3.2 Normalization Principle

Since $|\psi|^2$ is a Probability Distribution Function of the particle's position it should give 1 as a result when expanded in the whole space (De facto the particle will appear somewhere). That restriction is named **Normalization Principle**:

$$\boxed{\iiint_{\mathbb{R}^3} |\psi|^2 dV = 1} \tag{3}$$

Normally then, we define the **Inner Product** as:

$$\boxed{(\phi, \psi) = \iiint_{\mathbb{R}^3} \phi^* \psi \quad dV} \tag{4}$$

That operation defines an inner product, since it fulfils the prerequisites:

- $(\phi, \psi)^* = (\psi, \phi)$
- $(\phi, \lambda_1\psi_1 + \lambda_2\psi_2) = \lambda_1(\phi, \psi_1) + \lambda_2(\phi, \psi_2)$
- $(\psi, \psi) \geq 0$ και $(\psi, \psi) = 0 \iff \psi = 0$

1.3.3 Dirac's Notation

For a start, we define an orthonormal $\{\psi_i\}$ basis in the state-space with $(\psi_i, \psi_j) = \delta_{ij}$. Then a state will be defined as:

$$\psi = \sum_i c_i \psi_i \tag{5}$$

or in linear Algebra's language:

$$\psi = \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix} \tag{6}$$

It's time we present Dirac's notation for Quantum Mechanics:

- States are represented as: $|\psi\rangle \equiv \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix}$
- We define the helping, transpose and conjugate vector: $\langle\psi| \equiv |\psi\rangle^\dagger \equiv \begin{pmatrix} c_1^* & \cdots & c_n^* \end{pmatrix}$
- Inner product is defined as: $\langle\psi|\phi\rangle \equiv \langle\psi|\,|\phi\rangle$

1.4 Observables - Operators - Measurements

A function maps values to other values. An operator does that to functions. Eg: the differentiation operator acts on all differentiable functions and creates other functions: $\hat{D}_x f(x) = \frac{d}{dx}\{f(x)\} = f'(x)$. We will focus on linear operators, that is operators with the properties:

$$\begin{aligned} \hat{A}(\lambda_1 |\psi_1\rangle + \lambda_2 |\psi_2\rangle) &= \lambda_1 \hat{A} |\psi_1\rangle + \lambda_2 \hat{A} |\psi_2\rangle \\ (\lambda_1 \langle\phi_1| + \lambda_2 \langle\phi_2|)\hat{A} &= \lambda_1 \langle\phi_1| \hat{A} + \lambda_2 \langle\phi_2| \hat{A} \end{aligned} \tag{7}$$

As it seems, in the compose expression $\langle\phi| \hat{A} |\phi\rangle$, parentheses are not needed, since $(\langle\phi| \hat{A})\,|\phi\rangle \equiv \langle\phi|\,(\hat{A}\,|\phi\rangle)$. States $|\psi\rangle$ are inaccessible to us. Upon measurement $|\psi\rangle$ collapses to a single value. Before measurement, we cannot predict the result; only its expected value. To extract the average (or expected) value for a physical quantity we use its corresponding operator $\hat{A}$ to compute the expression $\boxed{\langle\psi| \hat{A} |\psi\rangle}$. For these operators to correspond to physical quantities, they must have real eigenvalues, that is to be Hermitian: [1]

$$\hat{A} = \hat{A}^\dagger \tag{8}$$

[1] **Hermitian ή Self-Conjugate** are the matrices that satisfy: $\hat{A}_{nm} = \hat{A}^*_{mn}$. Equivalently: the operators which satisfy: $\langle\psi|\hat{A}\phi\rangle \equiv \langle\hat{A}\psi|\phi\rangle\,, \quad \forall\phi,\psi$

2 Quantum Computing

2.1 Introduction - Qubit

Mutatis Mutandis we define a basis in our State-Space (known as the "Computational Basis"): the states $|0\rangle$ and $|1\rangle$. These states represent an orthonormal 2D **basis** and correspond to the usual vectors:

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{9}$$

A Quantum-Bit (**Qubit**) [2] is defined as a linear combination of basis vectors:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}, \quad \alpha, \beta \in \mathbb{C} \tag{10}$$

2.2 Superposition - Bloch's Sphere

2.2.1 Normalization

The meaning is the same with that of Quantum Mechanics: If we have a Qubit $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$, then from the **normalization** principle (since upon measurement one of the two states will appear):

$$\begin{aligned} 1 &= \langle\psi|\psi\rangle \implies \\ 1 &= (\alpha^*\,\langle 0| + \beta^*\,\langle 1|)(\alpha^*\,|0\rangle + \beta^*\,|1\rangle) \implies \\ 1 &= |\alpha|^2\,\langle 0|0\rangle + |\beta|^2\,\langle 1|1\rangle + \alpha^*\beta\,\langle 0|1\rangle + \alpha\beta^*\,\langle 1|0\rangle \overset{\langle\psi_i|\psi_j\rangle=\delta_{ij}}{\implies} \\ 1 &= |\alpha|^2 + |\beta|^2 \end{aligned} \tag{11}$$

2.2.2 Superposition

So, a Qubit can be "between" the two states $|0\rangle$ and $|1\rangle$, but when measured only one of them will appear. That property is called **"Superposition"** and gives us the advantage over classical computing. Its good results will be revealed later, when combined with the other basic Quantum property: Entanglement.

2.2.3 Bloch's Sphere

If we use the polar representation of complex numbers, a one-Qubit state can be written as:

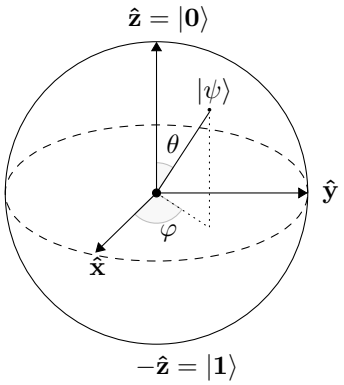


Figure 4: Bloch's Sphere

$$|\psi\rangle = \alpha e^{i\chi}\,|0\rangle + \beta e^{i\phi}\,|1\rangle$$

From the normalization principle we get: $\boxed{\beta = +\sqrt{1-\alpha^2}}$. Thus, we can remove a variable. Actually, we cannot know the quantum states, since they are unapproachable; and our best prediction is the expected value $\langle A \rangle = \langle \psi | \hat{A} | \psi \rangle$ of operators acting on them:

$$\langle A \rangle \overset{(A_{ij}=A_{ji}^*)}{==} \alpha^2 A_{00} + \beta^2 A_{11} + 2\alpha\beta Re(A_{10} e^{i(\chi-\phi)})$$

We note that this expected value depends on the difference $(\chi - \phi)$, so we can remove another variable. We arrived at a much more practical, but equivalent, representation:

$$|\psi\rangle = \cancel{e^{i\delta}} \left(\cos(\frac{\theta}{2}) |0\rangle + e^{i\phi} \sin(\frac{\theta}{2}) |1\rangle \right)$$

where the δ angle (known as global phase) can be deleted, since it will never affect the probabilities of a Qubit.

So, States-Vectors have a measure of 1 and 2 degrees of freedom. That's why we can represent them geometrically as points on the shell of a unitary Sphere (**Bloch's Sphere**). By convention, we place basis states $|0\rangle$ and $|1\rangle$ on the sphere's North and South pole respectively, imitating the spin measurements. To change this quantum state, means to move this point on another place of the shell via rotations around the 3 (or other) axes.

Angles θ and ϕ are the polar angle and azimuth on the sphere, respectively.

2.3 Entanglement - Bell States

2.3.1 Two Qubits - Tensor Product

Until now, we described the State-Space for a single Qubit. But what happens when we have 2 Qubits? Initially, we must expand our space to describe a larger number of Qubits. Suppose we have two Qubits, in the states:

$$\begin{aligned} |\psi_1\rangle &= \alpha |0\rangle + \beta |1\rangle \\ |\psi_2\rangle &= \gamma |0\rangle + \delta |1\rangle \end{aligned} \tag{12}$$

The desired space will have dimensions: $4 = 2 \times 2$ and the state describing the **whole** system, will be computed by the **tensor product** between the two vectors:

$$|\psi\rangle = |\psi_1\rangle \otimes |\psi_2\rangle \tag{13}$$

which by definition is:

$$\begin{pmatrix} \alpha \\ \beta \end{pmatrix} \otimes \begin{pmatrix} \gamma \\ \delta \end{pmatrix} \equiv \begin{pmatrix} \alpha \begin{pmatrix} \gamma \\ \delta \end{pmatrix} \\ \beta \begin{pmatrix} \gamma \\ \delta \end{pmatrix} \end{pmatrix} = \begin{pmatrix} \alpha\gamma \\ \alpha\delta \\ \beta\gamma \\ \beta\delta \end{pmatrix} \tag{14}$$

For example: if the first Qubit is in state $|0\rangle$ and the second in state $|1\rangle$, then the whole system is in the state:

$$|0\rangle \otimes |1\rangle = \begin{pmatrix}1\\0\end{pmatrix} \otimes \begin{pmatrix}0\\1\end{pmatrix} = \begin{pmatrix}0\\1\\0\\0\end{pmatrix} \tag{15}$$

As basis vectors, we obviously define the tensor products of the basis vectors:

$$\begin{aligned} |00\rangle = |0\rangle \otimes |0\rangle = \begin{pmatrix}1\\0\\0\\0\end{pmatrix} \qquad |01\rangle = |0\rangle \otimes |1\rangle = \begin{pmatrix}0\\1\\0\\0\end{pmatrix} \\ |10\rangle = |1\rangle \otimes |0\rangle = \begin{pmatrix}0\\0\\1\\0\end{pmatrix} \qquad |11\rangle = |1\rangle \otimes |1\rangle = \begin{pmatrix}0\\0\\0\\1\end{pmatrix} \end{aligned} \tag{16}$$

This idea easily expands to more than two Qubits (since the tensor product is an associative operation).

2.4 Entanglement

Let's examine tensor product from the reverse perspective. Suppose we have the two Qubit state:

$$\frac{|00\rangle + |11\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}} \begin{pmatrix}1\\0\\0\\1\end{pmatrix} \tag{17}$$

Trying to decompose this state into two one-Qubit sub-states, we discover that such an induction is impossible:

$$\nexists \alpha, \beta, \gamma, \delta \in \mathbb{C} : \qquad \frac{1}{\sqrt{2}} \begin{pmatrix}1\\0\\0\\1\end{pmatrix} = \begin{pmatrix}\alpha \times \gamma\\ \alpha \times \delta\\ \beta \times \gamma\\ \beta \times \delta\end{pmatrix} \tag{18}$$

That is, the Qubits are "connected" and cannot be disconnected. This example is equivalent to a system of two coins in which, upon tossing we get two heads or two tails, but no other combination. That means that the Qubits are "interwoven" and by processing one we change the whole system. That property allows us to have a form of parallel computing on the same "hardware".

2.4.1 Multiple Qubits

This idea easily expands to multiple Qubits. Tensor product for n 2D-vectors is (by definition):

$$\begin{pmatrix}\alpha_1\\ \beta_1\end{pmatrix}\otimes\begin{pmatrix}\alpha_2\\ \beta_2\end{pmatrix}\otimes\cdots\otimes\begin{pmatrix}\alpha_n\\ \beta_n\end{pmatrix}=\begin{pmatrix}\alpha_1\alpha_2\cdots\alpha_{n-1}\alpha_n\\ \alpha_1\alpha_2\cdots\alpha_{n-1}\beta_n\\ \alpha_1\alpha_2\cdots\beta_{n-1}\alpha_n\\ \vdots\\ \beta_1\beta_3\cdots\beta_{n-1}\alpha_n\\ \beta_1\beta_3\cdots\beta_{n-1}\beta_n\end{pmatrix} \tag{19}$$

or in Dirac's notation:

$$|\Psi\rangle=\bigotimes_{i=1}^{n}|\psi_i\rangle \tag{20}$$

We define a basis $\{|b_i\rangle\}$ accordingly:

$$\begin{aligned}|0\cdots0\rangle=|0\rangle\otimes|0\rangle\otimes\cdots\otimes|0\rangle=\begin{pmatrix}1\\0\\\vdots\\0\end{pmatrix}\qquad |0\cdots1\rangle=|0\rangle\otimes|0\rangle\otimes\cdots\otimes|1\rangle=\begin{pmatrix}0\\1\\\vdots\\0\end{pmatrix}\\ \vdots\\ |1\cdots0\rangle=|1\rangle\otimes|1\rangle\otimes\cdots\otimes|0\rangle=\begin{pmatrix}0\\\vdots\\1\\0\end{pmatrix}\qquad |1\cdots1\rangle=|1\rangle\otimes|1\rangle\otimes\cdots\otimes|1\rangle=\begin{pmatrix}0\\\vdots\\0\\1\end{pmatrix}\end{aligned} \tag{21}$$

and a quantum state can be written as a linear combination of basis states:

$$|\Psi\rangle=\sum_{i=0}^{2^n-1}c_i|b_i\rangle\qquad \mu\varepsilon\sum_{i=0}^{2^n-1}|c_i|^2=1 \tag{22}$$

2.5 Circuit - Gates

2.5.1 Principles

Quantum Circuits, like electrical circuits, is a way of representing Quantum Computations with "wires", gates and measurements. However, there is a difference. No feedback is allowed, that is no output can be used to the same circuit as input: the circuit only "moves forward".

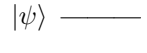


Figure 5: Qubit as a «wire»

2.5.2 One-Qubit Gates

Basic one-Qubit gates are rotation gates (around the 3 main axes on Bloch's Sphere) and Pauli gates:

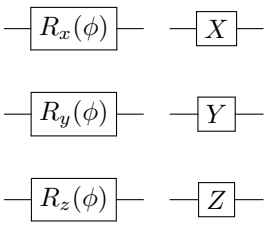


Figure 6: Basic Gates

$$R_x(\theta) = \begin{pmatrix} \cos(\theta/2) & -i\sin(\theta/2) \\ -i\sin(\theta/2) & \cos(\theta/2) \end{pmatrix} \qquad X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$
$$R_y(\theta) = \begin{pmatrix} \cos(\theta/2) & -\sin(\theta/2) \\ \sin(\theta/2) & \cos(\theta/2) \end{pmatrix} \qquad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$
$$R_z(\theta) = \begin{pmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{pmatrix} \qquad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

Also a very common gate is Hadamard gate (or gate of Equal Superposition):

Figure 7: Hadamard Gate

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

2.5.3 Many-Qubit Gates

The simplest many-Qubit gate is the CNOT Gate (Controlled NOT) and acts on 2 Qubits. If the first Qubit ("Control Qubit") is $|1\rangle$, then on the other qubit ("Target Qubit") an X gate will be applied (which corresponds to the logical negation operator since $X|0\rangle = |1\rangle$, $X|1\rangle = |0\rangle$):

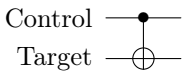


Figure 8: CNOT

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

From the matrix we see that if the first Qubit is Zero, then the second won't change:

$$|00\rangle \longrightarrow |00\rangle\,, \qquad |01\rangle \longrightarrow |01\rangle$$

Whereas, if the first Qubit if One, the second will flip:

$$|10\rangle \longrightarrow |11\rangle\,, \qquad |11\rangle \longrightarrow |10\rangle$$

With the base gates and the CNOT gate we can make any complex gate.

2.6 Examples - Simple Circuits

2.6.1 Swap

Simple circuit that swaps two Qubits using CNOT gates. Extremely useful, since all Qubits are not adjacent in real Quantum Computers.

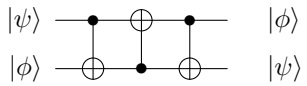
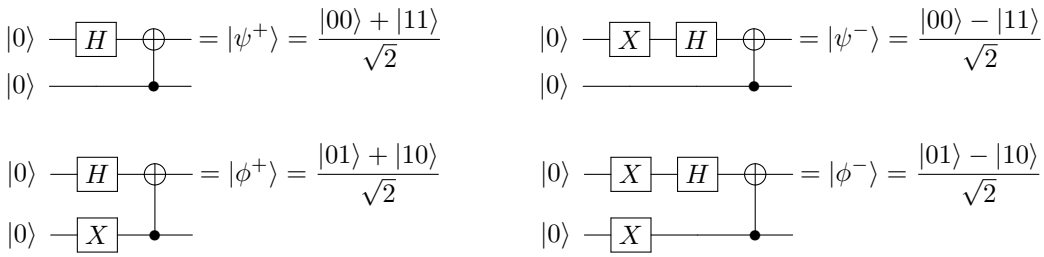


Figure 9: SWAP

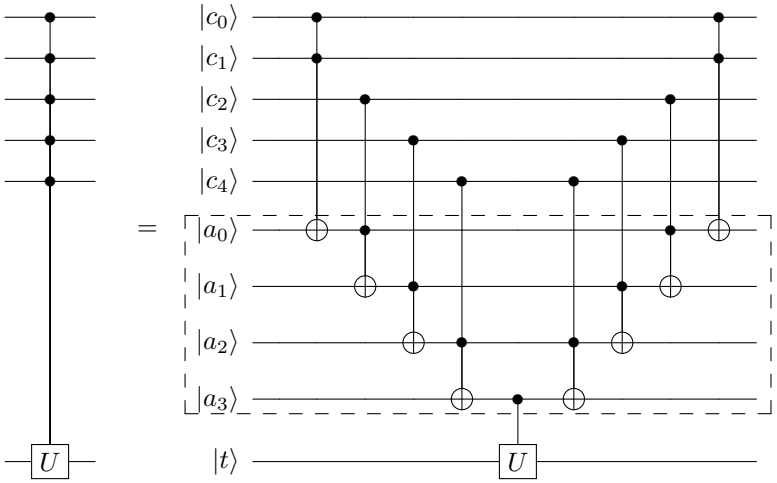
2.6.2 Bell States

Circuits constructing the maximum entanglement states $|\psi^+\rangle, |\psi^-\rangle, |\phi^+\rangle, |\phi^-\rangle$, known as Bell states:



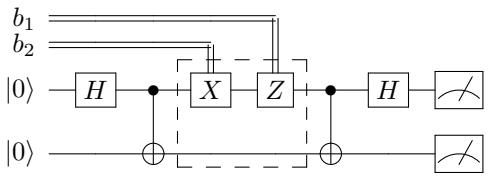
2.6.3 Multiple Controlled Gate

Circuit that constructs multiple controlled gate using ancillary Qubits.



2.6.4 Superdense Coding

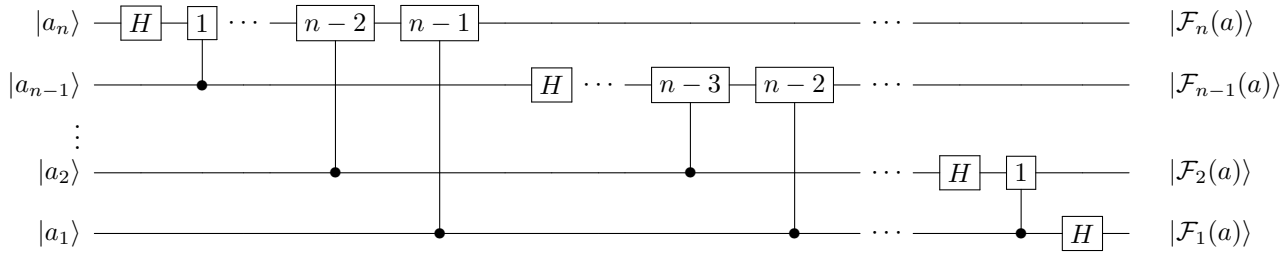
Simple circuit [3] that shows the advantage of Quantum Computing over Classical Computing. With X and Z gates, we can transfer two bits of information by encoding it into a single Qubit.



2.6.5 Quantum Fourier Transform

Circuit that computes the Fourier Transform [4] in the computational basis $|0\rangle, |1\rangle, \cdots, |n-1\rangle$:

$$Q\hat{F}T\,|x\rangle = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} e^{i\frac{2\pi xk}{n}}\,|k\rangle$$



Where the gate $\boxed{k}$ corresponds to the Phase Gate $P(\frac{\pi}{2^k}) = \begin{pmatrix} 1 & 0 \\ 0 & e^{\frac{i\pi}{2^k}} \end{pmatrix}$.

The Quantum Fourier Transform for a vector of n elements can be computed with $\Theta(n^2)$ operations, while the classical FT with $\Theta(n2^n)$. The Quantum FT is exponentially faster. Actually, it is so fast that it allows us to implement many algorithms efficiently, even simple addition.

2.6.6 Quantum Fourier Adder

We can construct a **Binary Adder** [5], where the addition happens in phase space. Specifically, if we have two operands A and B into qubit groups $|A\rangle$, $|B\rangle$, then addition can be done in phase space with this circuit:

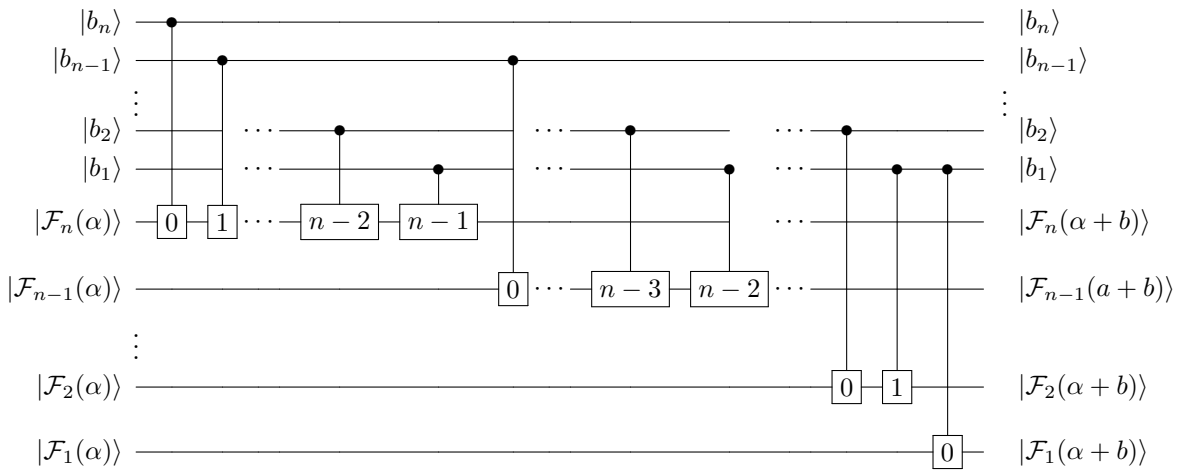


Figure 10: Addition in Phase Space

With an Inverse Fourier Transform we fetch the result.

2.6.7 Phase KickBack

Every Quantum gate is being applied in a normalized state and it creates another normalized state. That means that the eigenvalue equation will have the form: $\hat{U}|\psi\rangle = \lambda|\psi\rangle$, where: $\lambda^*\lambda\langle\psi|\psi\rangle = 1 \implies |\lambda|^2 = 1 \implies \boxed{\lambda = e^{2\pi i\phi}}$.

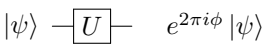


Figure 11: Global Phase

We can extract this global phase $e^{2\pi i\phi}$ that a gate adds, having its controlled version ($C\hat{U} = \hat{U} \oplus \hat{I}$).

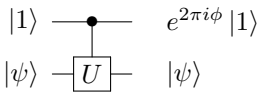


Figure 12: Global Phase KickBack

Indeed:

$$CU\left(|1\rangle|\psi\rangle\right) = \left(e^{2\pi i\phi}|1\rangle\right)|\psi\rangle$$

However, it's still a global phase, so it can't be measured. There is a trick with which we can convert this global phase into a local one.

2.6.8 Quantum Phase Estimation (QPE)

Combining the two previous ideas we can "read" the phase being applied to a state by a gate. The phase $0 \leq \phi \leq 1$ covers the angle range $\{0, 360^o\}$. In the binary system such a number is represented with t bits as $0, \phi_0\phi_1...\phi_t$ and precision in the order of $\frac{1}{2^t}$. Quantum Fourier Transform transforms the state $|x\rangle$ from the

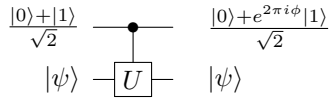


Figure 13: Phase KickBack

computational basis to the state $Q\hat{F}T\,|x\rangle = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} e^{i\frac{2\pi xk}{n}} |k\rangle$. We want the opposite: to "print" ϕ's binary representation as a base state. So we fetch it on phase space with controlled U gates and with Inverse QFT we print it as $|\phi_0\phi_1...\phi_t\rangle$. [2]

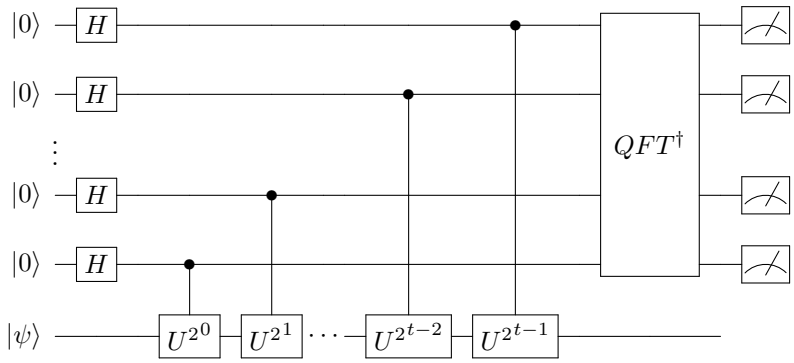


Figure 14: Quantum Phase Estimation

2.6.9 Order Finding Algorithm

In a modulo N arithmetic we define as **order** of a number α the smallest positive number r (greater than zero), for which it holds:

$$\alpha^r \equiv 1 \pmod{N}$$

Finding the order is a difficult process. However, the function $f(r) = \alpha^r \pmod{N}$ is periodic and actually with a period of r. Thus, instead of calculating the order, we need to find the period of this periodic $f(r)$ function. A mathematical tool for finding periods is Fourier Transform. So we apply the Quantum Phase Estimation on the gate $U_x\,|y\rangle = |xy \pmod{N}\rangle$ and with some post-processing we fetch the order r very fast.

2.6.10 Shor's Algorithm

Shor's algorithm [6] is used for factorizing integers. Given a number N we ask for its prime factors. The algorithm is this:

1. We choose a random number α in the range $\{2...N-1\}$.
2. Unless we randomly found a factor, it holds: $gcd(\alpha, N) = 1$
3. We find the order r: $\alpha^r \equiv 1 \pmod{N}$
4. Check 1) If r is odd, return to step 1
5. Check 2) If $\alpha^{\frac{r}{2}} \equiv -1 \pmod{N}$, return to step 1

[2] A gates U rises to a power k, when it's repeated k times

6. If we passed the checks, we found 2 factors of N: $p, q = gcd(\alpha^{\frac{r}{2}} \pm 1, N)$

The probability of passing the two checks is higher than 50%. The difficult step is finding the order; but in a quantum computer it can be done very efficiently, since the Quantum Fourier Transform is **exponentially** faster.

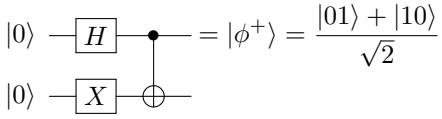
Shor's algorithm is considered classic and it's one of the first big algorithms that showed the clear and applicable advantage of quantum computing. All our cryptography depends on the difficulty of factorizing numbers.

2.6.11 Density Operator

Density Operator (or Matrix) is the compact representation of a Quantum State as a sum of **projection** operators:

$$\rho = \sum_j p_j \left|\psi_j\right\rangle \left\langle\psi_j\right|$$

For example, for the maximum entangled Bell state:



we have

$$\rho = \left(\frac{1}{\sqrt{2}}\begin{bmatrix}1\\0\\0\\1\end{bmatrix}\right)\left(\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 0 & 0 & 1\end{bmatrix}\right) = \frac{1}{2}\begin{bmatrix}1 & 0 & 0 & 1\\0 & 0 & 0 & 0\\0 & 0 & 0 & 0\\1 & 0 & 0 & 1\end{bmatrix}$$

It is clear that the state is not pure, but mixed, since it has non-zero diagonal elements.

2.6.12 Grover's Algorithm

Grover's algorithm [7] is a quantum algorithm, which is used to search the single element in a function's $f(x)$ domain, which gives different value from all the rest. If there is no order in the function's domain, we would need $O(N)$ queries for N elements; however, with the quantum algorithm we need $\Omega(\sqrt{N})$. This algorithm shows the indisputable advantage that quantum computers have in searching a much bigger space.

In specific, if we have a function $f : \{0, 1, .., N-1\} \to \{0, 1\}$, with only one value ω for which $f(x = \omega) = 1$. Our goal is to find ω. We use the quantum operator:

$$\begin{cases} U_\omega \left|x\right\rangle = -\left|x\right\rangle, & \text{for } x = \omega, \text{ that is } f(x) = 1 \\ U_\omega \left|x\right\rangle = \left|x\right\rangle, & \text{for } x \neq \omega, \text{ that is } f(x) = 0 \end{cases}$$

which acts upon $n = \lceil log_2 N \rceil$ Qubits. It can also be written as $U_\omega |x\rangle = (-1)^{f(x)} |x\rangle$

Algorithm 1 Grover's Algorithm

INPUT: An oracle U_ω
RESULT: ω

$|s\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle$
for iterations **do**
  Apply U_ω
  $U_s = 2 |s\rangle \langle s| - I$
end for

After $r(N) \leq \lceil \frac{\pi}{4} \sqrt{N} \rceil$ repetitions we have ω.

The operator $U_s = 2 |s\rangle \langle s| - I$, is called diffusion operator and it increases the probability of state $|\omega\rangle$, after every query from the oracle U_ω.

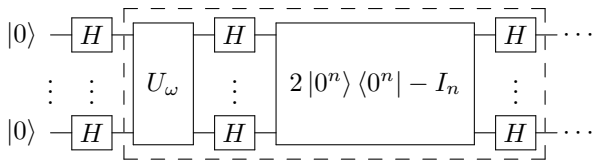


Figure 15: Grover's Algorithm

2.6.13 Deutsch-Jozsa Algorithm

Yet another algorithm that shows the advantage of quantum computing is the Detsch-Jozsa algorithm [2]. Given a function $f : \{0,1\}^n \to \{0,1\}$ which is either constant or balanced (it returns 0 in half its domain and 1 in its other half), we can deduce its type with only one query. We construct the operator U_f which acts upon a quantum register x and a single Qubit y and maps $|x\rangle |y\rangle \to |x\rangle |y \oplus f(x)\rangle$, where $\oplus$ is addition modulo 2.

The algorithm is this:

Algorithm 2 Deutsch–Jozsa Algorithm

INPUT: An oracle U_f

RESULT: Type of f: constant / balanced

$|\psi_0\rangle = |0\rangle^{\otimes n} |1\rangle$ ▷ Prepare

$|\psi_1\rangle = \frac{1}{\sqrt{2^{n+1}}} \sum_{x=0}^{2^n-1} |x\rangle (|0\rangle - |1\rangle)$ ▷ Broadcast H

$|\psi_2\rangle = \frac{1}{\sqrt{2^{n+1}}} \sum_{x=0}^{2^n-1} (-1)^{f(x)} |x\rangle (|0\rangle - |1\rangle)$ ▷ Apply U

$|\psi_3\rangle = \frac{1}{2^n} \sum_{y=0}^{2^n-1} \left[\sum_{x=0}^{2^n-1} (-1)^{f(x)} (-1)^{x*y} \right] |y\rangle$ ▷ Broadcast H to 1st register

$P(|0\rangle^{\otimes n}) = \left| \frac{1}{2^n} \sum_{x=0}^{2^n-1} (-1)^{f(x)} \right|^2 = 0, 1$ ▷ Measure

where $x * y = x_0 y_0 \oplus x_1 y_1 \oplus ... \oplus x_{n-1} y_{n-1}$ is the bit-wise sum of products. Upon measurement, if the register is full of zeros then f is constant; otherwise f is balanced.

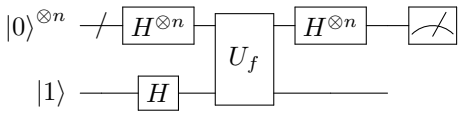
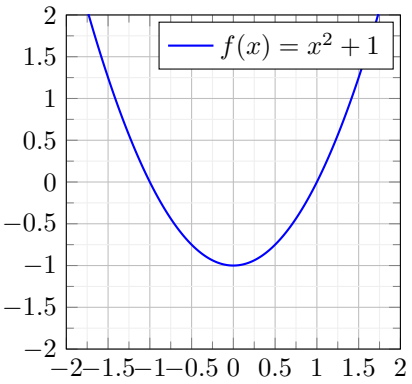


Figure 16: Deutsch-Jozsa Algorithm

3 Optimization

3.1 Goal

Optimization is the branch of mathematics that cares about finding the optimal solution to a specific problem. The most common and important problem that appears in the real world is the **minimization** of a function. Some times the solution can be found analytically. For example, if we seek function's $f(x) = x^2 - 1$ minimum, we can use the derivative $f'(x) = 2x$ to compute the gradient changing points $f'(x) = 0 \implies x = 0$ and then compute the minimum point $f(x_0) = -1$ in position $x_0 = 0$.



3.2 Optimizers

3.2.1 Introduction

In general, an analytic computation of the gradient is not always possible; and even if we have a closed form for the derivative it is difficult to solve the equation $f'(x) = 0$. Thus, we need an algorithm with which to approximate a function's minimum having the ability to compute the function's **derivative** in every point. This algorithm is known as Gradient Descent:

Algorithm 3 Gradient Descent Prototype

INPUT: A random value x, Number of iterations N
OUTPUT: Minimum of f(x)

for i in $(1, N)$ **do**
  $x \leftarrow x - f'(x)$
end for

Since the gradient $f'(x)$ points towards the direction in which the function decreases, we move in this direction.

3.2.2 Gradient Descent

There are two problems with this approach: Firstly, in the area around the random point x we chose, f may have a local, not global, minimum. And secondly, the gradient may be too big, slowing down the convergence. That's why we introduce the parameter η which is called **Learning Rate** and it usually has a small value $\eta \in (0.01, 0.1)$ making a smaller step each iteration. Hence, the complete algorithm is written as:

Algorithm 4 Gradient Descent

INPUT: A random value x, Number of iterations N
OUTPUT: Minimum of f(x)

$\eta = 0.01$
for i in $(1, N)$ **do**
    $x \leftarrow x - \eta * f'(x)$
end for

3.2.3 Gradient Descent Optimizer with momentum

A very small learning rate assures that we don't branch off away from the minimum, but it slows the convergence a lot. A solution to that is to begin with a small learning rate and increase it gradually. This variable that adjusts the learning rate is called **momentum**. A single optimization algorithm with momentum is the following:

Algorithm 5 Gradient Descent with Momentum

INPUT: A random value x, Number of iterations N
OUTPUT: Minimum of f(x)

$\eta = 0.01$
$m = 0$
$\alpha = 0$
for i in $(1, N)$ **do**
    $\alpha \leftarrow m\alpha + \eta f'(x)$
    $x \leftarrow x - \eta * f'(x)$
end for

Computations get harder when our functions which we want to minimize are multi-variable functions. But the idea remains the same. The only difference is that instead of a derivative, we use the gradient $f(x)$ in our algorithms.

3.2.4 Adam Optimizer

Probably the most potent optimizer for general use is the Adam Optimizer [8]. And it is a variation of Gradient Descent with two momenta.

Algorithm 6 Adam Optimizer

INPUT: A random initialization x, Number of iterations N
OUTPUT: Minimum of f(x)

$\eta = 0.01$
$\beta_1 = 0.9$
$\beta_2 = 0.99$
$\epsilon = 1e - 08$
$a, b = 0$
for i in $(1, N)$ **do**
    Prepare
    $a \leftarrow \beta_1 a + (1 - \beta_1)\nabla f(x)$
    $b \leftarrow \beta_2 b + (1 - \beta_2)(\nabla f(x))^{\odot 2}$
    $\eta \leftarrow \eta \frac{\sqrt{(1-\beta_2)}}{(1-\beta_1)}$

    Update
    $x \leftarrow x - \eta \frac{a}{\sqrt{b}+\epsilon}$
end for

3.3 Stochasticity

However good an optimizer is, there is always the possibility that it will be trapped in a local minimum. A solution to this problem is to use another vector g instead of the gradient for our parameter updates. This new vector behaves more arbitrarily, allowing us to escape local minima, while it maintains the gradient's statistic behaviour; it is chosen from a random distribution, such that is expected value is the real gradient: $\mathbb{E}(g) = \nabla f$.

4 Neural Networks

4.1 Neuron

A **neuron** is defined as an information processing unit. Each neuron has some inputs and an output. The simplest and most obvious choice of output is a **linear combination** of the inputs. If we denote the inputs as x_i and the weights of the linear combination as w_i, then our output will be:

$$y = \sum_{i=1}^{m} w_i x_i$$

It is common to include a bias, that is a constant unit together with the linear combination. In linear algebra's notation:

$$y = \begin{bmatrix} b & w_1 & \cdots & w_m \end{bmatrix} \cdot \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_m \end{bmatrix}$$

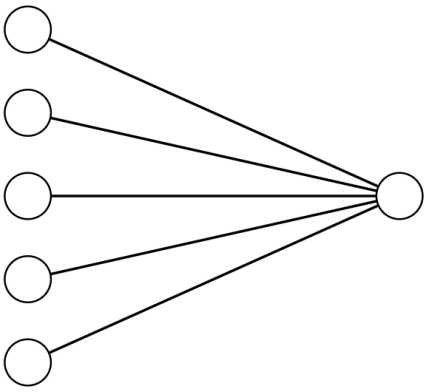


Figure 17: Neuron

4.2 Neural Network

The next logical step is to connect many neurons in a layer.

$$\vec{y} = \begin{bmatrix} b_1 & w_{11} & \cdots & w_{1m} \\ b_2 & w_{21} & \cdots & w_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_n & w_{n1} & \cdots & w_{nm} \end{bmatrix} \cdot \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_m \end{bmatrix} = \begin{bmatrix} b_1 + \sum_{i=1}^{m} w_{1i} x_i \\ b_2 + \sum_{i=1}^{m} w_{2i} x_i \\ \vdots \\ b_n + \sum_{i=1}^{m} w_{ni} x_i \end{bmatrix}$$

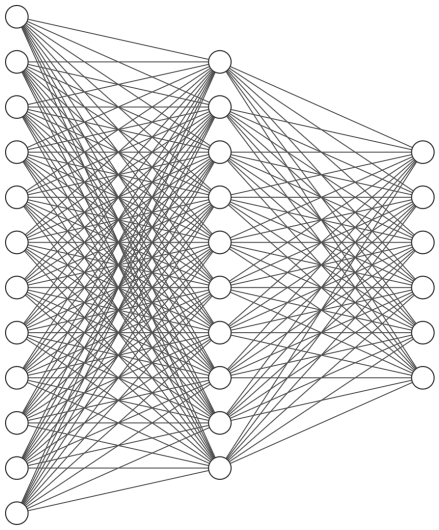
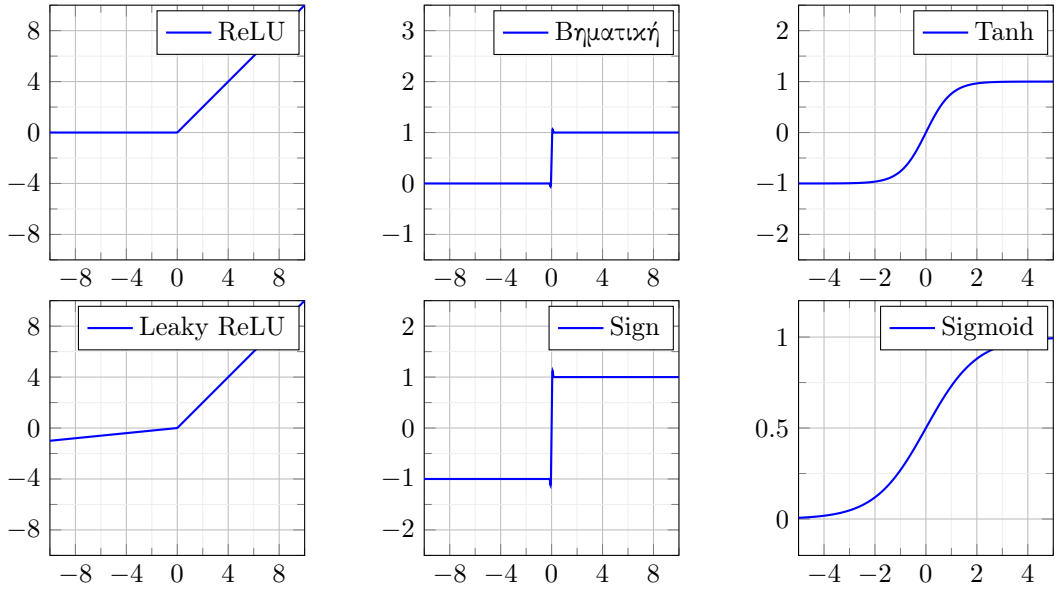


Figure 18: Neural Network

We always prefer linear operations; however, they do not suffice. Indeed, placing a second layer of linear combinations next to the first we won't have won any physical meaning; and that's because linear combinations of linear combinations are, again, linear combinations. So, we would have double the variables for the same result.

Unfortunately, we must apply non-linear functions to the output of our neurons. This process makes a neuron "useful", and that is why it is called **"activation function"**. Some of the most

common functions are:



Usually, a function with easy-to-use derivative and bounded output is used, like sigmoid or hyperbolic tangent.

4.3 Linear Classifier - Perceptron

A simple, yet powerful, template is perceptron. It consists of a single neuron with inputs $\{x_i\}$ and weights $\{w_i\}$, plus a bias. The classification is the sign of the output computation:

$$u = b + \sum_{i=1}^{m} w_i x_i$$
$$y = \text{sign}(u) = \pm 1$$

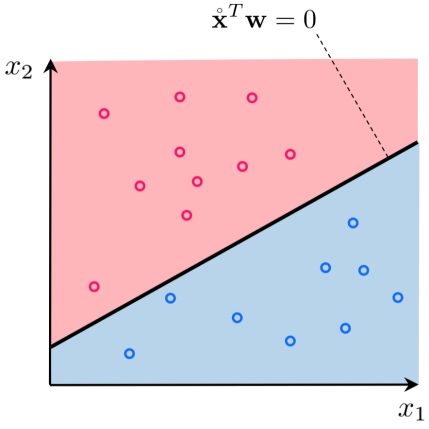


Figure 19: Perceptron

Perceptron is used for data **classification** into two categories. Its purpose is to achieve a correct classification and for this to be done, our data must be linearly separable. Using many perceptrons we can classify data into many categories.

4.4 Support Vector Machine (SVM)

The SVM template is a special case of the Perceptron classifier. Its purpose is not only to classify correctly data into two categories, but to achieve the strictest possible separation between them, maximizing the minimum distance between the points and the separation line.

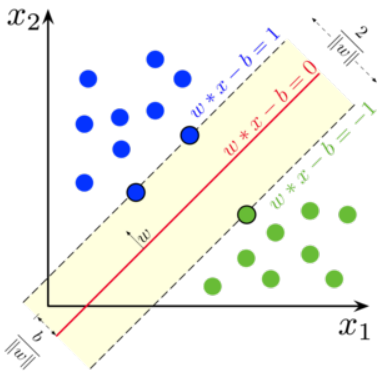


Figure 20: SVM

4.5 Non-linearly separable data

In the general case, our data may not be linearly separable, so we artificially increase the dimensions of our samples until they are linearly separable. This transformation is called Kernel Trick. With a function $\phi(x)$ we map our data to a bigger space, where they are linearly separable. However, there is not a single function that works for all samples, so every time we must find a new transformation.

4.6 Convolutional Neural Network

An extremely powerful neural network template for **image** analysis is Convolutional Neural Network [9]. Before the image goes as the networks input, it is pre-processed and some characteristic features are extracted. These features are subjected to further processing until they are fed into the neural network for classification. The neural network exploits the 2-Dimensional form of the data (usually pictures) and the locality that exists in the input (neighbouring pixels are expected to be more or less the same) and it drastically reduces the data dimensionality before inserting it into the network, improving a lot the processing speed.

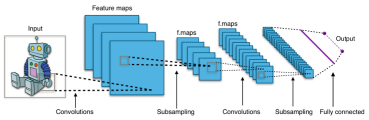


Figure 21: CNN

4.7 Generative Adversarial Network

Another neural network template are Generative Adversarial Networks [10]. These neural networks consist of two structures: a generator and a discriminator. They are trained from the same input dataset (eg pictures of cats), but the generator trains to produce fake data imitating the training data, while the discriminator trains to distinguish real data from the fake ones that the generator provides. The generator's purpose is to deceive the discriminator that its output fake data is real, while the discriminator's purpose is

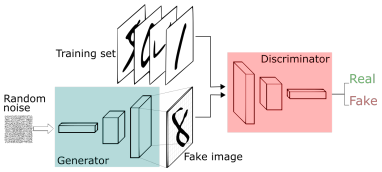


Figure 22: GAN

to not be deceived (that is, to be able to classify data as real or fake). Thus, they have a somewhat common loss function:

$$\max_{D} V(D) = \mathbb{E}_{x \sim p_{data}(x)}[logD(x)] + \mathbb{E}_{z \sim p_z(z)}[log(1 - D(G(z)))]$$

$$\min_{G} V(G) = \mathbb{E}_{z \sim p_z(z)}[log(1 - D(G(z)))] \quad \text{or} \quad \mathbb{E}_{z \sim p_z(z)} - [logD(G(z))]$$

It seems from the equations that D tries to classify correctly the data that it takes from the real distribution x (first term) and the fake data constructed by the generator G (second term). Respectively, the generator tries to minimize the probability that the discriminator will confirm his data as counterfeit.

It is one of the most impressive kinds of neural networks, it achieves extraordinarily realistic results and it relies on a simple and elegant idea.

4.8 Neural Network Training

To train a neural network, meaning to compute the optimized parameters with respect to a given objective function, we need its gradient in every point. The gradient's computation with arithmetic methods, eg explicitly with the derivative definition, is very expensive. An analytic computation of the gradient of each neural network is practically impossible, since we have many non-linear functions nested.

There exists an efficient algorithm for the computation of the gradient and its called Back-Propagation. With this algorithm, we compute the output of the network executing it forwards. Then, we traverse it backwards, computing every element of ∇f, the moment we encounter the corresponding node.

5 Quantum Circuit Optimization

Quantum Computers are particularly good for searching solutions into **spaces with many dimensions**; for this reason, they seam ideal for Neural Networks and optimization in general. They have already shown their greatness in problems of Quantum Chemistry (eg [11]).

5.1 Purely Quantum Optimization

One method is to introduce the problem's data into the circuit and leave it to do all the optimizations by itself (with one execution of the circuit and without supervision).

5.1.1 Adiabatic Computing

A similar computation strategy (not with a circuit) is **Adiabatic Computing** [12]. In Quantum Mechanics the operator that corresponds to the energy of the system is called Hamiltonian and is denoted with an $\hat{H}$. The **Quantum Adiabatic Theorem** assures us that a physical system will adapt to its environment, if it is driven by small changes. Indeed, if we can construct the Hamiltonian $\hat{H} = (1-t)\hat{H}_i + t\hat{H}_f, t \in (0,1)$, then as time t increases the Hamiltonian will move from state $\hat{H}_i$ to state $\hat{H}_f$. The only thing we have to do is to code the objective function into the operator $\hat{H}_f$ and wait for the system to converge by itself. Company D-Wave is working with Adiabatic Quantum Computing.

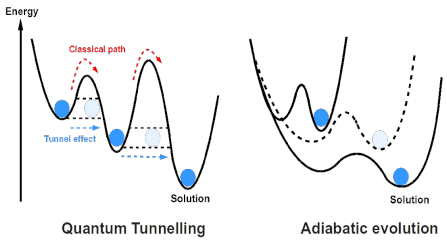


Figure 23: Adiabatic Computing

5.1.2 QAOA

Another purely quantum optimization method, implemented with a circuit, is the algorithm Quantum Approximate Optimization Algorithm (QAOA) [13]:

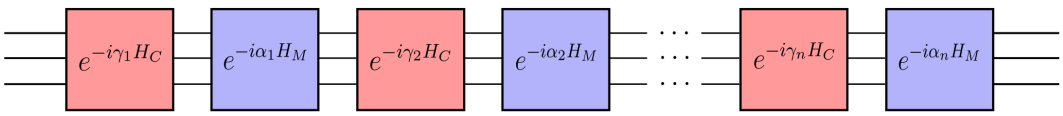


Figure 24: QAOA Algorithm

Algorithm 7 QAOA

INPUT: f(z) , β_i, γ_i
GOAL: $z \approx \underset{z\in\{0,1\}^n}{argmin} f(z)$

Compute $\hat{C}$ such that: $\hat{C}\,|z\rangle = f(z)\,|z\rangle$ ▷ Cost Hamiltonian
Compute: $B = \bigotimes_n X$ ▷ Mixer Hamiltonian
Construct: $U(C,\gamma) = e^{-i\gamma C}$ ▷ Cost Gate
Construct: $U(B,\beta) = e^{-i\beta B}$ ▷ Mixer Gate

$|\psi\rangle = |+\rangle^{\otimes n}$ ▷ Broadcast Hadamard
for i in {0,p} **do**
  $|\psi\rangle \leftarrow U(B,\beta_i)U(C,\gamma_i)\,|\psi\rangle$
end for

Measure $|\psi\rangle$

Initially, we construct an equal superposition of all possible states and then we use two gates: the first "computes" the cost that we want to minimize and the second mixes between states. With the correct choice of parameters β and γ we achieve convergence.
For combinatorial optimization problems, the algorithm QAOA is efficient enough and very easy to implement: eg Max-Cut [14], Knapsack [15].

5.2 Hybrid Optimization Algorithms

In contrast with the other methods, we can classically train a quantum circuit.

5.2.1 Quantum SVM (QSVM)

Imitating the classical Support Vector Machines, we can define Quantum Support Vector Machines (QSVM) [16]. The idea is exactly the same, but we take advantage of the exponentially bigger space in which quantum states have access.

5.2.2 Quantum Convolutional Neural Network (QNN)

With the same thinking we can construct a Quantum Convolutional Neural Network (Quonvolutional NN) for image analysis [17]. Example.

5.2.3 Quantum GAN

We can also construct Quantum Adversarial Neural Networks, with Quantum Generator [18] and classical Discriminator. In addition, for better control, less noise (and simulation capability) many small generators are used which when combined produce the whole image.

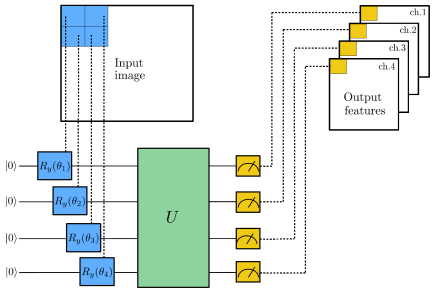


Figure 25: QNN

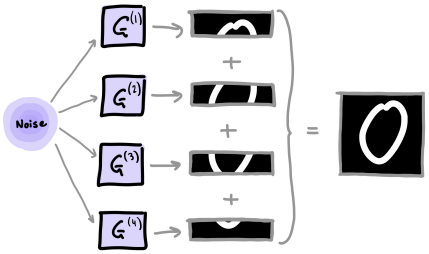


Figure 26: QGAN

6 Simulator

6.1 Design Principles

In contrast with the well-known simulators (eg Qiskit [19], Cirq, PennyLane [20]) which are written mainly for educational purposes and with emphasis on user's ease, this simulator has great **performance** as its purpose; for that reason, ideal programming language is C++, since it has the speed of C, but some very useful built-in data structures (eg vector, map).

Design Principles of the Simulator are:

- **Simplicity**: It provides the basic Quantum Gates and basic functionalities, such as: experiment execution, StateVector simulation, probabilities computation, computation of expectation value of operator and it is very easy to use.
- **Compatibility**: It does not depend on any library other than the basic C++ library.
- **Speed**: All data structures and functionalities are made from scratch aiming for performance.
- **Flexibility**: Groupings of data into small classes are used, so that they are organized autonomously into small structures; this allows us to change the implementation of many functionalities or data structures, without having to modify the rest of the code. Also, all code files inherit some basic constant parameters (such as the precision of our numbers) from the file "parameters.h", making it very simple to built a custom library by changing only one line.
- **Open Source**: The source code is available for every use.

6.2 Mathematics tools

This implementation does not use any mathematics library. The mathematics tools that are needed for quantum circuits are:

- Complex Numbers
- Square Matrices and specifically matrices with dimensions powers of 2 (we can exploit this for many optimizations)
- Vectors (or Column-Matrices)
- Operations with Matrices and Vectors. In particular: Tensor product, Matrix-Vector Product, Inner product of two vectors, Construction of Transpose-Conjugate Matrix
- Printing Matrices and Vectors

6.2.1 Complex Numbers

For complex numbers we use the datatype **c_type** defined in the file "parameters.h" and it is either a double or a float, depending to the desired precision for our calculations. In addition, we overload the usual operators for mixing real numbers with complex ones (eg comp a = 3.0*z;)

6.2.2 Square Matrices

Since matrices are not known in compilation time (so they cannot be static), we must allocate memory in run-time. If we implement matrices as list of lists (eg std::vector< std::vector<T> > Array) we would need many calls for **memory allocation**. Instead we will use an **one-dimensional** array, which we will treat as two-dimensional.

Specifically, we define the class **SqMatrix** for square matrices ($M_{n\times n}$), which consists of an one-dimensional array of n^2 elements and a value n for knowing the dimensions of the matrix (without having to recalculate them each time). Upon this implementation, we build a simple interface for square matrices with the methods **get** and **set**, with which we read from, and write to, arrays. Thus, we avoid multiple calls for memory allocation and we substitute double dereferences with single ones, which further improves the performance of matrices, which are the core of our calculations.

That is, for a matrix of dimensions $A_{n\times n}$ we have a list of $K[n^2]$ elements and the element $a[i][j]$ is in reality the element $K[i * n + j]$.

6.3 Gate - Gate Layer

A gate is actually a square matrix, plus a list of qubits in which this array acts on. So naturally, we define a class called Application which contains the matrix, the aforementioned list, along with the derivative of the matrix if that gate is parametric (we need the derivative for circuit optimization).

Gates that act on sets of qubits which are disjoint, can be applied at the same time in a circuit. They can be combined into a big gate which is computed as the tensor product of them:

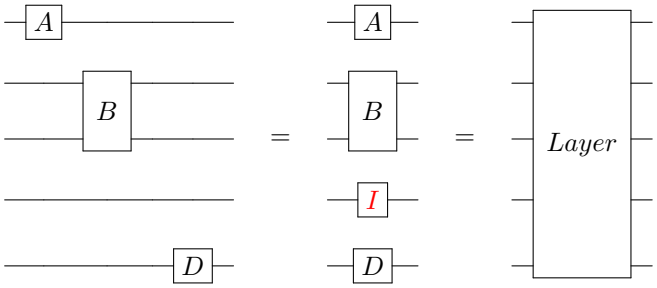


Figure 27: Layer of Gates as a Gate

$$Layer = D \otimes I \otimes B \otimes A$$

Although, the layer-gate has $2^n \times 2^n = 2^{2n}$ elements, while small matrices (if they are single-qubit gates) have all together $2 \times 2 \times n = 4n$ elements. For this reason we compute the big array only at run-time and not at insertion.

6.4 Circuit

The Circuit Class models quantum circuits. We apply an arbitrary gate with the app method, while most common gates are built-in and can be applied with their corresponding methods: eg RX(2, 0.672) for the gate RotationX(0.672) on Qubit 2.

The user can construct the circuit and then:

- Execute an **experiment** (measurement) with the measure method.
- Calculate the **probabilities** for each basis state with the method print_probabilities.
- Print the **StateVector** with the print_state_vector method.
- Print the **expectation value** $\langle E \rangle$ for an operator with the method exp_value.
- Calculate the **gradient** $\nabla \langle E \rangle$ in respect to the circuit parameters for every parametric gate (in the order that the user gave them) with the exp_value_grad method.

6.4.1 Layer Compression

Every gate that acts on any subset of the qubits, affects the whole StateVector. So every time we need to construct the whole layer-gate, filling it with identity matrices for the qubits where there is no real gate. That process is too computationally expensive, especially when we have many small gates. We choose to **compress** as much as we can the circuit, such that it has the least possible number of layers.

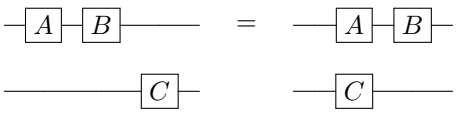


Figure 28: Layer Compression

We need to place gates as left as we can. To achieve this, we must know the first available position for every Qubit in the circuit. So, if a gate acts on many Qubits, its application will be in the first common available layer.

The algorithm for layer compression (in pseudolanguage) is the following:

Algorithm 8 Gate application

INPUT:
available: index of the first available layer for every qubit
layers: Circuit Layout. Stores gate applications layer-by-layer
Application a: {Gate: Square Matrix, qubits: [int] }
RESULT:
Puts gate in circuit

```
layer_idx = 0

for qubit in a->qubits do
    layer_idx = max(layer_idx, available[qubit])        ▷ Find first common index
end for

if len(layers) < layer_idx + 1 then                     ▷ Out of Circuit
    layers.append([a])                                  ▷ New Layer of applications
else
    layers[layer_idx].append(a)                         ▷ Push this application in existing layer
end if

for qubit in a->qubits do                               ▷ Update available positions for these qubits
    available[qubit] = layer_idx + 1
end for
```

6.4.2 Permutations

Having constructed the circuit, we only need execute it. To execute it, we begin with an empty StateVector $|0\rangle$ and we apply layer-gates one-by-one until we reach the end. However, there is a chance that some layers have gates which do not act on neighbouring qubits, and even if they do, they may have another ordering. For example:

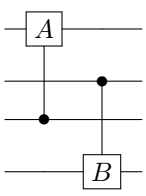


Figure 29: Shuffled Qubits

These two gates belong to the same layer (since the sets of qubits that they are applied unto are disjoint), even if we cannot draw them overlapping one-another. We cannot explicitly construct the big gate as tensor product of small gates, considering that the gates are not in order. If we had the gates in order, things would be fairly easy:

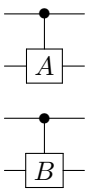


Figure 30: Gates in order

A naive solution would be to construct the big matrix corresponding to the layer, mapping its elements according to the qubit permutation. However, it would be more efficient to construct the matrix as if the gates were in order and then just **permute** the Qubits in the StateVector (which has many less elements).

We need an algorithm that computes the Qubit permutation for every layer according to the gates we encounter and also returns the number of Qubits that correspond to non-trivial gates, so that we can fill the rest with Identity Matrices.

Algorithm 9 Make Permutation

INPUT:
layer: A layer of Gate applications
qubits: The number of qubits in the circuit
RESULT:
Permutation and number of qubits affected in this layer

```
allocated = [false]*qubits                          ▷ Keep track of allocated qubits
gates = len(layer)                                  ▷ Gates in this layer
last_used_qubit = 0

for gate in range(gates) do                         ▷ Iterate over Gates
    gate_size = len(layer[gate].qubits)             ▷ Size of particular Gate

    for qubit in range(gate_size) do                ▷ permutation[fake_qubit] = real_qubit
        permutation[qubit+last_used_qubit] = layer[gate].qubits[qubit]
        allocated[layer[gate].qubit] = true         ▷ allocated[real_qubit] = true
    end for
last_qubit_used += gate_size
end for
qubits_used = last_used_qubit                       ▷ Keep a copy to return

for iterator in range(qubits) do                    ▷ Fill permutation with unallocated qubits
    if allocated[iterator] == false then
        permutation[last_used_qubit++] = iterator
    end if
end for

Return (permutation, qubits_used)
```

6.4.3 Application on Permutation

Having the Qubit permutation we need to make a big layer-matrix filling with Identity matrices for the unused Qubits:

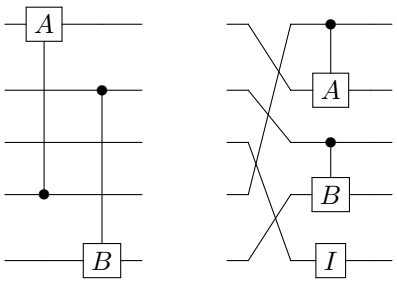


Figure 31: Before and After Permutations

Our final step is:

1. Make the new StateVector given the permutation
2. Apply the layer-gate on this mapped StateVector
3. Reverse the mapping to extract the real StateVector

Basically, we construct a similarity transformation $S = PDP^{-1}$, where matrix P^{-1} does the permutation, P does the reverse permutation and D applies the layer-gate:

Algorithm 10 Apply Layer Gate on permuted qubits

INPUT:
StateVector: The State-Vector
Permutation: Qubit permutation
qubits: The number of qubits in the circuit
Gate: Layer-Gate
RESULT:
The new State-Vector after this layer is applied

```
mapped_state_vector = StateVector
SV_size = 1 « qubits                                   ▷ State Vector size

for pos in range(len(StateVector)) do                  ▷ Iterate Basis States
    mapped_pos = 0                                     ▷ Find index of mapped State Vector
    for fake in range(qubits) do
        real = permutation[fake]
        real_bit = (pos » real) & 1
        mapped_pos |= (real_bit « fake)                ▷ replace fake qubit with real one
    end for
    mapped_state_vector[mapped_pos] = StateVector[pos]
end for

mapped_state_vector = gate * mapped_state_vector       ▷ Matrix Mul.

for pos in range(len(StateVector)) do                  ▷ Undo mapping
    mapped_pos = 0
    for fake in range(qubits) do
        real = permutation[fake]
        real_bit = (pos » real) & 1
        mapped_pos |= (real_bit « fake)
    end for
    StateVector[pos] = mapped_state_vector[mapped_pos] ▷ Reverse
end for
```

6.5 Measurements

After Circuit execution, having the StateVector, the only thing left is to measure our results.

6.5.1 StateVector

Since it is a simulation, the user can calculate the exact StateVector; this cannot be done with real quantum circuits. The simulator offers this functionality with the print_state_vector method.

6.5.2 Probabilities

The user can also measure the probabilities of each basis state with the print_probabilities method. Probabilities are easily extracted for the StateVector: by taking the square of the complex's absolute value.

6.5.3 Random Experiment

Even if the simulator is mainly a StateVector simulator, it can also simulate an experiment as if it was a real Quantum Computer. To achieve this we must make a probability distribution function for our basis states. Using the <random> library of C++ we can construct such a distribution. We use the random number generator **Mersenne Twister** for random numbers and the std::discrete_distribution class for simulating a discrete distribution. The results are stored in a dictionary using C++'s std::map data structure.

The user can simulate a measurement with the measure(shots) method, giving the number of measurements he/she wants to take place.

6.5.4 Obervables

Apart from simple measurements in the basis states, the user may want to measure the expectation value for an observable given the corresponding Hamiltonian. This functionality is also provided with the exp_value(Observable, qubits) method.

6.6 Differentiable Programming

One of the simulator's operations is to return the gradient for the expected value $\nabla\langle E\rangle$ of an operator with respect to the circuit parameters. For example, in this circuit:

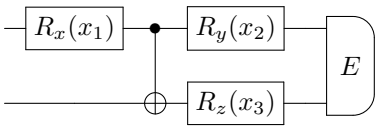


Figure 32: Parametric Circuit

we measure the value of E into the computational basis; given many measurements this value's average will converge towards the expected value $\langle E\rangle$, which can be calculated with analytic, yet expensive, calculations.

6.6.1 Circuit Optimization

In Quantum Optimization we are asked to minimize the expectation value of a Hamiltonian operator, and the most common way is the Gradient Descent algorithm (or some variation of it):

$$x^{(t+1)} \leftarrow x^{(t)} - \eta * \nabla\langle E(x^{(t)})\rangle$$

The computationally difficult step is the gradient computation $\nabla\langle E\rangle$ for a parametric circuit. The obvious, but naive, way is to use the derivative definition and execute the circuit $O(p)$ times,

where p is the number of parameters:

$$\nabla\langle E\rangle_i = \frac{\partial\langle E\rangle}{\partial x_i} = \lim_{\epsilon\to 0}\frac{\langle E\rangle_{x_i+\epsilon} - \langle E\rangle}{\epsilon}$$

That solution works, but is very expensive, because we need to execute the circuit once per parameter.

6.6.2 Parameter Shift Rule

An arithmetic calculation of gradient using the derivative definition is not ideal, since it is based on the precision of the ϵ value. Instead, we usually use the Parameter Shift Rule[21], [22]; for most gates (eg Pauli Gates) the general rule is:

$$\nabla_{\theta_i}\langle\hat{E}\rangle(\boldsymbol{\theta}) = \frac{1}{2}\left[\langle\hat{E}\rangle\left(\boldsymbol{\theta} + \frac{\pi}{2}\hat{\mathbf{e}}_i\right) - \langle\hat{E}\rangle\left(\boldsymbol{\theta} - \frac{\pi}{2}\hat{\mathbf{e}}_i\right)\right].$$

This way, the best precision is guaranteed.

6.6.3 Adjoint Differentiation

As in classical neural networks, we need an algorithm for efficient gradient calculation; something similar to the BackPropagation algorithm.

In quantum circuits, there is such an efficient algorithm and is called **Adjoint Differentiation** [23]. In reality, the only thing we do with quantum circuits is to:

1. Begin from the initial state $|0\rangle$
2. Execute the circuit and get $|\Psi\rangle = U_n U_{n-1} ... U_0 |0\rangle$
3. Measure an observable $\langle M\rangle = \langle\Psi|M|\Psi\rangle$

However, these gates-matrices U_i are all unitary, so it holds: $U^\dagger U|\phi\rangle = |\phi\rangle$

Rewriting the expectation value as:

$$\langle M\rangle = \langle b|k\rangle = \langle\Psi|M|\Psi\rangle$$

where

$$\langle b| = \langle\Psi|M = \langle 0|U_1^\dagger \dots U_n^\dagger M$$

$$|k\rangle = |\Psi\rangle = U_n U_{n-1} \dots U_1|0\rangle$$

we notice that vectors $|b\rangle$ and $|k\rangle$ could have been chosen differently:

$$\langle b_n| = \langle 0|U_1^\dagger \dots U_n^\dagger M U_n$$

$$|k_n\rangle = U_{n-1} \dots U_1 |0\rangle$$

To change the splitting position in this expression, we remove one operator form the $|k\rangle$ vector and put it into the $\langle b|$:

$$\langle b_n| = \langle b| U_n$$

$$|k_n\rangle = U_n^\dagger |k\rangle$$

For the derivative:

$$\begin{aligned}\frac{\partial \langle M \rangle}{\partial \theta_i} &= \langle 0|U_1^\dagger \dots \frac{dU_i^\dagger}{d\theta_i} \dots M \dots U_i \dots U_1|0\rangle \\ &+ \langle 0|U_1^\dagger \dots U_i^\dagger \dots M \dots \frac{dU_i}{d\theta_i} \dots U_1|0\rangle \\ &= 2 \cdot \Re\left(\langle 0|U_1^\dagger \dots U_i^\dagger \dots M \dots \frac{dU_i}{d\theta_i} \dots U_1|0\rangle\right)\end{aligned}$$

Or using our ancillary vectors:

$$\frac{\partial \langle M \rangle}{\partial \theta_i} = 2\Re\left(\langle b_i| \frac{dU_i}{d\theta_i} |k_i\rangle\right)$$

with

$$\langle b_i| = \langle 0|U_1^\dagger \dots U_n^\dagger M U_n \dots U_{i+1}$$

$$|k_i\rangle = U_{i-1} \dots U_1 |0\rangle$$

When the moved operator is the derivative, we denote:

$$\langle \tilde{b}_i| = \langle b_i| \frac{dU_i}{d\theta_i}$$

and

$$\frac{\partial \langle M \rangle}{\partial \theta_i} = 2\Re\left(\langle \tilde{b}_i| k_i\rangle\right)$$

We only need to calculate:

$$|b_i\rangle = U_{i+1}^\dagger |b_{i+1}\rangle$$

$$|k_i\rangle = U_i^\dagger |k_{i+1}\rangle$$

recursively, starting from the end of the circuit (from the output StateVector).

The complete algorithm is:

Algorithm 11 Adjoint Differentiation

INPUT: The output StateVector of a Circuit, Every gate U_i of the circuit (in order of insertion)
OUTPUT: Vector $\nabla\langle E\rangle$

$|\lambda\rangle :=$ Circuit Output StateVector ▷ Execute Circuit
$|\phi\rangle := |\lambda\rangle$
$|\lambda\rangle \leftarrow \hat{E}\,|\lambda\rangle$
for $i \in \{P, ..., 1\}$ **do** ▷ Back-Propagate
  $|\phi\rangle \leftarrow \hat{U}_i^\dagger\,|\phi\rangle$
  $|\mu\rangle := |\phi\rangle$
  $|\mu\rangle \leftarrow (d\hat{U}_i/d\theta_i)\,|\mu\rangle$
  $\nabla\langle E\rangle_i = 2\Re\,\langle\lambda|\mu\rangle$
  if $i > 1$ **then**
    $|\lambda\rangle \leftarrow \hat{U}_i^\dagger\,|\lambda\rangle$
  end if
end for

The simulator supports this functionality with the exp_value_grad method, which returns the gradient of the circuit parameters over an operator. For this algorithm the circuit uses a dictionary with keys the serial number of a gate and values the gate's actual place in the circuit. This dictionary is implemented with C++'s std::map.

6.6.4 Optimizers

This simulator supports 3 optimizers at the moment. The simple Gradient Descent, the AdaGrad with changing learning rate and the well-known Adam [8].

6.7 Code Summary

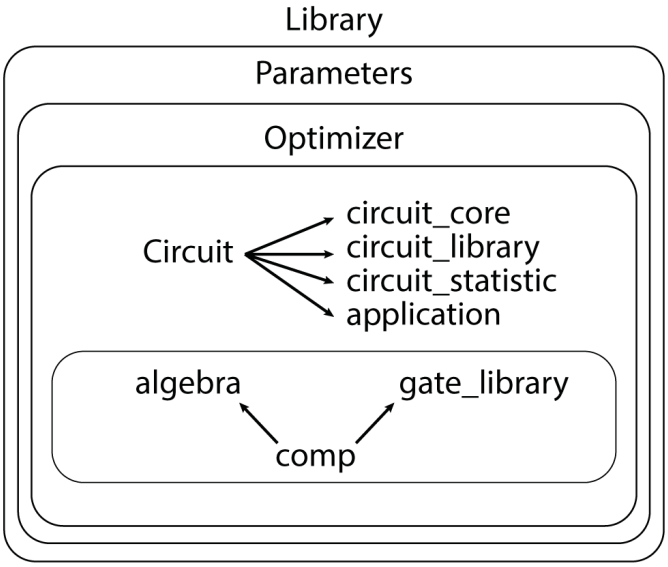


Figure 33: Simulator Layout

6.7.1 library

The file library.h includes all the source files and builds the library

6.7.2 parameters

In parameters.h file we define general parameters that all other files need, like the maximum number of Qubits, the precision in our calculations, the safety on our checks etc.

6.7.3 comp

The file comp.h is our complex number library. It defines the structure of complex numbers, it defines unary and binary operations.

6.7.4 algebra

The file algebra.h handles linear algebra operations. It defines the two-dimensional Square Matrix and on top of that it contains the mathematics tools we need: tensor product, inner product, expected value, transpose-adjoint matrix, inner product between matrix and vector.

6.7.5 gate_library

The file gate_library.h contains the definition of basic gates.

6.7.6 application

The file application.h defines the Application class which handles the application of a gate into some qubits.

6.7.7 circuit

The file circuit.h defines the circuit class. It is the basic class provided in the interface for the user.

6.7.8 circuit_core

The file circuit_core.h contains the algorithms needed for the circuit functionalities: gate insertion, permutations, layer application, circuit execution, expectation value calculation, gradient descent computation.

6.7.9 circuit_library

The file circuit_library.h contains the quantum gate library that the simulator uses by default; it is based upon the matrix definitions in the file gate_library.h

6.7.10 circuit_statistics

The file circuit_statistics contains measurement tools provided the circuit is already executed and the result StateVector computed.

6.7.11 optimizers

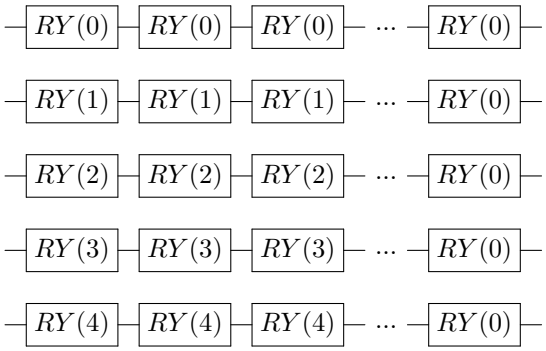
The file optimizers.h defines quantum circuit optimizers

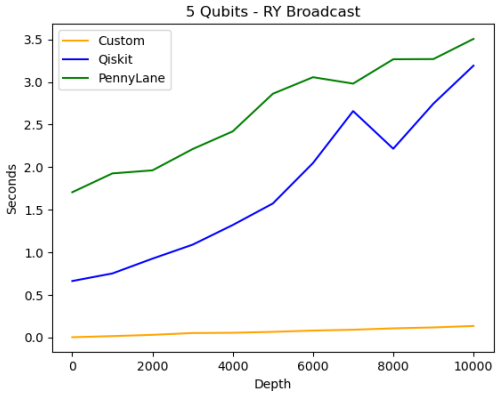
7 Measurements

For our measurements, we compare 3 benchmarks between our simulator, IBM's Qiskit (python) and PennyLane's lightning.qubit (C++).

7.1 5 Qubits - RY rotation Gates

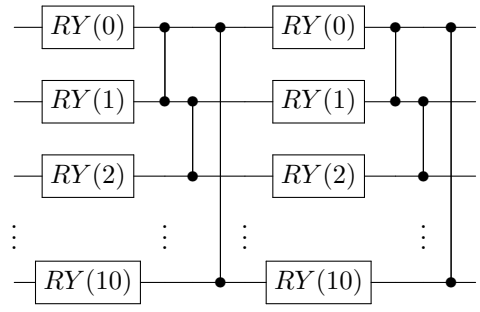
A 5-Qubit Circuit filled with RY rotation Gates and depth given from the user. We chose parametric RY single-Qubit gates, because they are the most computationally expensive.

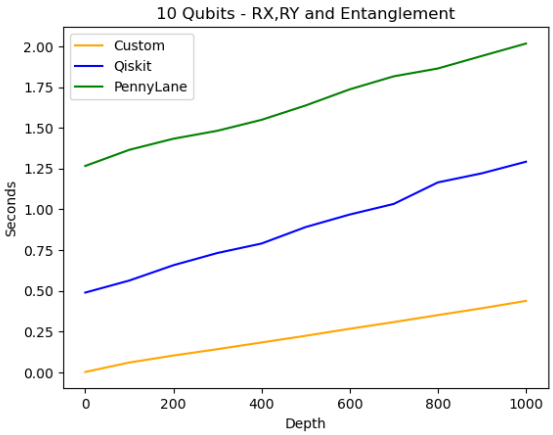




7.2 10 Qubits - RY, RX Gates and Entanglement

A 10-Qubit Circuit with parametric RY layers and entangling CZ layers. It is a neural network template, really difficult to simulate (since it is much entangled)

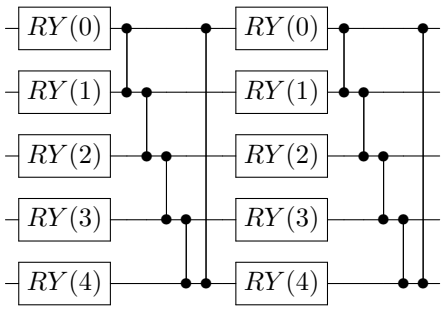


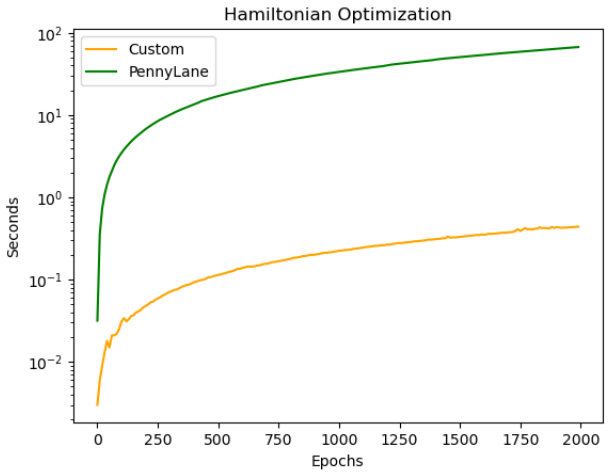


7.3 5 Qubits - Hamiltonian Optimization

A circuit like the one of the second test with 5 Qubits, but this time we optimize it according to the H_2 molecular Hamiltonian. Our goal is, to find a circuit that produces a state $|\psi\rangle$, such that the expectation value $\langle\psi| H_2 |\psi\rangle$ is minimized. We artificially create a non-linear transformation,

entangling our 4 desired Qubits with another ancillary one.





8 Conclusion

Our Simulator is much more faster than commercial ones, as expected (just by the language selection). However, we note that in circuit optimization our Simulator is 3 orders of magnitude faster than PennyLane's C++ Backend (lightning.qubit); that allows us to optimize much bigger circuits at this time.

9 Future Work

The simulator is theoretically complete, since it can simulate any possible quantum circuit. However, we can still add functionalities and improve its efficiency.

9.1 Libraries

On top of the simulator's core we can add libraries.

9.1.1 Optimization

One extension could be more optimizers:

1. Lie Algebra Optimizer [24] [25].Gradient Descent algorithms can be done directly in a Lie group, instead of the Euclidean Space. This method assures us convergence into the global minimum, but requires exponentially more parameters.
2. Quantum Natural Gradient Optimizer [26]. Optimizer with adaptive learning rate, via calculation of the diagonal or block-diagonal approximation to the Fubini-Study metric tensor. A quantum generalization of natural gradient descent.
3. RotoSelect Optimizer. A special case Optimizer for rotation gates [27].

9.1.2 Quantum Chemistry

Quantum Computers are especially good in quantum chemistry problems. A Quantum Chemistry library could be added like the one PennyLane has. Its basic functionalities would be:

1. Construction of Molecular Hamiltonians
2. Mappings between creation / annihilation operators and spins (eg Jordan-Wigner)
3. Hartree-Fock method

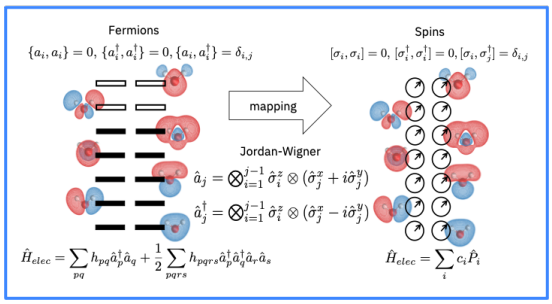


Figure 34: Jordan-Wigner Mapping

9.1.3 Quantum Optimization

A quantum optimization library would be useful. It's basic components would be:

1. Algorithms, like QAOA [13].

2. Neural Network Libraries (eg [28], [29], [30]).

3. Stochasticity for gradient computation.

9.2 Performance - Parallelism

We can always increase a program's performance. An easy improvement would be to introduce parallelism, either with multiple cores (OpenMP) or with an accelerator - graphics card (CUDA). Also, we could improve the basic algorithms of the circuit, like permutations or the tensor product, by using sparse arrays for layers with few gates.

10 Code

10.1 library

```
#include "parameters.h"

// MATHEMATICS CORE
#include "comp.h"
#include "algebra.h"


// ARRAY AND GATE HANDLERS
#include "gate_library.h"


// CIRCUIT
#include "circuit.h"
#include "circuit_core.h"
#include "circuit_statistics.h"
#include "circuit_library.h"


#include "optimizers.h"
```

10.2 parameters

```
// BUILDING PARAMETERS. CHANGE HERE AND ALL HEADER
// FILES WILL INHERIT THEM
#ifndef __PARAMETERS__
#define __PARAMETERS__ 420


// REMOVE THIS LINE TO EXCLUDE ALL CHECKS
#define __SAFE_LIBRARY__ 420
#define MAX_QUBITS 10


// COMPLEX NUMBER TYPE
typedef double c_type;
//typedef float c_type; // No difference really... keep double



#endif
```

10.3 comp

```
#ifndef __MY_COMPLEX_LIBRARY
#define __MY_COMPLEX_LIBRARY 420
#include <iostream> // cout
#include <math.h>         // sqrt
#include <stdlib.h>         // abs

#include "parameters.h"

// CORE LIBRARY FOR COMPLEX NUMBERS

// Constants
const c_type pi = 3.141592653589793238;


// CUSTOM COMPLEX NUMBER
typedef struct complex__{
        c_type real;
        c_type imag;
} comp;


// Complex Operations
```

```
inline comp         c                       (c_type real, c_type imag);
inline comp         conj                 (comp a);
inline comp         mul_conj          (comp a, comp b);
inline c_type         distance          (comp a, comp b);
inline void           print_comp          (comp a);


// Complex Number Constructor
inline comp c(c_type _real, c_type _imag) {
        return (comp) {_real, _imag};
}


inline comp conj (comp a) {
        return (comp) {a.real, -a.imag};
}

// COMPLEX CONJUGATE MULTIPLICATION a*b
inline comp mul_conj(comp a, comp b) {
        return (comp) {a.real*b.real + a.imag*b.imag,
                a.real*b.imag - a.imag*b.real};
}


// print as: a + ib
inline void print_comp (comp a) {
        if(a.real == 0.0 and a.imag == 0.0) {
                std::cout << "0  ";
                return;
        }

        if(a.real != 0.0)
                std::cout << a.real;

        if(a.imag != 0.0) {
                if(a.imag > 0.0) {
                        std::cout << "+i";
                        if(a.imag != 1.0)
                                std::cout <<  a.imag;
                }
                else {
                        std::cout << "-i";
                        if(a.imag != -1.0)
                                std::cout << -a.imag;
                }
        }

        std::cout << "  ";
}


// euclidean distance of two complex numbers
inline c_type distance (comp a, comp b) {
        // manhattan distance
        return abs(a.real - b.real) + abs(a.imag - b.imag);
}




// Complex Arithmetics

// COMPLEX OPERATOR COMPLEX

inline comp operator + (comp c1, comp c2) {
    return comp {c1.real + c2.real, c1.imag + c2.imag};
}
```

```
inline comp operator - (comp c1, comp c2) {
    return comp {c1.real - c2.real, c1.imag - c2.imag};
}

inline comp operator * (comp c1, comp c2) {
    return (comp) {c1.real*c2.real - c1.imag*c2.imag,
                c1.real*c2.imag + c1.imag*c2.real};
}

inline comp operator / (comp c1, comp c2) {
        c_type den = c2.real * c2.real + c2.imag * c2.imag;

    return (comp) {
            (c1.real * c2.real + c1.imag * c2.imag) / den,
                (c1.imag * c2.real - c1.real * c2.imag) / den
        };
}



// COMPLEX OPERATOR REAL & REAL OPERATOR COMPLEX

// position independant
inline comp operator + (comp c, c_type r) { return (comp) {r + c.real, c.imag}; }
inline comp operator + (c_type r, comp c) { return (comp) {r + c.real, c.imag}; }
inline comp operator * (comp c, c_type r) { return (comp) {r * c.real, r * c.imag}; }
inline comp operator * (c_type r, comp c) { return (comp) {r * c.real, r * c.imag}; }

// position dependant
inline comp operator - (comp c, c_type r) { return (comp) {c.real - r, c.imag}; }
inline comp operator - (c_type r, comp c) { return (comp) {r - c.real, -c.imag}; }
inline comp operator / (comp c, c_type r) { return (comp) {c.real / r, c.imag / r}; }
inline comp operator / (c_type r, comp c) {
        c_type den = c.real * c.real + c.imag * c.imag;
    return (comp) { (r * r + c.imag * c.imag) / den, (- r * c.imag) / den };
}



#endif
```

10.4 algebra

```
#ifndef __ALGEBRA__
#define __ALGEBRA__ 420

#include <vector>
#include "parameters.h"
#include "comp.h"



// Define Custom 2D Square Matrix
// Implement with 1D-flattened array
class SqMatrix {
        private:
                unsigned int _dims;         // rows-columns
                std::vector<comp> _elems;
        public:
                inline SqMatrix(unsigned int d);
                inline SqMatrix(std::vector<std::vector <comp>> e);

                inline comp get(unsigned int r, unsigned int c) const;
                inline void set(unsigned int r, unsigned int c, comp e);
                inline unsigned int dims() const { return _dims; }
};
```

```
// Init
inline SqMatrix::SqMatrix(unsigned int d) {
        _dims  = d;
        _elems = std::vector<comp> ( d*d );
}

// Init from vector of vector
inline SqMatrix::SqMatrix(std::vector<std::vector <comp>> e) {
        unsigned int d = e.size();

        #ifdef __SAFE_LIBRARY__
                for(unsigned int i = 0; i < d; ++i){
                        if(e[i].size() != d) {
                                std::cout << "Matrix Constructor Error: "
                                        << "2D vector is not square\n";
                                exit(1);
                        }
                }
        #endif

        _elems = std::vector<comp> (d*d);
        _dims  = d;
        for(unsigned int i = 0; i < d; ++i)
                for(unsigned int j = 0; j < d; ++j)
                        _elems[i * d + j] = e[i][j];
}


// GET HANDLER
inline comp SqMatrix::get(unsigned int r, unsigned int c) const {
        // you can ignore this. vector will crash anyway
        #ifdef __SAFE_LIBRARY__
                if(r > _dims-1){
                        std::cout << "SqMatrix Get_Element Error:"
                                "Element row index " << r << " out of range\n";
                        exit(1);
                }
                if(c > _dims-1){
                        std::cout << "SqMatrix Get_Element Error:"
                                "Element column index " << c << " out of range\n";
                        exit(1);
                }
        #endif

        return _elems[r * _dims + c];
}

// SET HANDLER
inline void SqMatrix::set(unsigned int r, unsigned int c, comp e){
        // you can ignore this. vector will crash anyway
        #ifdef __SAFE_LIBRARY__
                if(r > _dims-1){
                        std::cout << "SqMatrix Set_Element Error:"
                                "Element row index " << r << " out of range\n";
                        exit(1);
                }
                if(c > _dims-1){
                        std::cout << "SqMatrix Set_Element Error:"
                                "Element column index " << c << " out of range\n";
                        exit(1);
                }
        #endif

        _elems[r * _dims + c] = e;
}



// LINEAR ALGEBRA
```

```
// Tensor product for square matrices
SqMatrix square_tensor_prod(
        const SqMatrix &arr1,
        const SqMatrix &arr2)
{
        // tensor product dimension
        unsigned int dim = arr1.dims() * arr2.dims();
        unsigned int p = arr2.dims(); // power of two

        // find which power of two is p
        unsigned int power;
        for(power = 0; p >> power != 1; ++power) ;

        // =========== FILL ARRAY =========== //
        SqMatrix elements (dim);


        for(unsigned int i = 0; i < dim; ++i) {
                for(unsigned int j = 0; j < dim; ++j)
                        // C_ij = a[i/p][j/p] * b[i%p][j%p];
                        // dimensions are power of two!!! optimize / and %
                        elements.set(i, j,
                                arr1.get(i>>power,j>>power) * arr2.get(i&(p-1),j&(p-1)) );
        }

        return elements;
}


// return an I array with dimensions dim x dim
SqMatrix Identity_matrix(unsigned int dim) {

        SqMatrix result(dim);

        for(unsigned int i = 0; i < dim; ++i)
                for(unsigned int j = 0; j < dim; ++j)
                        result.set(i, j, c((i == j) ? 1.0 : 0.0, 0.0) );

        return result;
}




/*
        / S 0 0 0 \ / psi1 \
        | 0 S 0 0 | | psi2 |
        | 0 0 S 0 | | .... |
        \ 0 0 0 S / \ psin /
*/
// Given S and Statevector only!!!
std::vector<comp> Sparse_Array_dot_state(
        const SqMatrix &array,
        const std::vector<comp> &state)
{
        // how many S's in the diagonal?
        unsigned int Id_dim = state.size() / array.dims();


#ifdef __SAFE_LIBRARY__
        // =========== CHECK DIMENSIONS! =========== //
        if(state.size() != Id_dim * array.dims()) {
                std::cout << "Array Dot State Error: "
                        "Array dimensions(" << array.dims() <<
                        ") and State dimensions(" << state.size() <<
                        ") don't match\n";
                        exit(1);
        }
```

```
#endif

        unsigned int dim = state.size();
        std::vector<comp> res(dim);
        unsigned int s_dim = array.dims();


        // for every sub-matrix S in the diagonal
        for(unsigned int s = 0; s < Id_dim; ++s) {
                // Iterate S
                for(unsigned int i = 0; i < s_dim; ++i) {
                        comp c_i = c(0.0, 0.0);
                        // c_i = SUM a_ik b_k
                        for(unsigned int k = 0; k < s_dim; ++k)
                                // offset for statevector --> some S arrays
                                c_i = c_i + array.get(i,k) * state[s_dim * s + k];

                        // result row in statevector
                        res[s_dim * s + i] = c_i;
                }
        }
        return res;
}



bool is_hermitian( const SqMatrix &Obs ) {
        // ============ CHECK ELEMENTS ============ //
        // Hermitian means: a_ij = a_ji*
        for(unsigned int i = 0; i < Obs.dims(); ++i) {
                for(unsigned int j = 0; j < Obs.dims(); ++j)
                        // allow a small error --> their distance
                        if( distance(Obs.get(i,j) , conj(Obs.get(j,i))) > 0.0001 )
                                return false;
        }

        return true;
}



// MUST BE DONE WITH SPARSE ARRAYS, TOO!
// Returns:   <A> = <Psi|A|Psi>
c_type expectation_value (
        const SqMatrix &Obs,
        const std::vector<comp> &state )
{
        #ifdef __SAFE_LIBRARY__
                // =========== ARRAY CHECKS =========== //
                if(!is_hermitian(Obs)) {
                        std::cout << "Expectation Value Error: "
                                "Observable matrix is not Hermitian!\n";
                        exit(1);
                }
        #endif

        // compute
        unsigned int dim = state.size();
        comp tmp_res = c(0.0, 0.0);
        for(unsigned int i = 0; i < dim; ++i) {
                comp tmp = c(0.0, 0.0);
                for(unsigned int j = 0; j < dim; ++j)
                        tmp = tmp + Obs.get(i,j) * state[j];
                tmp_res = tmp_res + mul_conj(state[i], tmp);
        }

        // check imaginary part. Should be zero or very small
        #ifdef __SAFE_LIBRARY__
```

```
                // this complex should be real.
                if( abs(tmp_res.imag) > 0.0001 ) {
                        std::cout << "Expectation Value Error: "
                                "Result has big imaginary part\n";
                        exit(1);
                }
        #endif


        return tmp_res.real;
}



// Return transpose conjugate of a matrix
inline SqMatrix dagger(const SqMatrix & m) {
        unsigned int d = m.dims();

        SqMatrix res(d);

        // b_ij = a_ji*
        // RE-ORDER FOR CACHE EFFICIENCY
        for(unsigned int i = 0; i < d; ++i)
                for(unsigned int j = 0; j < d; ++j)
                        res.set(i,j, conj(m.get(j,i)) );

        return res;
}



// inner product of two complex vectors
comp inner_product(
        const std::vector<comp> &v1,
        const std::vector<comp> &v2)
{
        unsigned int s1 = v1.size();
        #ifdef __SAFE_LIBRARY__
                if(s1 != v2.size()) {
                        std::cout << "Inner Product Error: "
                                "vector sizes are different\n";
                        exit(1);
                }
        #endif

        comp res = c(0.0, 0.0);
        for(unsigned int i = 0; i < s1; ++i)
                res = res + mul_conj(v1[i], v2[i]);

        return res;
}


////////////////////////////////

// DEBUG INFO
void print_vector(std::vector<c_type> vec) {
        std::cout << "Printing Vector\n";
        for(unsigned int i = 0; i < vec.size(); ++i)
                std::cout << vec[i] << " ";
        std::cout << "\n";
}


void print_matrix(const SqMatrix &m) {
        int r = m.dims();

        for(int i = 0; i < r; ++i) {
                for(int j = 0; j < r; ++j)
                        print_comp( m.get(i, j) );
```

```
                std::cout << "\n";
        }
}

////////////////////////////////
// SqMatrix operators



// Make tensor product as operator %
inline SqMatrix operator % (
        const SqMatrix &arr1,
        const SqMatrix &arr2)
{
        return square_tensor_prod(arr1, arr2);
}


// Add arrays
inline SqMatrix operator + (
        const SqMatrix &arr1,
        const SqMatrix &arr2)
{
        #ifdef __SAFE_LIBRARY__
                if(arr1.dims() != arr2.dims()) {
                        std::cout << "SqMatrix operator + error: "
                                "Different sizes\n";
                                exit(1);
                }
        #endif

        unsigned int n = arr1.dims();

        SqMatrix res(n);

        for(unsigned int i = 0; i < n; ++i)
                for(unsigned int j = 0; j < n; ++j)
                        res.set(i,j, arr1.get(i,j) + arr2.get(i,j));

        return res;
}


// Add arrays
inline SqMatrix operator - (
        const SqMatrix &arr1,
        const SqMatrix &arr2)
{
        #ifdef __SAFE_LIBRARY__
                if(arr1.dims() != arr2.dims()) {
                        std::cout << "SqMatrix operator + error: "
                                "Different sizes\n";
                                exit(1);
                }
        #endif

        unsigned int n = arr1.dims();

        SqMatrix res(n);

        for(unsigned int i = 0; i < n; ++i)
                for(unsigned int j = 0; j < n; ++j)
                        res.set(i,j, arr1.get(i,j) - arr2.get(i,j));

        return res;
}
```

```
// Make number * array operator
inline SqMatrix operator * (
        const c_type number,
        const SqMatrix & arr)
{
        unsigned int n = arr.dims();

        SqMatrix res(n);

        for(unsigned int i = 0; i < n; ++i)
                for(unsigned int j = 0; j < n; ++j)
                        res.set(i,j, arr.get(i,j) * number);

        return res;
}
#endif
```

10.5 gate_library

```
#ifndef __ARRAY_LIBRARY__
#define __ARRAY_LIBRARY__ 420

#include <vector>
#include <math.h>
#include "parameters.h"
#include "comp.h"
#include "algebra.h"


// contains arrays and
// matrices built on top of those arrays


// Standard Gate ARRAYS!!!!


// NON- PARAMETRIC GATES

// ===== ONE QUBIT OPEARATIONS =====
// 1 0
// 0 1
const std::vector<std::vector<comp>> arr_I {
        { c(1.0,0.0), c(0.0,0.0) },
        { c(0.0,0.0), c(1.0,0.0) }
};

// 1                1  1
// sqrt(2)        1 -1
const c_type sqrt1_2 = sqrt(0.5);
std::vector<std::vector<comp>> arr_H {
        { c(sqrt1_2,0.0), c( sqrt1_2,0.0)},
        { c(sqrt1_2,0.0), c(-sqrt1_2,0.0)}
};

// Pauli Gates
// 0 1
// 1 0
const std::vector<std::vector<comp>> arr_X {
        { c(0.0,0.0), c(1.0,0.0) },
        { c(1.0,0.0), c(0.0,0.0) }
};

// 0 -i
// i  0
const std::vector<std::vector<comp>> arr_Y {
        { c(0.0,0.0), c(0.0,-1.0) },
        { c(0.0,1.0), c(0.0, 0.0) }
};

// 1  0
```

```
// 0 -1
const std::vector<std::vector<comp>> arr_Z {
        { c(1.0,0.0), c( 0.0,0.0) },
        { c(0.0,0.0), c(-1.0,0.0) }
};

// 1  0                        1 0
// 0  e^{i pi/2} =        0 i
const std::vector<std::vector<comp>> arr_S {
        { c(1.0,0.0), c(0.0,0.0) },
        { c(0.0,0.0), c(0.0,1.0) }
};

// 1  0
// 0  e^{i pi/4}
const std::vector<std::vector<comp>> arr_T {
        { c(1.0,0.0), c(0.0,0.0) },
        { c(0.0,0.0), c(cos(pi/4),sin(pi/4)) }
};



// ===== TWO QUBIT OPERATIONS =====
// 1 0 0 0
// 0 1 0 0
// 0 0 0 1
// 0 0 1 0
const std::vector<std::vector<comp>> arr_CNOT {
        { c(1.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0) },
        { c(0.0,0.0), c(1.0,0.0), c(0.0,0.0), c(0.0,0.0) },
        { c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(1.0,0.0) },
        { c(0.0,0.0), c(0.0,0.0), c(1.0,0.0), c(0.0,0.0) }
};

// 1 0 0  0
// 0 1 0  0
// 0 0 0 -i
// 0 0 i  0
const std::vector<std::vector<comp>> arr_CY {
        { c(1.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0) },
        { c(0.0,0.0), c(1.0,0.0), c(0.0,0.0), c(0.0,0.0) },
        { c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,-1.0) },
        { c(0.0,0.0), c(0.0,0.0), c(0.0,1.0), c(0.0,0.0) }
};

// 1 0 0  0
// 0 1 0  0
// 0 0 1  0
// 0 0 0 -1
const std::vector<std::vector<comp>> arr_CZ {
        { c(1.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0) },
        { c(0.0,0.0), c(1.0,0.0), c(0.0,0.0), c(0.0,0.0) },
        { c(0.0,0.0), c(0.0,0.0), c(1.0,0.0), c(0.0,0.0) },
        { c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(-1.0,0.0) }
};

// 1 0 0 0
// 0 0 1 0
// 0 1 0 0
// 0 0 0 1
const std::vector<std::vector<comp>> arr_SWAP {
        { c(1.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0) },
        { c(0.0,0.0), c(0.0,0.0), c(1.0,0.0), c(0.0,0.0) },
        { c(0.0,0.0), c(1.0,0.0), c(0.0,0.0), c(0.0,0.0) },
        { c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(1.0,0.0) }
};

// ===== THREE QUBIT OPERATIONS =====
const std::vector<std::vector<comp>> arr_TOFFOLI {
        { c(1.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0) },
```

```
        { c(0.0,0.0), c(1.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0) },
        { c(0.0,0.0), c(0.0,0.0), c(1.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0) },
        { c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(1.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0) },
        { c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(1.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0) },
        { c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(1.0,0.0), c(0.0,0.0), c(0.0,0.0) },
        { c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(1.0,0.0) },
        { c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(1.0,0.0), c(0.0,0.0) }
};



/*
 * PARAMETRIC OPERATIONS
 */


// ===== ONE QUBIT OPEARATIONS =====

// Rotation gates
std::vector<std::vector<comp>> arr_RX (c_type angle) {
        return {
                { c(cos(angle/2.0),0.0), c(0.0,-sin(angle/2.0)) },
                { c(0.0,-sin(angle/2.0)), c(cos(angle/2.0),0.0) }
        };
}


std::vector<std::vector<comp>> arr_RY (c_type angle) {
        return {
                { c(cos(angle/2.0),0.0), c(-sin(angle/2.0),0.0) },
                { c(sin(angle/2.0),0.0), c(cos(angle/2.0),0.0) }
        };
}


std::vector<std::vector<comp>> arr_RZ (c_type angle) {
        return {
                { c(cos(angle/2.0),-sin(angle/2.0)), c(0.0,0.0) },
                { c(0.0,0.0), c(cos(angle/2.0),sin(angle/2.0)) }
        };
}


// 1 0
// 0 e^{i phi}
std::vector<std::vector<comp>> arr_P (c_type angle) {
        return {
                { c(1.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(cos(angle),sin(angle)) }
        };
}


// ===== TWO QUBIT OPEARATIONS =====
std::vector<std::vector<comp>> arr_CRX (c_type angle) {
        c_type phi = angle/2.0;

        return {
                { c(1.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(1.0,0.0), c(0.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(0.0,0.0), c(cos(phi),0.0), c(0.0,-sin(phi)) },
                { c(0.0,0.0), c(0.0,0.0), c(0.0,-sin(phi)), c(cos(phi),0.0) }
        };
}

std::vector<std::vector<comp>> arr_CRY (c_type angle) {
        c_type phi = angle/2.0;

        return {
```

```
                { c(1.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(1.0,0.0), c(0.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(0.0,0.0), c(cos(phi),0.0), c(-sin(phi),0.0) },
                { c(0.0,0.0), c(0.0,0.0), c(sin(phi),0.0), c(cos(phi),0.0) }
        };
}

std::vector<std::vector<comp>> arr_CRZ (c_type angle) {
        c_type phi = angle/2.0;

        return {
                { c(1.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(1.0,0.0), c(0.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(0.0,0.0), c(cos(phi),-sin(phi)), c(0.0,0.0) },
                { c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(cos(phi), sin(phi)) }
        };
}


std::vector<std::vector<comp>> arr_CP (c_type angle) {
        return {
                { c(1.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(1.0,0.0), c(0.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(0.0,0.0), c(1.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(0.0,0.0), c(0.0,0.0),c(cos(angle),sin(angle)) }
        };
}



/*
 * GRADIENTS OF PARAMETRIC GATES
 */
std::vector<std::vector<comp>> arr_RX_dif (c_type angle) {
        return {
                { c(-sin(angle/2.0)/2.0,0.0), c(0.0,-cos(angle/2.0)/2.0) },
                { c(0.0,-cos(angle/2.0)/2.0), c(-sin(angle/2.0)/2.0,0.0) }
        };
}



std::vector<std::vector<comp>> arr_RY_dif (c_type angle) {
        return {
                { c(-sin(angle/2.0)/2.0,0.0), c(-cos(angle/2.0)/2.0,0.0) },
                { c( cos(angle/2.0)/2.0,0.0), c(-sin(angle/2.0)/2.0,0.0) }
        };
}



std::vector<std::vector<comp>> arr_RZ_dif (c_type angle) {
        return {
                { c(-sin(angle/2.0)/2.0,-cos(angle/2.0)/2.0), c(0.0,0.0) },
                { c(0.0,0.0), c(-sin(angle/2.0)/2.0,cos(angle/2.0)/2.0) }
        };
}


// 1 0
// 0 e^{i phi}
std::vector<std::vector<comp>> arr_P_dif (c_type angle) {
        return {
                { c(1.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(-sin(angle),cos(angle)) }
        };
}


// ===== TWO QUBIT OPEARATIONS =====
```

```
std::vector<std::vector<comp>> arr_CRX_dif (c_type angle) {
        c_type phi = angle/2.0;

        return {
                { c(1.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(1.0,0.0), c(0.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(0.0,0.0), c(-sin(phi)/2.0,0.0), c(0.0,-cos(phi)/2.0) },
                { c(0.0,0.0), c(0.0,0.0), c(0.0,-cos(phi)/2.0), c(-sin(phi)/2.0,0.0) }
        };
}

std::vector<std::vector<comp>> arr_CRY_dif (c_type angle) {
        c_type phi = angle/2.0;

        return {
                { c(1.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(1.0,0.0), c(0.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(0.0,0.0), c(-sin(phi)/2.0,0.0), c(-cos(phi)/2.0,0.0) },
                { c(0.0,0.0), c(0.0,0.0), c(cos(phi)/2.0,0.0), c(-sin(phi)/2.0,0.0) }
        };
}

std::vector<std::vector<comp>> arr_CRZ_dif (c_type angle) {
        c_type phi = angle/2.0;

        return {
                { c(1.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(1.0,0.0), c(0.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(0.0,0.0), c(-sin(phi)/2.0,-cos(phi)/2.0), c(0.0,0.0) },
                { c(0.0,0.0), c(0.0,0.0), c(0.0,0.0), c(-sin(phi)/2.0,cos(phi)/2.0) }
        };
}


std::vector<std::vector<comp>> arr_CP_dif (c_type angle) {
        return {
                { c(1.0,0.0), c(0.0,0.0), c(0.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(1.0,0.0), c(0.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(0.0,0.0), c(1.0,0.0), c(0.0,0.0) },
                { c(0.0,0.0), c(0.0,0.0), c(0.0,0.0),c(-sin(angle),cos(angle)) }
        };
}



// Debug info
void print_array(std::vector<std::vector<comp>> arr) {
        unsigned int dim1 = arr.size();

        for(unsigned int i = 0; i < dim1; ++i) {
                unsigned int dim2 = arr[i].size();

                for(unsigned int j = 0; j < dim2; ++j)
                        print_comp(arr[i][j]);

                std::cout << "\n";
        }

}



const SqMatrix empty(0);



// NON-PARAMETRIC MATRICES
// ===== ONE QUBIT OPERATIONS =====
const SqMatrix matrix_I(arr_I);
```

```
const SqMatrix matrix_H(arr_H);
const SqMatrix matrix_X(arr_X);
const SqMatrix matrix_Y(arr_Y);
const SqMatrix matrix_Z(arr_Z);
const SqMatrix matrix_S(arr_S);
const SqMatrix matrix_T(arr_T);

// ===== TWO QUBIT OPERATIONS =====
const SqMatrix matrix_CNOT(arr_CNOT);
const SqMatrix matrix_CY(arr_CY);
const SqMatrix matrix_CZ(arr_CZ);
const SqMatrix matrix_SWAP(arr_SWAP);

// ===== THREE QUBIT OPERATIONS =====
const SqMatrix matrix_TOFFOLI(arr_TOFFOLI);


// PARAMETRIC MATRICES
// ===== ONE QUBIT OPERATIONS =====
SqMatrix matrix_RX(c_type angle) { return SqMatrix(arr_RX(angle)); }
SqMatrix matrix_RY(c_type angle) { return SqMatrix(arr_RY(angle)); }
SqMatrix matrix_RZ(c_type angle) { return SqMatrix(arr_RZ(angle)); }
SqMatrix matrix_P (c_type angle) { return SqMatrix(arr_P (angle)); }
// SqMatrix matrix_ROT(c_type phi, c_type theta, c_type omega) { return SqMatrix(arr_ROT(phi, theta, omega)); }
// ===== TWO QUBIT OPERATIONS =====
SqMatrix matrix_CRX(c_type angle) { return SqMatrix(arr_CRX(angle)); }
SqMatrix matrix_CRY(c_type angle) { return SqMatrix(arr_CRY(angle)); }
SqMatrix matrix_CRZ(c_type angle) { return SqMatrix(arr_CRZ(angle)); }
SqMatrix matrix_CP(c_type angle) { return SqMatrix(arr_CP(angle)); }



// DERIVATIVES OF GATES
// ===== ONE QUBIT OPERATIONS =====
SqMatrix matrix_RX_dif(c_type angle) { return SqMatrix(arr_RX_dif(angle)); }
SqMatrix matrix_RY_dif(c_type angle) { return SqMatrix(arr_RY_dif(angle)); }
SqMatrix matrix_RZ_dif(c_type angle) { return SqMatrix(arr_RZ_dif(angle)); }
SqMatrix matrix_P_dif (c_type angle) { return SqMatrix(arr_P_dif (angle)); }
// SqMatrix matrix_ROT(c_type phi, c_type theta, c_type omega) { return SqMatrix(arr_ROT(phi, theta, omega)); }
// ===== TWO QUBIT OPERATIONS =====
SqMatrix matrix_CRX_dif(c_type angle) { return SqMatrix(arr_CRX_dif(angle)); }
SqMatrix matrix_CRY_dif(c_type angle) { return SqMatrix(arr_CRY_dif(angle)); }
SqMatrix matrix_CRZ_dif(c_type angle) { return SqMatrix(arr_CRZ_dif(angle)); }
SqMatrix matrix_CP_dif(c_type angle) { return SqMatrix(arr_CP_dif(angle)); }
#endif
```

10.6 application

```
#ifndef __APPLICATION__
#define __APPLICATION__ 420

#include <iostream>
#include <vector>

#include "parameters.h"
#include "comp.h"
#include "algebra.h"



// GATE APPLICATION
class Application {
        public:
                // gate
                SqMatrix application_gate;
                // gate derivative
                SqMatrix application_gate_dif;
                // qubits this gate is applied unto
                std::vector<unsigned int> qubits;
```

```
                // Application Constructor
                Application(SqMatrix g, SqMatrix g_dif, std::vector<unsigned int> q )
                : application_gate(g), application_gate_dif(g_dif), qubits(q)

                {
#ifdef __SAFE_LIBRARY__
                        // Check qubits-gate dimensions!
                        unsigned int qubs = q.size();
                        if (g.dims() != (unsigned int) (1<<qubs)) {
                                std::cout << "Application Constructor Error:"
                                        "qubits don't match array size\n";
                                        exit(1);
                        }

                        // Check for duplicates in qubit list
                        std::vector<bool> qubit_used(MAX_QUBITS, false);
                        for(unsigned int i = 0; i < qubs; ++i) {
                                if(qubit_used[q[i]] == true) {
                                        std::cout << "Application Constructor Error: "
                                                "Dulpicate Qubit Detected!\n";
                                        exit(1);
                                }
                                qubit_used[q[i]] = true;
                        }
#endif
                };
};
#endif
```

10.7 circuit

```
#ifndef __QUANTUM_CIRCUIT__
#define __QUANTUM_CIRCUIT__ 420


#include <iostream>         // input/output
#include <vector>         // all my arrays
#include <map>                  // dictionary for measure();
#include <bitset>    // convert integer to bin_string
#include <random>         // discrete_distribution
#include <math.h>         // floor
#include <unordered_map>          // application SN --> actual gate
#include <tuple>         // (Layer, Layer_number)

#include "parameters.h"
#include "algebra.h"
#include "comp.h"
#include "application.h"
#include "gate_library.h"



#define max(a,b) ((a)>(b)?(a):(b))


class Circuit {
        private:
                // Core Utilities
                unsigned int qubits;
                std::vector<unsigned int> available_layer;
                std::vector< std::vector<Application> > layers;
                std::vector<comp> state_vector;
                bool executed;


                // Adjoint Differentiation
                // a map that keeps track of application Serial Numbers
                std::unordered_map<unsigned int,
                        std::tuple<unsigned int, unsigned int>> app_id;
```

```
                unsigned int applications;
                unsigned int parametric_gates;


        public:
                // Constructor
                Circuit(unsigned int q);



                // =========== circuit_core.h =========== //
                void app (Application &a);
                void run();
                c_type exp_value (
                        const SqMatrix &Obs,
                        const std::vector<unsigned int> &qubs);

                // returns Nabla <E> for an observable E
                std::vector<c_type> exp_value_grad(
                        const SqMatrix &Obs,
                        const std::vector<unsigned int> &qubs);



                // ======== circuit_statistics.h ======== //
                void print_circuit();
                void print_state_vector(bool integer);
                void print_probabilities(bool integer, bool cent);
                void measure(unsigned int shots);




                // ========== circuit_library.h ========= //
                // NON-PARAMETRIC GATES
                void I(unsigned int qubit);
                void H(unsigned int qubit);
                void X(unsigned int qubit);
                void Y(unsigned int qubit);
                void Z(unsigned int qubit);
                void S(unsigned int qubit);
                void T(unsigned int qubit);
                void CNOT(unsigned int control, unsigned int target);
                void CY(unsigned int control, unsigned int target);
                void CZ(unsigned int control, unsigned int target);
                void SWAP(unsigned int qubit1, unsigned int qubit2);

                // PARAMETRIC GATES
                void RX(unsigned int qubit, c_type angle);
                void RY(unsigned int qubit, c_type angle);
                void RZ(unsigned int qubit, c_type angle);
                void P (unsigned int qubit, c_type angle);
                // void ROT(unsigned int qubit, c_type phi, c_type theta, c_type omega);
                void CRX(unsigned int control, unsigned int target, c_type angle);
                void CRY(unsigned int control, unsigned int target, c_type angle);
                void CRZ(unsigned int control, unsigned int target, c_type angle);
                void CP(unsigned int control, unsigned int target, c_type angle);
                void TOFFOLI(unsigned int control1, unsigned int control2, unsigned int target);
};


// Constructor
Circuit::Circuit(unsigned int q) {
        #ifdef __SAFE_LIBRARY__
                // check qubit range
                if(q == 0) {
                        std::cout << "Zero Qubits Given!\n";
                        exit(1);
                }
```

```
                if(q > MAX_QUBITS) {
                        std::cout << "More than " << MAX_QUBITS
                                << " Qubits! You are a mad man....\n";
                        exit(1);
                }
        #endif


        // Initialize data
        state_vector = std::vector<comp> ( 1<<q, c(0.0, 0.0) );
        state_vector[0] = c(1.0, 0.0);
        qubits = q;
        available_layer = std::vector<unsigned int> (q,0);
        applications = 0;
        parametric_gates = 0;
        executed = false;
};
#endif
```

10.8 circuit_core

```
#include "circuit.h"


// Circuit Core utilities


// ========== APPLY GATE ON CIRCUIT ========== //
void Circuit::app(Application &a) {

        unsigned int pos_in_layer;

        // ========== FIND THE RIGHT LAYER ========== //
        unsigned int layer_index = 0;
        unsigned int max_qubit_used = 0;
        for(unsigned int i = 0; i < a.qubits.size(); ++i) {
                layer_index = max(layer_index, available_layer[a.qubits[i]]);
                max_qubit_used = max(max_qubit_used, a.qubits[i]);
        }


#ifdef __SAFE_LIBRARY__
        // ========== QUBIT OUT OF RANGE ========== //
        if(max_qubit_used > qubits - 1) {
                std::cout << "Application Error: "
                        "Qubit " << max_qubit_used << " out of Range! "
                        "(0, " << qubits - 1 << ")\n";
                exit(1);
        }
#endif


        // ========== NEW CIRCUIT LAYER? ========== //
        if( layers.size() < layer_index + 1) {
                layers.push_back( std::vector<Application> {a} );
                pos_in_layer = 0;
        }
        else {
                layers[layer_index].push_back(a);
                pos_in_layer = layers[layer_index].size() - 1;
        }


        // ========== UPDATE AVAILABLE QUBIT POSITIONS ========== //
        for(unsigned int i = 0; i < a.qubits.size(); ++i)
                available_layer[a.qubits[i]] = layer_index + 1;


        // map application's Serial Number to its place on the circuit
        app_id[applications++] = std::make_tuple(layer_index, pos_in_layer);
```

```
        executed = false;
}



// apply gate on State_Vector
// given a permutation map for the qubits
void apply_on_map(
                std::vector<comp> &SV,
                std::vector<unsigned int> &permutation,
                unsigned int qubits,
                SqMatrix &gate)
{
        // =========== MAP STATEVECTOR =========== //
        std::vector<comp> mapped_state_vector = SV;

        // iterate state vector and map fake qubits to real qubits
        unsigned int SV_size = 1 << qubits;
        for(unsigned int pos = 0; pos < SV_size; ++pos) {

                unsigned int mapped_pos = 0;
                for(unsigned int fake = 0; fake < qubits; ++fake) {
                        unsigned int real = permutation[fake];
                        // replace fake qubit with the real one
                        unsigned int real_bit = (pos >> real) & 1;
                        mapped_pos |= (real_bit << fake);
                }
                mapped_state_vector[mapped_pos] = SV[pos];
        }

        // =========== APPLY LAYER =========== //
        mapped_state_vector =
                Sparse_Array_dot_state(gate, mapped_state_vector);

        // =========== UNDO MAPPING OF STATEVECTOR =========== //
        for(unsigned int pos = 0; pos < SV_size; ++pos) {

                unsigned int mapped_pos = 0;
                for (unsigned int fake = 0; fake < qubits; ++fake) {
                        unsigned int real = permutation[fake];
                        unsigned int real_bit = (pos >> real) & 1;
                        mapped_pos |= (real_bit << fake);
                }
                // reverse map
                SV[pos] = mapped_state_vector[mapped_pos];
        }
}


// returns qubits used for gates
void make_permutation(
                std::vector<unsigned int> & permutation,
                unsigned int qubits,
                const std::vector<Application> & layer)
{
        std::vector<bool> allocated(qubits, false);

        // PREMARE MAP AND PARAMETERS
        int gates = layer.size();
        unsigned int last_used_qubit = 0;

        // ITERATE LAYER OF GATES
        for(int i = 0; i < gates; ++i) {

                unsigned int gate_size = layer[i].qubits.size();

                // ITERATE GATE
                for(unsigned int qubit = 0; qubit < gate_size; ++qubit) {
```

```
                        permutation[qubit + last_used_qubit] = layer[i].qubits[qubit];
                        allocated[layer[i].qubits[qubit]] = true;
                }
                last_used_qubit += gate_size; // qubits used on this gate
        }

        // =========== FILL PERMUTATION WITH UNALLOCATED WIRES =========== //
        for(unsigned int un_it = 0; un_it < qubits; ++un_it) {
                if(allocated[un_it] == false)
                        permutation[last_used_qubit++] = un_it;
        }
}



// Tensor product of many gates
void fill_gate(
                SqMatrix & layer_array,
                const std::vector<Application> & layer)
{
        // Layer (by definition) is never empty!
        layer_array = layer[0].application_gate;

        // fill with gates given
        int gates = layer.size();
        for(int i = 1; i < gates; ++i) {
                layer_array = layer[i].application_gate % layer_array;
        }
}


// Tensor product of many gates + fill with Identities
void full_fill_gate(
                SqMatrix & layer_array,
                const std::vector<Application> & layer,
                unsigned int qubits )
{
        // Layer (by definition) is never empty!
        layer_array = layer[0].application_gate;

        // fill with gates given
        int gates = layer.size();
        for(int i = 1; i < gates; ++i) {
                layer_array = layer[i].application_gate % layer_array;
        }

        unsigned int rest_wires_dim = (1<<qubits) / layer_array.dims();

        SqMatrix big_id = Identity_matrix(rest_wires_dim);

        layer_array = big_id % layer_array;
}



// =========== RUN AND UPDATE STATE VECTOR =========== //
void Circuit::run(){

        // don't run the same circuit twice
        if(executed) return;

        // =========== CLEAR STATE VECTOR =========== //
        unsigned int SV_size = 1 << qubits;
        state_vector[0] = c(1.0, 0.0);
        for(unsigned int i = 1; i < SV_size; ++i)
                state_vector[i] = c(0.0, 0.0);


        // =========== LAYER BY LAYER =========== //
        int depth = layers.size();
```

```
        for(int i = 0; i < depth; ++i) {


                // IDEA: Split full layers to half-half
                // Maybe it is a good optimization for full layers
                // Needs proof

                std::vector<unsigned int> permutation(qubits);

                make_permutation(permutation, qubits, layers[i]);
                SqMatrix layer_array(0);        // init as emptys
                fill_gate(layer_array, layers[i]);
                apply_on_map(state_vector, permutation, qubits, layer_array);

        }

        executed = true;
}



// Takes A and list of qubits A is applied onto
// Returns:   <A> = <Psi|A|Psi>
c_type Circuit::exp_value (
                const SqMatrix &Obs,
                const std::vector<unsigned int> &qubs)
{
#ifdef __SAFE_LIBRARY__
        // =========== COMPATIBILITY CHECKS =========== //
        unsigned int qub_len = 1 << qubs.size();
        if(Obs.dims() != qub_len) {
                std::cout << "Expectation Value Error: "
                        "Observable matrix size doesn't match qubits given\n";
                exit(1);
        }
        // =========== QUBIT CHECKS =========== //
        std::vector<bool> qubit_used(MAX_QUBITS, false);
        for(unsigned int i = 0; i < qubs.size(); ++i) {
                // Check range
                if(qubs[i] > qubits - 1) {
                        std::cout << "Expectation Value Error: "
                                "Qubit " << qubs[i] << " out of Range "
                                "(0," << qubits-1 << ")\n";
                        exit(1);
                }
                // Check duplicates
                if(qubit_used[qubs[i]] == true) {
                        std::cout << "Expectation Value Error: "
                                "Dulpicate Qubit Detected!\n";
                        exit(1);
                }
                qubit_used[qubs[i]] = true;
        }
#endif

        run();

        // Like run(); only one layer containing Obs gate
        std::vector<Application> layer = {Application(Obs, empty, qubs)};

        std::vector<unsigned int> permutation(qubits);

        make_permutation(permutation, qubits, layer);

        SqMatrix layer_array(0);        // empty
        full_fill_gate(layer_array, layer, qubits);

        // Don't call apply_on_map since we don't need to unmap
        // =========== MAP STATEVECTOR =========== //
        std::vector<comp> mapped_state_vector = state_vector;
```

```
        // iterate state vector and map real qubits to fake qubits
        unsigned int SV_size = 1 << qubits;
        for(unsigned int pos = 0; pos < SV_size; ++pos) {

                unsigned int mapped_pos = 0;
                for(unsigned int fake = 0; fake < qubits; ++fake) {
                        unsigned int real = permutation[fake];
                        // replace fake qubit with the real one
                        unsigned int real_bit = (pos >> real) & 1;
                        mapped_pos |= (real_bit << fake);
                }
                mapped_state_vector[mapped_pos] = state_vector[pos];
        }

        c_type exp_val = expectation_value(layer_array, mapped_state_vector);

        return exp_val;
}



// https://arxiv.org/pdf/2009.02823.pdf
// returns Nabla <E> for an observable E
std::vector<c_type> Circuit::exp_value_grad(
                const SqMatrix &Obs,
                const std::vector<unsigned int> &qubs)
{

#ifdef __SAFE_LIBRARY__
        // =========== COMPATIBILITY CHECKS =========== //
        unsigned int qub_len = 1 << qubs.size();
        if(Obs.dims() != qub_len) {
                std::cout << "Expectation Value Error: "
                        "Observable matrix size doesn't match qubits given\n";
                exit(1);
        }
        // =========== QUBIT CHECKS =========== //
        std::vector<bool> qubit_used(MAX_QUBITS, false);
        for(unsigned int i = 0; i < qubs.size(); ++i) {
                // Check range
                if(qubs[i] > qubits - 1) {
                        std::cout << "Expectation Value Error: "
                                "Qubit " << qubs[i] << " out of Range "
                                "(0," << qubits-1 << ")\n";
                        exit(1);
                }
                // Check duplicates
                if(qubit_used[qubs[i]] == true) {
                        std::cout << "Expectation Value Error: "
                                "Dulpicate Qubit Detected!\n";
                        exit(1);
                }
                qubit_used[qubs[i]] = true;
        }
#endif


        run();


        // result vector --> Nabla <E>
        std::vector<c_type> grad (parametric_gates);
        unsigned int param = parametric_gates - 1;

        // Find <E>
        // YOU MUST RUN THE CIRCUIT FIRST!
        run();
        std::vector<comp> phi    = state_vector;
```

```
std::vector<comp> lambda = state_vector;


// ============================================== //
// ======= |Lambda> = Observable |Lambda> ======= //
// ============================================== //

// A layer containing only Obs array
std::vector<Application> layer = {Application(Obs, empty, qubs)};
std::vector<unsigned int> perm(qubits);

// PERMUTE
make_permutation(perm, qubits, layer);
// MAKE GATE
SqMatrix filled_gate(0);        // init as empty
fill_gate(filled_gate, layer);
// APPLY GATE ON GIVEN STATEVECTOR WITH GIVEN MAP
apply_on_map(lambda, perm, qubits, filled_gate);



// ============================================== //
// ========== ITERATE GATES IN REVERSE ========== //
// ============================================== //


// iterate backwards all gates
for(int app = applications-1; app >= 0; --app) {

        // look up in hash --> find gate in circuit
        std::tuple coordinates = app_id[app];
        Application gate =
                layers[std::get<0>(coordinates)][std::get<1>(coordinates)];


        // ============================================== //
        // ============ |Phi> = Ui^dag |Phi> ============ //
        // ============================================== //

        SqMatrix arr = dagger(gate.application_gate);
        std::vector<unsigned int> gate_qubs = gate.qubits;

        // Prepare Layer and permutation
        std::vector<Application> layer = {Application(arr, empty, gate_qubs)};
        std::vector<unsigned int> permutation(qubits);


        make_permutation(permutation, qubits, layer);
        SqMatrix filled_gate(0);        // init as empty
        fill_gate(filled_gate, layer);
        apply_on_map(phi, permutation, qubits, filled_gate);


        // ============================================== //
        // ======= |Mi> = (dUi^dag / dthetai) |Mi> ====== //
        // ============================================== //

        arr = gate.application_gate_dif;

        // PARAMETRIC GATE!
        if(arr.dims() > 1)
        {
                std::vector<comp> mi = phi;

                layer = {Application(arr, empty, gate_qubs)};
                make_permutation(permutation, qubits, layer);

                filled_gate = SqMatrix(0);        // init as empty
                fill_gate(filled_gate, layer);
```

```
                        apply_on_map(mi, permutation, qubits, filled_gate);

                        // Nabla <E>i
                        grad[param--] = 2.0 * inner_product(lambda, mi).real;
                }


                // UPDATE LAMBDA
                if(param > 0) {

                // ============================================== //
                // ========= |Lambda> = Ui^dag |Lambda> ========= //
                // ============================================== //

                arr = dagger(gate.application_gate);

                // Prepare Layer and permutation
                layer = {Application(arr, empty, gate_qubs)};
                make_permutation(permutation, qubits, layer);

                filled_gate = SqMatrix(0);        // init as empty
                fill_gate(filled_gate, layer);
                apply_on_map(lambda, permutation, qubits, filled_gate);
                }


        } // applications loop

        return grad;
}
```

10.9 circuit_library

```
#include "circuit.h"


// ============================================== //
// ============ STANDARD GATE LIBRARY ============ //
// ============================================== //


void Circuit::I(unsigned int qubit){
        Application a(matrix_I, empty, {qubit});
        app(a);
}
void Circuit::H(unsigned int qubit){
        Application a(matrix_H, empty, {qubit});
        app(a);
}
void Circuit::X(unsigned int qubit){
        Application a(matrix_X, empty, {qubit});
        app(a);
}
void Circuit::Y(unsigned int qubit){
        Application a(matrix_Y, empty, {qubit});
        app(a);
}
void Circuit::Z(unsigned int qubit){
        Application a(matrix_Z, empty, {qubit});
        app(a);
}
void Circuit::S(unsigned int qubit){
        Application a(matrix_S, empty, {qubit});
        app(a);
}
void Circuit::T(unsigned int qubit){
        Application a(matrix_T, empty, {qubit});
        app(a);
}
```

```
void Circuit::CNOT(unsigned int control, unsigned int target){
        Application a(matrix_CNOT, empty, {target, control});
        app(a);
}
void Circuit::CY(unsigned int control, unsigned int target){
        Application a(matrix_CY, empty, {target, control});
        app(a);
}
void Circuit::CZ(unsigned int control, unsigned int target){
        Application a(matrix_CZ, empty, {target, control});
        app(a);
}
void Circuit::SWAP(unsigned int qubit1, unsigned int qubit2){
        Application a(matrix_SWAP,  empty,{qubit1, qubit2});
        app(a);
}

void Circuit::TOFFOLI(unsigned int control1, unsigned int control2, unsigned int target){
        Application a(matrix_TOFFOLI, empty, {target, control1, control2});
        app(a);
}



// include derivative for these

void Circuit::RX(unsigned int qubit, c_type angle){
        Application a(matrix_RX(angle), matrix_RX_dif(angle), {qubit});
        app(a);
        parametric_gates++;
}
void Circuit::RY(unsigned int qubit, c_type angle){
        Application a(matrix_RY(angle), matrix_RY_dif(angle), {qubit});
        app(a);
        parametric_gates++;
}
void Circuit::RZ(unsigned int qubit, c_type angle){
        Application a(matrix_RZ(angle), matrix_RZ_dif(angle), {qubit});
        app(a);
        parametric_gates++;
}
void Circuit::P (unsigned int qubit, c_type angle){
        Application a(matrix_P(angle), matrix_P_dif(angle), {qubit});
        app(a);
        parametric_gates++;
}
void Circuit::CRX(unsigned int control, unsigned int target, c_type angle){
        Application a(matrix_CRX(angle), matrix_CRX_dif(angle), {target, control});
        app(a);
        parametric_gates++;
}
void Circuit::CRY(unsigned int control, unsigned int target, c_type angle){
        Application a(matrix_CRY(angle), matrix_CRY_dif(angle), {target, control});
        app(a);
        parametric_gates++;
}
void Circuit::CRZ(unsigned int control, unsigned int target, c_type angle){
        Application a(matrix_CRZ(angle), matrix_CRZ_dif(angle), {target, control});
        app(a);
        parametric_gates++;
}
void Circuit::CP(unsigned int control, unsigned int target, c_type angle){
        Application a(matrix_CP(angle), matrix_CP_dif(angle), {target, control});
        app(a);
        parametric_gates++;
}
```

10.10 circuit_statistics

```cpp
#include "circuit.h"

// ================================================ //
// ============== CIRCUIT STATISTICS ============== //
// ================================================ //



// Print StateVector in binary - integer
void Circuit::print_state_vector(bool integer = false) {

    std::cout << "============ Printing State Vector ============ \n";

        unsigned int dim = state_vector.size();
        for(unsigned int base_state = 0; base_state < dim; ++base_state){

                // Print base state in binary
                if(integer == false) {
                std::string filled_base =
                    std::bitset< MAX_QUBITS >(base_state).to_string();
                std::string base_string(filled_base, MAX_QUBITS - qubits, qubits);

                std::cout << "|" << base_string << ">\t" << "= ";
            }
            // Print base state as integer
            else {
                    std::cout << "|" << base_state << ">\t" << "= ";
            }
                print_comp(state_vector[base_state]);
                std::cout << "\n";
        }
}



// Debug info --> Print layer
void Circuit::print_circuit() {
        std::cout << "============= Printing Circuit ============= \n";

        for(unsigned int i = 0; i < layers.size(); ++i) {
                std::cout << "===== Printing Layer: " << i << " =====\n";

                for(unsigned int j = 0; j < layers[i].size(); ++j){
                        std::cout << "Gate:\n";
                        print_matrix(layers[i][j].application_gate);

                        std::cout << "Applied on Qubit(s): ";
                        for(unsigned int k = 0; k < layers[i][j].qubits.size(); ++k)
                                std::cout << layers[i][j].qubits[k] << " ";
                        std::cout << "\n";
                }
        }
}




// Print probabilities of base_states
void Circuit::print_probabilities(bool integer = false, bool cent = false) {
    std::cout << "============ Printing probabilities ============ \n";

        unsigned int dim = state_vector.size();
        for(unsigned int base_state = 0; base_state < dim; ++base_state){

                // Print base state in binary
                if(integer == false) {
                std::string filled_base =
```

```
                    std::bitset< MAX_QUBITS >(base_state).to_string();
                std::string base_string(filled_base, MAX_QUBITS - qubits, qubits);

                std::cout << "P(|" << base_string << ">)\t" << "= ";
            }
            // Print base state as integer
            else {
                    std::cout << "P(|" << base_state << ">)\t" << "= ";
            }
            comp z = state_vector[base_state];
            // |z|^2 = a^2 + b^2
                c_type prob = z.real * z.real + z.imag * z.imag;
                if(cent == false)
                        std::cout << prob << "\n";
                else
                        std::cout << prob*100 << "%\n";
        }

}



// measure
void Circuit::measure(unsigned int shots){

        std::cout << "============ Running Experiment ============ \n";
        std::cout << "Shots: " << shots << "\n";

    // Setup the random bits
    std::random_device rd;         // Random Device
    std::mt19937_64 gen(rd());         // Mersenne Twister pseudo-random generator

    // Set Distribution Weights
        unsigned int SPACE_SIZE = 1 << qubits;
        std::vector<int> weights (SPACE_SIZE);
        for(unsigned int i = 0; i < SPACE_SIZE; ++i) {
                comp z = state_vector[i];
                c_type prob = z.real * z.real + z.imag * z.imag;
                weights[i] = floor(prob*1000000);         // make integers
        }

    // Create the distribution with those weights
    std::discrete_distribution<> d(weights.begin(), weights.end());

    // Run experiment
    std::map<int, int> counts;
    for(unsigned int shot = 0; shot < shots; ++shot) {
            ++counts[d(gen)];
    }

    // Print measurements
    for(auto p : counts) {
            // Make base_state in binary
                std::string filled_base =
                        std::bitset< MAX_QUBITS >(p.first).to_string();
                std::string base_string(filled_base, MAX_QUBITS - qubits, qubits);
                // Print result
                std::cout << "#(|" << base_string << ">)\t" << "= "
                        << p.second << "\n";
    }
}
```

10.11 optimizers

```
#ifndef __OPTIMIZERS__
#define __OPTIMIZERS__ 420

#include <math.h>
#include <random>
#include <time.h>
```

```cpp
#include "parameters.h"
#include "comp.h"


// DEFINE OPTIMIZER PARAMETERS
#define EPOCHS 500
#define STEP_SIZE 0.01
#define VERBOSE true




// random weights in range (-2pi, 2pi)
std::vector<c_type> random_weights (unsigned int weights_len) {

        srand((unsigned)time(0));
        std::vector<c_type> weights (weights_len, 0.0);

        for(unsigned int i = 0; i < weights_len; ++i) {
                int random_num = (rand()%100000); // in range (0, 999)
                // map (0,999) to (-2pi, 2pi)
                c_type resize = 2*pi/100000;// shrink to (0, 4pi)
                c_type shift  = pi;                         // offset --> 2 pi
                weights[i] = (resize*random_num) - shift;
        }

        return weights;
}




// Adagrad Optimizer
// Take cost, weights --> returns optimized weights
std::vector<c_type> ADAGRAD (
        unsigned int qubits,
        void make_circuit (Circuit &, std::vector<c_type> &),
        SqMatrix Obs,
        std::vector<unsigned int> Obs_qubs,
        std::vector<c_type> init_weights,

        unsigned int epochs = EPOCHS,
        c_type step_size = STEP_SIZE,
        c_type eps = 1e-08,
        bool verbose = VERBOSE
)
{
        // PARAMETERS
        std::vector<c_type> weights = init_weights;
        unsigned int weights_len = weights.size();
        std::vector<c_type> learning_rate (weights_len, step_size);
        std::vector<c_type> alpha (weights_len, 0.0);


        if(verbose)
                std::cout << "========== ADAGRAD =========\n"
                        "Epochs:\t" << epochs << "\t Learning Rate:\t" <<
                        step_size << "\n====================================\n";



        // iterations
        for(unsigned int epoch = 1; epoch <= epochs; ++epoch) {

                // Make Circuit
                Circuit c(qubits);
                make_circuit(c, weights);
```

```
                // Compute Grad for observable
                std::vector<c_type> grad = c.exp_value_grad(Obs, Obs_qubs);


                // UPDATE
                for(unsigned int grad_i = 0; grad_i < weights_len; ++grad_i) {
                        alpha[grad_i]  += grad[grad_i] * grad[grad_i];
                        learning_rate[grad_i] = step_size / sqrt(alpha[grad_i] + eps);
                        weights[grad_i] = weights[grad_i] - learning_rate[grad_i] * grad[grad_i];
                }

                // DEBUG PRINTING
                if(verbose && epoch % 10 == 0) {
                        std::cout << "Epoch:\t" << epoch << "\t<E>: ";
                        std::cout << c.exp_value(Obs, Obs_qubs) << "\n";
                }
        }

        return weights;
}




// Adam Optimizer
// Take cost, weights --> returns optimized weights
std::vector<c_type> ADAM(
        unsigned int qubits,
        void make_circuit (Circuit &, std::vector<c_type> &),
        SqMatrix Obs,
        std::vector<unsigned int> Obs_qubs,
        std::vector<c_type> init_weights,

        unsigned int epochs = EPOCHS,
        c_type step_size = STEP_SIZE,
        c_type beta1_init = 0.9,
        c_type beta2_init = 0.99,
        c_type eps = 1e-08,
        bool verbose = VERBOSE
)
{
        // WEIGHTS
        std::vector<c_type> weights = init_weights;
        unsigned int weights_len = weights.size();
        // PARAMETERS
        c_type learning_rate = step_size;
        c_type alpha = 0.0, beta = 0.0;
        c_type beta1 = beta1_init;
        c_type beta2 = beta2_init;


        if(verbose)
                std::cout << "=========== ADAM OPTIMIZER ==========\n"
                        "Epochs:\t" << epochs << "\t Learning Rate:\t" <<
                        learning_rate << "\n====================================\n";



        // iterations
        for(unsigned int epoch = 1; epoch <= epochs+1; ++epoch) {

                // Make Circuit
                Circuit c(qubits);
                make_circuit(c, weights);

                // Compute Grad for observable
                std::vector<c_type> grad = c.exp_value_grad(Obs, Obs_qubs);
```

```
                // UPDATE
                for(unsigned int grad_i = 0; grad_i < weights_len; ++grad_i) {
                        alpha = beta1 * alpha + (1.0 - beta1) * grad[grad_i];
                        beta  = beta2 * beta  + (1.0 - beta2) * grad[grad_i] * grad[grad_i];
                        learning_rate = step_size * sqrt(1.0 - pow(beta2, epoch))
                                / (1.0 - pow(beta1, epoch));
                        weights[grad_i] = weights[grad_i] - learning_rate * alpha / (sqrt(beta) + eps);
                }

                // DEBUG PRINTING
                if(verbose && epoch % 10 == 0) {
                        std::cout << "Epoch:\t" << epoch << "\t<E>: ";
                        std::cout << c.exp_value(Obs, Obs_qubs) << "\n";
                }
        }

        return weights;
}




// Gradient Descent Optimizer
// Take cost, weights --> returns optimized weights
std::vector<c_type> GD(
        unsigned int qubits,
        void make_circuit (Circuit &, std::vector<c_type> &),
        SqMatrix Obs,
        std::vector<unsigned int> Obs_qubs,
        std::vector<c_type> init_weights,

        unsigned int epochs = EPOCHS,
        c_type learning_rate = STEP_SIZE,
        bool verbose = VERBOSE
)
{
        if(verbose)
                std::cout << "========== GRADIENT DESCENT =========\n"
                        "Epochs:\t" << epochs << "\t Learning Rate:\t" <<
                        learning_rate << "\n====================================\n";

        // Parameters
        std::vector<c_type> weights = init_weights;
        unsigned int weights_len = weights.size();


        // iterations
        for(unsigned int epoch = 1; epoch <= epochs; ++epoch) {

                // Make Circuit
                Circuit c(qubits);
                make_circuit(c, weights);

                // Compute Grad for observable
                std::vector<c_type> grad = c.exp_value_grad(Obs, Obs_qubs);

                // Update weights
                for(unsigned int grad_i = 0; grad_i < weights_len; ++grad_i) {
                        weights[grad_i] = weights[grad_i] - learning_rate * grad[grad_i];
                }

                // Print Results
                if(verbose && epoch % 10 == 0) {
                        std::cout << "Epoch:\t" << epoch << "\t<E>: ";
                        std::cout << c.exp_value(Obs, Obs_qubs) << "\n";
                }
        }
```

```
        return weights;
}

#endif
```

11 Appendix A - Linear Algebra

11.1 Basic Definitions

A **Matrix** is defined as an orthogonal layout of objects (usually numbers). eg:

$$A = \begin{pmatrix} 1 & \pi & -2 \\ e & 0 & 7 \\ 3i & 9 & -\pi \end{pmatrix}$$

- The **element** in row i and column j is written as α_{ij}. eg: $\alpha_{12} = \pi$
- An array of $n \times 1$ dimensions is called **vector**. eg $\vec{v} = \begin{pmatrix} 1.0 \\ 0 \\ e \end{pmatrix}$
- The Matrix B made out of the matrix A with $\beta_{ij} = \alpha_{ji}$ is called **Transpose** of A and is denoted as $B = A^T$
- The Matrix B made out of the matrix A with $\beta_{ij} = \alpha^*_{ji}$ is called **Hermitian or self-adjoint** and is denoted as $B = A^\dagger$

Matrix addition and subtruction is done element-wise as the usual operations. If A is a matrix with dimensions $(m \times n)$ and B a matrix with dimensions $(n \times p)$, we define the **inner product** $\Gamma = A \cdot B$ as:

$$\gamma_{ij} = \sum_{k=1}^{n} \alpha_{ik}\beta_{kj}$$

11.2 Vector Spaces

11.2.1 Metric - Basis

A measure for elements X of a set is the **Euclidean Norm** (or metric) which is defined (given an inner product):

$$\|X\| \equiv \sqrt{X^\dagger X}$$

Distance function d between two elements is defined as the (positive real value):

$$d(X, Y) \equiv \|X - Y\|$$

Linear Combination of vectors $\{\alpha_i\}$ is called every produced vector of the form:

$$\beta_1\vec{\alpha_1} + \cdots + \beta_k\vec{\alpha_k}$$

A collection k of vectors is called **linearly independent** if

$$\beta_1\vec{\alpha_1} + \cdots + \beta_k\vec{\alpha_k} = \vec{0} \iff \forall i : \beta_i = 0$$

A collection of linearly independent vectors is called **basis**. A basis is called **orthonormal** iff $\forall ij, \quad \alpha_i^T \cdot \alpha_j = \delta_{ij}$.

The set of linear combinations of a basis is called **Vector Space**. Usually, we choose as basis the vectors which have a single 1 in their elements:

$$\vec{e_1} = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ 0 \end{pmatrix} \qquad \vec{e_2} = \begin{pmatrix} 0 \\ 1 \\ \vdots \\ 0 \\ 0 \end{pmatrix} \qquad \cdots \qquad \vec{e_{n-1}} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ 0 \end{pmatrix} \qquad \vec{e_n} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}$$

11.3 Tensor Product

Tensor Product (or Kronecker Product) is an operation between matrices which results in a much bigger matrix. Specifically, for two matrices $A_{m\times n}$ and $B_{p\times q}$, the result is a matrix $A \otimes B_{pm\times qn}$.

This operation is defined as:

$$A \otimes B = \begin{bmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{bmatrix}$$

or more explicitly:

$$A \otimes B = \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12} & \cdots & a_{11}b_{1q} & \cdots & \cdots & a_{1n}b_{11} & a_{1n}b_{12} & \cdots & a_{1n}b_{1q} \\ a_{11}b_{21} & a_{11}b_{22} & \cdots & a_{11}b_{2q} & \cdots & \cdots & a_{1n}b_{21} & a_{1n}b_{22} & \cdots & a_{1n}b_{2q} \\ \vdots & \vdots & \ddots & \vdots & & & \vdots & \vdots & \ddots & \vdots \\ a_{11}b_{p1} & a_{11}b_{p2} & \cdots & a_{11}b_{pq} & \cdots & \cdots & a_{1n}b_{p1} & a_{1n}b_{p2} & \cdots & a_{1n}b_{pq} \\ \vdots & \vdots & & \vdots & \ddots & & \vdots & \vdots & & \vdots \\ \vdots & \vdots & & \vdots & & \ddots & \vdots & \vdots & & \vdots \\ a_{m1}b_{11} & a_{m1}b_{12} & \cdots & a_{m1}b_{1q} & \cdots & \cdots & a_{mn}b_{11} & a_{mn}b_{12} & \cdots & a_{mn}b_{1q} \\ a_{m1}b_{21} & a_{m1}b_{22} & \cdots & a_{m1}b_{2q} & \cdots & \cdots & a_{mn}b_{21} & a_{mn}b_{22} & \cdots & a_{mn}b_{2q} \\ \vdots & \vdots & \ddots & \vdots & & & \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{p1} & a_{m1}b_{p2} & \cdots & a_{m1}b_{pq} & \cdots & \cdots & a_{mn}b_{p1} & a_{mn}b_{p2} & \cdots & a_{mn}b_{pq} \end{bmatrix}$$

And if we denote the integer division and modulo as // and %, then we have a closed formula for the elements of the resulting matrix:

$$(A \otimes B)_{i,j} = a_{i//p,j//q} b_{i\%p,j\%q}$$

(given that indexing begins from zero)

11.3.1 Eigenvalue Equations

In linear algebra, eigenvector of a linear transformation (in our case a matrix multiplication) is a non-zero vector which changes at most by a constant value, when the forementioned transformation

acts on it.

This constant is called eigenvalue and the eigenvalue equation is:

$$\hat{A}\vec{v} = \lambda\vec{v}$$

where $\hat{A}$ is the transformation, $\vec{v}$ the eigenvector and λ the eigenavlue.

For Unitary matrices, like the ones we use in quantum computing:

$$\hat{U}\left|\psi\right\rangle = e^{i2\pi\phi}\left|\psi\right\rangle$$

12 Appendix B - Probabilities & Statistics

12.1 Random Experiment

Random Experiment is called a natural process in which we cannot predict the outcome; for example the roll of a dice. The possible outcomes of an experiment may be discrete (eg roll of a dice) or continuous (the moment when an earthquake happens)

12.2 Probability Definition

When the experiment is really random and not biased, then we expect it to be somewhat "fair". For experiments with discrete spectrum, we define probability of an event as the ratio of favorable outcomes over the total number of outcomes. For example, rolling a dice the probability of ending up with an even number is $3/6=1/2$

12.3 Probability Distribution

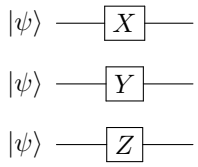
We call probability distribution the function that evaluates the probability of each outcome. Of course, its values are non-negative and the sum on its domain must yield 1.

13 Basic Quantum Gates

13.1 Signle Qubit Gates

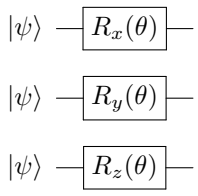
13.1.1 Pauli Gates

The most common gates are Pauli Gates. These 3 gates rotate the Qubit around the 3 axes X,Y,Z in Bloch's Sphere by 180 degrees.



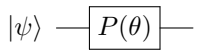
13.1.2 Rotation Gates

Apart from Pauli Gates which do 180 degrees rotation, there are the gates Rx,Ry,Rz which do controlled rotation.



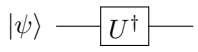
Also, a basic gate is the phase gate:

$$P(\lambda) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{pmatrix}$$



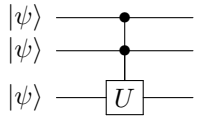
13.1.3 Miscellaneous Gates

Very often we use inverse gates to undo an operation. These gates correspond to the transpose - conjugate matrix of the initial gate.



Useful is the power of a gate. By definition: A gate is risen to a power k, when it is repeated k times.

The most general gate that can be contructed is $U = e^{ia}R_z(\beta)R_y(\gamma)R_z(\delta)$



13.2 Multiple-Qubit Gates

All single-Qubit Gates can be controlled by many Qubits.

References

[1] Frank Laloe Claude Cohen-Tannoudji Bernard Diu. *Quantum Mechanics. Volumne 1*. Wiley, 1991.

[2] I. Chuang M. Nielsen. *Quantum Computation and Quantum Information. 10th Anniversary Edition*. Cambridge University Press, 2010.

[3] Charles H. Bennett and Stephen J. Wiesner. "Communication via one- and two-particle operators on Einstein-Podolsky-Rosen states". In: *Phys. Rev. Lett.* 69 (20 Nov. 1992), pp. 2881–2884. DOI: 10.1103/PhysRevLett.69.2881. URL: https://link.aps.org/doi/10.1103/PhysRevLett.69.2881.

[4] D. Coppersmith. *An approximate Fourier transform useful in quantum factoring*. 2002. DOI: 10.48550/ARXIV.QUANT-PH/0201067. URL: https://arxiv.org/abs/quant-ph/0201067.

[5] Lidia Ruiz-Perez and Juan Carlos Garcia-Escartin. "Quantum arithmetic with the quantum Fourier transform". In: *Quantum Information Processing* 16.6 (Apr. 2017). DOI: 10.1007/s11128-017-1603-1. URL: https://doi.org/10.1007%2Fs11128-017-1603-1.

[6] Peter W. Shor. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer". In: *SIAM Journal on Computing* 26.5 (Oct. 1997), pp. 1484–1509. DOI: 10.1137/s0097539795293172. URL: https://doi.org/10.1137%2Fs0097539795293172.

[7] Lov K. Grover. *A fast quantum mechanical algorithm for database search*. 1996. DOI: 10.48550/ARXIV.QUANT-PH/9605043. URL: https://arxiv.org/abs/quant-ph/9605043.

[8] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. DOI: 10.48550/ARXIV.1412.6980. URL: https://arxiv.org/abs/1412.6980.

[9] Keiron O'Shea and Ryan Nash. *An Introduction to Convolutional Neural Networks*. 2015. DOI: 10.48550/ARXIV.1511.08458. URL: https://arxiv.org/abs/1511.08458.

[10] Ian J. Goodfellow et al. *Generative Adversarial Networks*. 2014. DOI: 10.48550/ARXIV.1406.2661. URL: https://arxiv.org/abs/1406.2661.

[11] Qi Gao et al. "Applications of quantum computing for investigations of electronic transitions in phenylsulfonyl-carbazole TADF emitters". In: *npj Computational Materials* 7.1 (May 2021). DOI: 10.1038/s41524-021-00540-6. URL: https://doi.org/10.1038%2Fs41524-021-00540-6.

[12] Tameem Albash and Daniel A. Lidar. "Adiabatic quantum computation". In: *Reviews of Modern Physics* 90.1 (Jan. 2018). DOI: 10.1103/revmodphys.90.015002. URL: https://doi.org/10.1103%2Frevmodphys.90.015002.

[13] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. *A Quantum Approximate Optimization Algorithm*. 2014. DOI: 10.48550/ARXIV.1411.4028. URL: https://arxiv.org/abs/1411.4028.

[14] Ritajit Majumdar et al. *Optimizing Ansatz Design in QAOA for Max-cut*. 2021. DOI: 10.48550/ARXIV.2106.02812. URL: https://arxiv.org/abs/2106.02812.

[15] Pierre Dupuy de la Grand'rive and Jean-Francois Hullo. *Knapsack Problem variants of QAOA for battery revenue optimisation*. 2019. DOI: 10.48550/ARXIV.1908.02210. URL: https://arxiv.org/abs/1908.02210.

[16] Vojtěch Havlíček et al. "Supervised learning with quantum-enhanced feature spaces". In: *Nature* 567.7747 (Mar. 2019), pp. 209–212. DOI: 10.1038/s41586-019-0980-2. URL: https://doi.org/10.1038%2Fs41586-019-0980-2.

[17] Maxwell Henderson et al. *Quanvolutional Neural Networks: Powering Image Recognition with Quantum Circuits*. 2019. DOI: 10.48550/ARXIV.1904.04767. URL: https://arxiv.org/abs/1904.04767.

[18] He-Liang Huang et al. "Experimental Quantum Generative Adversarial Networks for Image Generation". In: *Physical Review Applied* 16.2 (Aug. 2021). DOI: 10.1103/physrevapplied.16.024051. URL: https://doi.org/10.1103%2Fphysrevapplied.16.024051.

[19] Daniel Koch et al. *Fundamentals In Quantum Algorithms: A Tutorial Series Using Qiskit Continued*. 2020. DOI: 10.48550/ARXIV.2008.10647. URL: https://arxiv.org/abs/2008.10647.

[20] Ville Bergholm et al. *PennyLane: Automatic differentiation of hybrid quantum-classical computations*. 2018. DOI: 10.48550/ARXIV.1811.04968. URL: https://arxiv.org/abs/1811.04968.

[21] K. Mitarai et al. "Quantum circuit learning". In: *Physical Review A* 98.3 (Sept. 2018). DOI: 10.1103/physreva.98.032309. URL: https://doi.org/10.1103%2Fphysreva.98.032309.

[22] Maria Schuld et al. "Evaluating analytic gradients on quantum hardware". In: *Physical Review A* 99.3 (Mar. 2019). DOI: 10.1103/physreva.99.032331. URL: https://doi.org/10.1103%2Fphysreva.99.032331.

[23] Tyson Jones and Julien Gacon. *Efficient calculation of gradients in classical simulations of variational quantum algorithms*. 2020. DOI: 10.48550/ARXIV.2009.02823. URL: https://arxiv.org/abs/2009.02823.

[24] THOMAS SCHULTE-HERBRÜGGEN et al. "GRADIENT FLOWS FOR OPTIMIZATION IN QUANTUM INFORMATION AND QUANTUM DYNAMICS: FOUNDATIONS AND APPLICATIONS". In: *Reviews in Mathematical Physics* 22.06 (July 2010), pp. 597–667. DOI: 10.1142/s0129055x10004053. URL: https://doi.org/10.1142%2Fs0129055x10004053.

[25] Roeland Wiersema and Nathan Killoran. *Optimizing quantum circuits with Riemannian gradient flow*. 2022. DOI: 10.48550/ARXIV.2202.06976. URL: https://arxiv.org/abs/2202.06976.

[26] James Stokes et al. "Quantum Natural Gradient". In: *Quantum* 4 (May 2020), p. 269. DOI: 10.22331/q-2020-05-25-269. URL: https://doi.org/10.22331%2Fq-2020-05-25-269.

[27] Mateusz Ostaszewski, Edward Grant, and Marcello Benedetti. "Structure optimization for parameterized quantum circuits". In: *Quantum* 5 (Jan. 2021), p. 391. DOI: 10.22331/q-2021-01-28-391. URL: https://doi.org/10.22331%2Fq-2021-01-28-391.

[28] William Huggins et al. "Towards quantum machine learning with tensor networks". In: *Quantum Science and Technology* 4.2 (Jan. 2019), p. 024001. DOI: 10.1088/2058-9565/aaea94. URL: https://doi.org/10.1088%2F2058-9565%2Faaea94.

[29] Román Orús. "A practical introduction to tensor networks: Matrix product states and projected entangled pair states". In: *Annals of Physics* 349 (Oct. 2014), pp. 117–158. DOI: 10.1016/j.aop.2014.06.013. URL: https://doi.org/10.1016%2Fj.aop.2014.06.013.

[30] Guillaume Verdon et al. *Quantum Graph Neural Networks*. 2019. DOI: 10.48550/ARXIV.1909.12264. URL: https://arxiv.org/abs/1909.12264.

Index