Kvantum programozás

Jegyzet

Czégel András

2021

Ez a jegyzet a 2021 ősszel induló Kvantum Programozás kurzushoz készül. Aki bármi pontatlanságot találna, kérem írjon emailt a czegel@inf.u-szeged.hu címre.

A kurzus létrejöttéhez köszönöm a segítséget Gazdag Zsoltnak, aki befogadta a kurzust a tanszékére, és Iván Szabolcsnak, aki utat mutatott mindehhez. Ezen kívül szeretném megköszönni a jegyzet első verzióinak lektorálását Abt Dávidnak és Nagy Noéminek, akik érdeklődve és kitartóan olvasgatva megtalálták a logikai és helyesírási hibákat, amelyekben bővelkedett a nyers szöveg. Valamint köszönöm Dezső Bencének is az órák felépítésének kialakításában nyújtott segítségét, érdeklődését és kritikus kérdéseit.

Tartalom

1 Bevezetés **6**

1.1 Kvantumszámítás rövid története 6

1.2 Bevezető a tárgyhoz 7

1.3 Jelenlegi alkalmazási területek 9

2 Elméleti háttér, intuíció **16**

2.1 Fizikai intuíció 16

2.2 Matematikai alapok 18

2.3 Számítástudományi alapok 24

3 Bevezetés a programozásba **27**

3.1 A qubit fogalma 27

3.2 Python 3.6+ 35

3.3 Qiskit, qasm 37

4 Szuperpozíció és általános kapuk **38**

4.1 Szuperpozíció 38

4.2 Érmedobás 41

4.3 Fázis 42

4.4 Egyéb egybites kapuk 44

5 Több qubites számítások, összefonódás **49**

5.1 Több qubites állapotok 49

5.2 Egybites kapuk sorrendje, alkalmazása 52

5.3 CNOT kapu, kontrollált operátorok 55

5.4 Összefonódás 57

6 Univerzális kapu, áramkörök építése 60

6.1 U kapu 60

6.2 Kontrollált forgatások 64

7 Kvantum algoritmusok tervezése és írása 66

7.1 IBM Quantum Experience használata 66

7.2 Tervezési stratégiák 76

8 Kvantum gyorsítás példákon 79

8.1 Lekérdezés-modell 79

8.2 Kvantum párhuzamosítás 79

8.3 Standard lekérdezés 84

8.4 Fázis lekérdezés 84

8.5 Deutsch-Jozsa algoritmus 85

9 Nagy sűrűségű kódolás, hibajavítás 91

9.1 Teleportálás 91

9.2 Nagy sűrűségű kódolás 95

9.3 Hibajavítás, problémák 96

10 Kvantum Fourier Transzformáció 100

10.1 Fourier transzformáció 100

10.2 QFT 102

10.3 QFT implementációja 103

11 Grover algoritmusa, Fázis becslés 107

11.1 Bevezetés: Grover algoritmusa 107

11.2 Fázis becslés 114

12 Shor faktorizáló algoritmusa **118**

12.1 Motiváció 118

12.2 Algoritmus leírása 119

12.3 Implementáció, példa 120

12.4 Alkalmazások 124

Bevezetés

1.1. Kvantumszámítás rövid története

A kvantumszámítás története egészen a XX. század elejére nyúlik vissza. Az emberiség amint felfedezett valamit, igyekezett a saját szolgálataiba állítani, és ez a kvantumfizika szabályai által alkotott rendszerrel sincs másképp.

A nyolcvanas évekre egyre nyilvánvalóbb lett, hogy a természet modellezése nehéz feladat[1]. Feymannak egy mondata nagy hírnevet kapott, miszerint *A természet nem klasszikus, és ha szimulálni szeretnéd a természetet, jobban jársz ha azt a kvantummechanikával teszed.*

A 80'-as évek elején Paul Benioff megalkotta a kvantum Turing-gépet. A számítás alapjául a qubitet vették, amely egy jól definiált egységként még *létrehozásra várt*.

A kutatási kedv egyre alábbhagyott, egészen a 90'-es évek közepéig, amikor is megjelent pár nagyon sokat ígérő algoritmus: Grover kereső algoritmusa és Shor algoritmusa faktorizálásra. Mindkét algoritmus olyat állított, amit előtte nem nagyon hittek az emberek: klasszikusnál akár exponenciálisan hatékonyabb problémamegoldást.

Ennek hatására fellendültek a kutatások, újabb algoritmusokat alkottak (bár a többsége még mindig ezeken az algoritmusokon alapult), és elkezdődtek a törekvések egy valódi kvantumszámítógép létrehozására.

A 2010-es években sorra jelentek meg a kvantumszámítást ígérő eszközök. Persze csak kisebb demonstrációra alkalmas gépekként, de az első lépést megtették.

A 2010-es évek vége fele több helyen bejelentették, hogy elérték a kvantumfölényt, vagyis egy kifejezetten ennek a demonstrálására írt programot le tudtak futtatni (itt érdemes megjegyezni, hogy ennek a programnak egyáltalán nincs valós felhasználási lehetősége). Illetve megjelentek kvantumfelhők is már.

[1]Egészen pontosan a többrészecskefizikai rendszerek szimulálása NP-nehéz

1.2. Bevezető a tárgyhoz

Most hogy a történetével megismerkedtünk, nézzük inkább a gyakorlati oldalát. A kurzus alapvetően gyakorlati, nézzünk hát egy kis intuíciót a dolgok mögött. Nézzünk pár alapvető kérdést.

Mi a kvantumszámítás?

Egy másik, a megszokottól eltérő számítási modell. A kvantumfizika szabályait kihasználó számításokról van szó. A klasszikus (a megszokott, normál számítógépeket, számítást innentől "klasszikus"-ként fogjuk látni) számítások esetén a legkisebb egységünk, a bitek, lényegében az áramerősség változásával, mértékével valósulnak meg. Ehhez tranzisztorokat használunk. Jelenleg 3-5 nanométeres GAAFET tranzisztorokat.

Az egyik legfőbb ellensége a tranzisztoroknak a kvantum alagút effektus (quantum tunnelling). Ezt úgy érdemes elképzelni, hogy tegyük fel, hogy a tranzisztorok pici kapuk: vagy átmegy rajtuk egy részecske, vagy nem. Ha elég pici ez a kapu, akkor a részecskék *át*. A csukott kapun is. Jelenleg a jól ismert Moore törvényének fennmaradását, hogy 1,5 évente duplázódik a számítási kapacitás, a tranzisztorok méretének csökkenésének köszönhetjük.

Ugyanakkor a tranzisztorokat felépítő szilikonnak is van mérete, ez alá pedig az évtizedek óta használt, nagyon kedvező tulajdonságú anyag nem tud bemenni. Ez 1 nanométer körül van. Persze, lehet más anyagot keresni, és lesz is másik nagy valószínűséggel, de jelenleg a szilikon ezt a korlátot adja.

Vissza a kérdéshez: amíg egy klasszikus számítógép elkerülni szeretné a kvantumjelenségeket, addig egy kvantumszámítógép kihasználni szeretné őket. Teljesen más működési elv, de teljes számítási modell: bármilyen klasszikus számítógépen megvalósítható program megvalósítható kvantumszámítógépen is elméletben. Gyakorlatban még nem. Ugyanakkor ahogy például klasszikusan több másolatot tárolva, másolatokon számolva tudunk programozni - amit kvantumszámítógépen nem lehet, kvantumszámítógépeket állapotok szuperpozíciójával, "együttesével" tudunk számítást végezni, amit klasszikus számítógépen nem lehet.

Mi a kvantumszámítás célja?

Egyelőre sokféle. Olyan problémák megoldása, és feltevése, amiket klasszikus számítógép nem tud vagy lassabban tud megoldani. Eredetileg fizikai rendszerek hatékony szimulálásához képzelték el, de jelenleg már az optimalizálás, kommunikáció, sőt még a gépi tanulás területén is jelentek meg kvantum programok.

Cél, hogy gyorsabbat, pontosabbat alkossunk, mint amit klasszikusan el lehet érni. És ilyen gyorsításra léteznek kvantum algoritmusok, amiket látni fogunk a kurzus második felében. Persze arra, hogy általános számításra, például egy webszerver szerepére, vagy egy PC szerepére fogjuk használni, nem érdemes számítani. De nagyobb szoftverekből hívott 1-1 szubrutin, akár optimalizációs algoritmusok használatakor lehet előnye a kvantumnak.

Hogyan működik? Szuperpozíció?

"A kvantumszámítógép exponenciálisan gyorsabban tud problémákat megoldani, mivel az összes számítást párhuzamosan el tudja végezni. Például keresési problémákon egyszerre az összes lehetséges inputot ki tudja próbálni szuperpozíció segítségével".

Szinte minden újságban, cikkben, írásban szerepel. Ez egy vicc. Semmi értelme.

A szuperpozíciót pl. úgy érdemes elképzelni, hogy van 1 liter vizünk, és annyi palackunk, ahány lehetséges megoldás. Ezt az 1 liter vizet szét tudjuk osztani a palackokban, akármilyen sokban, például 500 egyliteres palackban. Ugyanakkor amikor az a kérdés, melyik megoldás a helyes, nekünk ránézésre meg kell mondanunk, melyik palackban van a legtöbb víz, és csak azt választhatjuk ki.

Persze, tévedhetünk - ahogy méréskor a programunk is! És amennyire közel lehetnek a megoldások, kellően szétosztva a vizet, szinte esélyünk sincs jó százalékkal megmondani melyikben van a legtöbb.

Persze van mód, hogy számítás közben úgy öntögessük a vizet, hogy a végén sok maradjon az egyik palackban, de ez nehéz feladat. Ilyesmiről is fogunk majd beszélni.

Mi a kurzus célja?

Szeretnénk megismerkedni az alap koncepciókkal. Hogy mi az a "doboz", az az új szempontrendszer ami alapján egy kvantumszámítógép működik. Ami alapján algoritmusokat tudunk tervezni. A doboz, amiben gondolkodni tudunk. Ebből áll a félév első fele. Intuitívan felépítjük a számítási modellt, hogy utána, a félév második felében ebben gondolkodva algoritmusokkal ismerkedjünk, implementáljunk kódokat és futtassuk őket kvantumszámítógépen is.

A cél, hogy megértsük, hogy mi működik, mi nem, mit érdemes kihasználni, mit nem. Hogy egy nagyobb szoftverben megtaláljuk, hogy mit gyorsíthatnánk kvantumosan. Hogy egy adott problémára legyen fogalmunk, hogyan is lehetne megoldani kvantumszámítógéppel.

Ugyanakkor a kurzus BSc-s szinten nehéz matematikai koncepciókat tartalmaz, mégha ezeken minimalizálva is. Tehát inkább az intuíció, szoftveres rész van előtérben. A matematikai leírások céljából készült ez a jegyzet, órán nincs előtérben a formalizmus.

1.3. Jelenlegi alkalmazási területek

Az alkalmazási területek irányába sok kutatás folyik, mind iparilag mind akadémiailag. A jelenlegi kicsi kvantumszámítógépekkel még az eszközöket magukat nehéz kihasználni, de immár az eszközök fejlesztése is gyorsan halad.

Jelenleg (2021) valódi méretű problémákra a D-Wave rendelkezik felhasználható kvantum eszközzel. Ez az eszköz nem általános célú számítógép, sokan nem is tartják épp ezért kvantumszámítógépnek. Ugyanakkor a kvantumfizika szabályait kihasználó, számításra szolgáló eszközről van szó.

1.3.1. Kutatások

Egyre szélesedő területről van szó. Egyelőre nagyon elméleti szinten, ugyanis - bár már lehet - tesztelni csak nagyon kis problémákon lehet. Informatikus szemszögből nézve pár gondolat: a területnek nagyon erős számítástudományi alapjai vannak, és a jelenleg sokszor próba-szerencse módon megírt programok, amelyekről egy programozónak

nem feladata érteni a matematikai absztrakcióját - is matematikailag vannak leírva és definiálva.

Gondoljuk csak át milyen lenne, ha papíron kellene mindenképp helyes kódot írnunk! Borzalmas lenne. Na ennyire nem rossz a helyzet, a kódunkat ki is tudjuk próbálni már! Ez nagy dolog, mégha jelenleg csak pár bittel is.

Ezért bár a kurzus amennyire engedi a terület, gyakorlati, a jegyzetben nagyon sok matematikai leírás is szerepel, jelenleg programkódok mellett.

1.3.2. Ipari fejlesztések

Nagy cégek (pl. Google, IBM), nagyhatalmak állami szervei jelentős pénzeket fektetnek kvantumszámítógépek fejlesztésébe. Különböző vállalatok különböző modellekkel. Az államiak jelentős állami / szövetségi támogatással folytatnak kutatásokat, sokszor a magáncégek segítségével. A magáncégek modellje jellemzően inkább az emberek megismertetése a kvantuminformatika világával, és bevonásuk a fejlesztésekbe.

Sok gyakornoki állást írnak ki, illetve ezek a cégek többnyire szorosan együttműködnek (sokszor inkább versenyeznek) az akadémiával. Ugyanakkor ami a mi szempontunkból fontos: széles körben elérhető és használható kvantumszámítógépeket, felhőket hoznak létre. Ezek segítségével fogunk tudni mi is megismerkedni a kvantum programozás világával.

1.3.3. Programozási nyelvek, kvantum felhők

A kvantum felhők többnyire fizetős szolgáltatások. Hasonlóan érdemes elképzelni mint szüleink-nagyszüleink idejében a lyukkártyás gépeket üzemeltető számítóközpontok - de most fizika kontakt nélkül. Másik kép lehet, például ha valaki foglalkozott kompetitív programozással, (volt pl ACM-en, spojozott), megírjuk a kódot, majd beküldjük kiértékelésre. A számítógépeken ekkor a beküldött kódunk "beáll a sorba", majd mikor az összes hamarabb beküldött ki lett értékelve, a mienk is kiértékelődik, majd az eredményeket visszaküldi a felhő.

Ehhez természetesen nekünk nem kell értenünk, pontosan fizikailag hogyan valósul meg és fut le a kódunk. Ugyanakkor ahhoz, hogy jó programot tudjunk írni, elenged-

hetetlen, hogy értsük a jelenségeket, amelyeket a gépek és ezáltal mi is ki tudunk használni. Ismét példán szemléltetve: egy (egyetemi) C kód megírásához és futtatásához nekünk nem kell azzal foglalkoznunk, hogy pontosan milyen architektúrára írjuk, pontosan a processzoron a tranzisztorok hogy helyezkednek el, és magát a számításokat hogyan hajtja végre. De azt tudnunk kell, hogy bitjeink vannak, memóriánk, bináris aritmetikánk. Hogy amíg a memória engedi, akármennyi másolatot tudunk készíteni és tárolni egy értékről.[1]

A kvantum programozás még egészen gyerekcipőben jár: sok különböző, és utasítás szintű nyelv létezik, amelyek segítségével többnyire kvantum áramköröket lehet definiálni. Ezek tehát főként interpretált nyelvből kvantum áramkörre forduló szkriptek. Ugyanakkor már létezik fordított kvantum nyelv is, illetve egy, már a C-hez egyre jobban hasonlító is - mi ezt fogjuk használni a kurzuson.

Qiskit, open qasm 3.0 (IBM)

Kezdésképp rögtön a kurzus által használt környezet. A qiskit az IBM által fejlesztett kvantum felhő által használt nyelv.

Alapvetően egy python 3 könyvtárról van szó, sok beépített függvénnyel, algoritmussal, illetve az IBM Quantum Experience-en való ingyenes futtatási lehetőséggel. Emellett lokálisan, a kvantum kódot egy szimulált interpreter segítségével is tudjuk futtatni, tesztelni.

Az IBM projektje még az open qasm. Ahogy a neve is mutatja, egy open source quantum assembly nyervről van szó, amelynek 2021-ben adták ki használatba a 3.0-ás verzióját, amely segítségével már függvényeket is tudunk definiálni. A qasm fájlokat a qiskit parsere tudja feldolgozni, majd a qiskit python részének segítségével tudjuk futtatni.

A most rendelkezésre álló legnagyobb ingyenesen használható kvantumszámítógép 15 qubites, a legnagyobb, kutatók és oktatók számára még mindig ingyenesen elérhető 65 qubites. Hogy ezek mit is jelentenek, a kurzus további részében megtanuljuk. Egyelőre örülhetünk, hogy már tudjuk futtatni, tesztelni a kódjainkat.

Jó kvantum szimulátorok is léteznek, bár szimulálni a kvantumjelenségeket klasszikusan (nem-kvantum) igen költséges. De kevés qubit esetén még bőven használható.

[1] Ezek olyan dolgok mind, ami a kvantumszámításra nem jellemző.

Cirq (Google)

Egy másik python library által interpretált nyelv a cirq. A cirq a Google fejlesztése. A Google 2020 márciusában adta ki a tfq (TensorFlow Quantum) könyvtárát. Aki dolgozott már tensorflow-val, annak ismerős lehet - kvantumszámítás folyamatosan fejlődő területén tehát a gépi tanulás is helyet kapott.

Bár a jelenlegi zajos, kicsi gépeken még egész algoritmusokat nem lehet (és ezek gyorsasága miatt nem is érdemes) futtatni, hibrid algoritmusokat már igen. Ezek az algoritmusok jellemzően a klasszikus algoritmus költséges részét szervezik át kvantumszámítógépre.

Ilyen hibrid algoritmusok a tfq modelljei is. A Google célja szerint a programozónak ne kelljen tudnia, hogy a kódja egy része kvantum számítógépen fut. Azt majd az ő motorjuk optimalizálja. Ettől ugyan még messze vagyunk, de a törekvés biztató. Emellett természetesen ezt a feladatot is meg kell oldania valakinek, akinek értenie kell nagyon alaposan ezekhez az algoritmusokhoz, és az optimalizálásukhoz is.

Silq (ETH Zürich)

A Silq-et szánja az zürichi egyetem a következő szintű kvantum programozási nyelvnek. A szintaktikája amennyire lehet, hasonlít a C-re, bár a jelölésrendszere egy fokkal szokatlanabb. A programozási nyelvet szimulátoron ki lehet próbálni, tanulni.

A nyelv maga nagyon érdekes, a technika, amivel függvényeket, szubrutinokat kezel, illetve egyszerre a klasszikus és a kvantum regisztereket is tudja kezelni. A hibrid struktúrája miatt egy fokkal nehezebben tanulható, de ha elterjed, valóban lehet egy következő lépcső, egy magasabb szintű nyelv.

Amazon Braket

A Braket az Amazon kvantum felhőszolgáltatása. Az Amazonnak saját kvantumszámítógépparkja nincs, de szerződésben állnak egyéb cégekkel. Így elérhető többféle architektúra is: szupravezető qubit alapú, ioncsapdás qubit alapú, és annealer is.

Strawberry Fields (Xanadu)

A strawberry fields-sel fotonikus áramköröket lehet építeni, gráf alapon, és azt a Xanadu eszközén futtatni. A rendszerük használatához egy fokkal több fizikai háttér szükséges, ugyanis hardverközelibb nyelvről és megvalósításról van szó. A stawberry fields is egy python könyvtár, interpretálása, futtatása a feljebb említett python alapú nyelvekhez hasonlóan történik.

Q# (Microsoft)

A Q# volt az első fordított kvantum programozási nyelv. Bár az oktatási programjuk nagyon jó (majd látjuk a kurzus során), és a nyelv integrációja jelenlegi projektekbe is nagyon jó, a Microsoft nem rendelkezik kvantumfelhővel, hogy tesztelni is lehessen élesben a kódot. Szimulátorokon lehet, ami teljesen jó, viszont ha a Microsoft szerezni sem tud kvantumszámítógépet, akkor hosszú távon nem lesz használható a nyelv.

A Q# is kvantum áramkör alapú, kapu modellt kell építeni, és az fordul le. A Microsoft környezete pedig kiváló lehetőséget tud biztosítani egyéb alkalmazásokba való beépítéséhez.

PsiQuantum

Másik innovációt jelenleg a PsiQuantum jelenti. Ők egy kis startupként kezdték, és más irányban indultak el, mint a nagyvállalatok. Nem szupravezető qubitekkel, hanem foton alapú, hibajavított számítással igyekszik előretörni.

A bennük rejlő lehetőségeket sokan észrevették, egy igen erősen támogatott kisvállalatról van szó, nagy tervekkel.

2021-ben nagyjából 200 startup foglalkozik kvantumszámítással valamilyen módon.

1.3.4. Felhasználási területek

Egész sok területen van nagy potenciál a kvantumszmításban, bár még realizálni, de még tesztelni sem sikerült. Ez persze okot ad arra, hogy sokan kételkedjenek a kvantumszámítás jövőjében, viszont a matematikai modellek kétség kívül sok reményt adnak.

Most, hogy láttuk, mennyien foglalkoznak időt és pénzt nem kímélve a területtel, lássuk mi célból is történhet mindez:

Kvantum kémia

Az egyik legfőbb potenciális felhasználási terület a kvantum kémia. Új anyagok tulajdonságainak vizsgálatához, szimulációkhoz fontos láncszem a kvantumszámítógép, ugyanis ezen a területen a szimulációk klasszikusan nagyon nehéz problémának számítanak, ugyanakkor a kvantumszámítógépek a valós rendszert modellezve sokkal hatékonyabban tudják végrehajtani a szimulációkat.

A szimulációk hatékonysága felgyorsíthatja az újabb anyagok felfedezését és felhasználását.

Fizika

Ahogy Feynman idézetéből is látható, a kvantumfizikai jelenségek hatékony modellezéséhez kvantumfizikai jelenségekkel való számítás lehet a legjobb út.

Többrészecskefizikai rendszereket szimulálni, modellezni is klasszikusan nagyon nehéz feladat. Már egész kevés részecske esetén sem lehetséges. A kvantumszámítógépek egy erőssége lehet, lényegében "hazai pálya".

A világból érkező, természetes adat analóg, fizikai. Ezt jelenleg mi feldolgozzuk, digitalizáljuk, majd a digitális adatot elemezzük. Ez persze sokkal kisebb precíziót ad, és kevesebb lehetőséget biztosít az adatok elemzésére egy digitalizálás nélküli azonnali feldolgozás helyett.

Kvantum kommunikáció

Kvantum kommunikáció, azaz a koherens kvantumrendszerek küldözgetésével való kommunikáció fejlődik. Ahhoz, hogy kvantum internet kialakulhasson, ez elengedhetetlen lépés, hiszen a kvantum állapotok pontos klasszikus "eltárolására" és küldésére nincs lehetőség.

Ez viszont nagyon sok problémát vet fel, ugyanakkor nagyon izgalmas területté teszi a kommunikációt.

Képzeljük el, hogy vannak erős számítógépeink, de azok egymással nem tudnak kommunikálni. Lefuttatunk egy programot az egyiken, és utána, hogy folytathassuk, ki kell mentenünk valamire az outputot, átgyalogolni a másik géphez, majd ezen az adaton dolgozni tovább. Ez nem túl hatékony, főleg ha "kimenteni" sem tudjuk az adatunkat.

Titkosítás

Az egyik legnagyobb hír mindig a médiában, hogy "úú a kvantumszámítógép feltör mindent". Ez így nem igaz. Bizonyos titkosítási eljárásokat *majd több év múlva* tud feltörni egy kvantumszámítógép. És léteznek már kvantumbiztos (post-quantum) titkosítási eljárások.

Ez a terület csak részben kapcsolódik a tárgyunkhoz, ugyanis ezek a protokollok klasszikus titkosítási eljárások.

Ugyanakkor ha megdőlne, hogy $P \neq NP$, ami egy jelenleg igaznak széles körben elfogadott, bizonyítatlan feltevés[1], akkor ezek az eljárások is feltörhetők lennének.

Az RSA titkosítás feltörése egy szubexponenciális időigényű problémán (faktorizálás) alapul, ez pedig exponenciálisan gyorsítható kvantumszámítógépen. Ehhez viszont többmillió qubit szükséges, ami várható 2026-2028 között válik elérhetővé leghamarabb.

Optimalizálás, adatelemzés

Egy a képzésünkhöz már sokkal közelebb álló terület az optimalizálásé, illetve adatelemzésé.

Sok hatékony klasszikus (nem-kvantum) optimalizálási eljárást ismerünk, ezekre részletesebben lesz példánk az utolsó fejezetben. Konvex optimalizálásra létezik hatékony klasszikus algoritmus. Ugyanakkor hatékonyabb(nak vélt) kvantum-eljárás is, amely a diszkrét adathalmazunkból folytonos teret készítve keresi a globális optimumot.

A főkomponens analizísre 2013-ban jelent meg algoritmus, amely jelentősen gyorsítja aszimptotikusan a klasszikus verziót.

De gépi tanulási modellek is jelennek meg mind hibrid, mind tisztán kvantum formában.

[1] Kapsz egymillió dollárt ha bebizonyítod

Elméleti háttér, intuíció

Minden számítási modellnek megvannak a sajátosságai, amik csak rá jellemzők. Ez ha a klasszikus, "általános" számítási modellt nézzük a mindennapjainkból, akkor például a memória, az adat eltárolása, másolása, digitalizálása. Ha meg akarunk írni valami célból egy kódot, akkor tudjuk, hogy mik az eszközeink, lehetőségeink arra hogy megírjuk.

A következő pár órán azzal fogunk foglalkozni, hogy a kvantumszámítás építőköveit, eszköztárát megismerjük, és kialakítsunk egy "új dobozt" amiben gondolkodhatunk. Ebben a részben még konkrétumok nélkül, intuíciót és példákat követve nézzük meg, mivel is fogunk foglalkozni, illetve az szükséges matematikai hátteret ismerjük meg.

2.1. Fizikai intuíció

Azzal, hogy a tranzisztorok binárissá alakítják a folytonos jelet, majd mi ezt beleillesztjük a gondolkodásmódunkba, egy teljesen mesterséges, digitális világot teremtünk.

A természet nem ilyen. Ha a fizikáját szeretnénk modellezni, tulajdonságait kihasználni, akkor egy erre használható eszközt kell építenünk.

A világ amit ismerünk, látunk, amiben élünk, determinisztikus. Ok-okozati összefüggéseken növünk fel, alkalmazkodunk hozzá, hogy a világ így működik. Teszünk valamit, annak pontosan meghatározható következménye lesz. (Persze ekkor a fizikai dolgokra gondolva). Determinisztikus.

Ahova mi tartunk, az nem determinisztikus. Minden valamekkora valószínűséggel következik be. Már a bitek szintjén. Minden "valamennyire" egy *és* "valamennyire" nulla. Ilyesmi jelenség a processzorokban is előfordul, de a tranzisztorok feladata binárissá alakítani a jelet, majd hibajavító kódolással a lehetséges hibákat is kijavítjuk.

Ezt viszont mi kihasználni szeretnénk. Folytonos jelekkel, állapotokkal szeretnénk számításokat végezni. Ez pedig a hibajavítást is nagyon nehézzé teszi (és nagyon intenzíven kutatott témává). Habár minden valószínűség alapú, ezt a valószínűséget is lehet számolni.

A valószínűségek mellett pedig van pár jelenség, amit a kvantumszámítógépek kihasználhatnak.

2.1.1. Kvantumfizika jelenségei

Ebből van jópár, ami még a fizikusok számára is nehézzé, elsőre befogadhatatlanná teszi a területet. Amire ki fogunk térni, azok azok a jelenségek amit ki is szeretnénk használni.

Szuperpozíció. Valószínűségek. Bitek fel tudnak venni 0-ás és 1-es értéket. Qubitek mindent is közötte. Valamekkora eséllyel. Persze, összesen ezek a valószínűségek kiadják az 1-et, hiszen nem tűnik el a qubit. Ugyanakkor egy qubit pl lehet 30% eséllyel 0 és 70% eséllyel 1. Ez csak méréskor derül ki. Addig csak valószínűségeink vannak az egyes eredményekre, egyes qubitek állapotára.

Interferencia. A részecskéink, qubitek, egymással interferálni tudnak. Mintha hullámok lennének. Fázisaik vannak, és ellentétes fázisok kioltják egymást. Klasszikusan ezt úgy lehetne elképzelni, mintha lenne egy randint(0,3) függvény, ami mondjuk itt 0 és 3 között ad véletlenszerűen egy egész számot. De 1-et meg 3-at nem adhat, mert a fázisaik ellentétesek. Mivel valószínűségekkel kell számolnunk majd, ez igen hasznos lesz.

Összefonódás. A qubitjeinket össze tudjuk fonni. Ez azt jelenti, hogy függni tudnak egymás értékétől. Például az előző randint-es példán: most két számot dob ki a generátorunk 0 és 3 között. Kidobja az elsőt, látjuk, hogy 3. Mielőtt a másodikat kidobná, mi már tudjuk, hogy az csak 0 lehet, mert össze voltak fonódva, és a másik lehetséges kimenetel már csak a 0.

2.1.2. Felhasználás, szükséges alapok

Ahhoz hogy mi mindezt alkalmazni tudjuk, szükségünk lesz egy kis matematikai recapre, és egy új gondolkodásmódra. A rendszerek fizikai megértése nem lesz fontos, de a programozáshoz szükséges toolbox, és az ebben való gondolkodás elengedhetetlen lesz.

A jelenségekhez hozzátartozik, hogy a kvantumállapotokat, azaz a programunk pillanatnyi állapotát, változóink értékét stb *nem lehet* másolni. Természetesen létre lehet hozni ismét ugyanazt az állapotot, egyszerre több példányát is el lehet készíteni, de má-

solni nem lehet.

Emellett még ezek a rendszerek "törékenyek" is, minél hosszabb számítást szeretnénk végezni, annál pontatlanabb lesz a kimenetel. És ez a pontosság (nem olyan sok) idővel 0 lesz. Ezekkel és a kezelésükkel mind megismerkedünk majd a félév során.

2.2. Matematikai alapok

Nagy levegő...

A kurzuson mély matematikai ismeret nem szükséges, de alapokkal tisztában kell legyünk. A fejezet ezen része ezeken az ismereteken vezet végig minket.

A jegyzet tartalmazza az algoritmusok pontos leírását, ami a leadott anyagnak nem része, ugyanakkor hasznos és érdekes lehet. Ehhez pedig mindenképp szükséges egy kis lineáris algebra.

Tehát fontos megjegyezni, hogy ez az anyag a kurzuson számonkérve sehol nem lesz, viszont ha kvantum algoritmusokkal szeretne valaki foglalkozni a későbbiekben, akkor elengedhetetlen.

2.2.1. Komplex számok

A számok, számrendszerek általunk kitalált, megalkotott dolgok. A természet leírásához komplex számokra is szükségünk van.

A komplex számok lehetséges alakjai:

- Kanonikus alak: $z = a + bi$ ahol a és b is valós számok
- Trigonometrikus alak: $z = r(cos(\varphi) + i\ sin(\varphi))$ ahol r a vektorunk hossza (a komplex szám abszolút értéke), φ pedig a fázisszöge, argumentuma.
- Exponenciális alak: $r \times e^{i\varphi}$, a trigonometrikus alakhoz hasonló paraméterezéssel.

Konverziók:

- $a + bi = r(cos(\varphi) + isin(\varphi)) = re^{i\varphi}$

- ekkor $r = \sqrt{a^2 + b^2}$
- és $\varphi = arctan(\frac{b}{a})$

Elképzelésük: Vegyünk egy helyvektort a síkon. Ennek a koordinátái legyenek ekkor a és b, hossza r, a vízszintestől óramutató járásával ellentétesen a szöge legyen φ.

Műveletek:

- Összeadás: $(a + bi) + (c + di) = ((a + c) + (b + d)i)$ azaz csak összeadva mindent, és összevonva
- Szorzás: $(a + bi) \times (c + di) = (ac - bd + (ad + bc)i)$ azaz beszorozva minden mindennel, mint valós számokon, és alkalmazva, hogy $i \times i = -1$.
- Konjugálás: $\overline{a + bi} = a - bi$, ha elképzeljük vektorként, ez tükrözés az x tengelyre.

2.2.2. Lineáris algebra

A számításokhoz fontos lesz pár lineáris algebrai fogalom, művelet. Két kivétellel mind szerepel a diszkrét matematika, és a közelítő és szimbolikus számítások kurzuson. Mindkét kivétel a mátrixműveleteknél lesz.

Vektorok

A skalárokról, számokról volt már szó: komplex számaink vannak. A vektorok ezeknek a számoknak egy rendezett listája. A lista hossza lesz a vektor dimenziója. A vektorok alapesetben oszlopvektorok.

Jelölések:

$$|v\rangle = \begin{pmatrix} 1+2i \\ 0 \\ 4i \\ -1 \end{pmatrix} \qquad |w\rangle = \begin{pmatrix} 3 \\ 2+6i \\ -1+2i \\ 2i \end{pmatrix} \qquad \text{"ket vé", "ket vevé"}$$

$$\langle v| = |v^*\rangle = \begin{pmatrix} 1-2i, 0, -4i, -1 \end{pmatrix} \qquad \text{konjugált transzponált, "bra vé"}$$

$$\langle v|w\rangle = 3(1-2i) + 0(2{+}6i) + (-4i)(-1+2i) + 2i(-1) = 11-4i \qquad \text{belső szorzat}$$

$$|w\rangle\langle v| = \begin{pmatrix} 3-6i & 0 & -12i & -3 \\ 8+2i & 0 & 24-8i & -2-6i \\ 3+4i & 0 & 8+4i & 1-2i \\ 4+2i & 0 & 8 & -2i \end{pmatrix} \qquad \text{külső szorzat}$$

$$|v\rangle|w\rangle = |v\rangle \otimes |w\rangle \qquad \text{tenzorszorzat, ld. műveletek mátrixokon}$$

$$\sum_i v_i w_i = \langle v|w\rangle \qquad \text{és} \qquad 5|v\rangle = \begin{pmatrix} 5+10i \\ 0 \\ 20i \\ -5 \end{pmatrix}$$

Mátrixok, mátrixok tulajdonságai

Mátrixokkal is már biztos mindenki találkozott. Pár jelöléssel, kifejezéssel azonban lehet, hogy nem.

Tulajdonságok:

- A mátrix s-**ritka** (s-sparse), hogyha minden sorában és oszlopában maximum s elem nemnulla.
- A valós számokon szimmetrikusnak hívott mátrix ($A = A^T$, $A \in \mathbb{R}^{n\times m}$, $A_{ij} = A_{ji}$) komplex megfelelője a **hermitikus** (hermitian) mátrix. Azaz a komplex mátrix megegyezik a saját konjugált transzponáltjával: $A = A^*$, $A \in \mathbb{C}^{n\times m}$, $A_{ij} = \overline{A_{ji}}$. Hermitikus mátrixok jellemzően a fizikai mennyiségeket jelölő operátorok (pl. Hamilton-operátor). Illetve szimmetrikus (és pozitív szemidefinit) mátrixok pl. a kovarianca mátrixok is. Hermitikus mátrixok sajátértékei valósak.
- És a valós számokon ortogonálisnak hívott mátrix ($A^{-1} = A^T$, $A \in \mathbb{R}^{n\times m}$) komplex megfelelője az **unitér** (unitary) mátrix. Az unitér mátrix inverze megegyezik a

konjugált transzponáltjával. $A^{-1} = A^*$, $A \in \mathbb{C}^{n \times m}$. Az ortogonális/unitér mátrixok egymással összeszorozva is unitér mátrixok maradnak, illetve vektorral beszorozva őket a vektorok hossza nem fog változni, azaz kizárólag forgatási transzformációt hajtanak végre.

- Fontos még megemlíteni a definitségi osztályokat, legalábbis egyet mindenképp: **pozitív szemidefinit** az a mátrix, amelynek minden sajátértéke nemnegatív. Ez azért fontos, mert ha ez a mátrix be van szorozva egy vektorral, akkor ezt a vektort nem fogja tükrözni semelyik dimenzió mentén. Például elképzelve két dimenzióban, egy vektort a síkon nem fog másik síknegyedbe transzformálni.
- Mátrixoknak van **kondíciószám**uk, ami számításoknál fontos dolog. A kondíciószám arányszám, ha 1 az értéke, az a legjobb. Unitér mátrixok kondíciószáma 1. A kondíciószámot alulról korlátozza a legnagyobb és a legkisebb sajátérték hányadosa.
- A mátrixoknak vannak különböző normái. Ami nekünk fontos lesz a normák közül, az az **operátornorma**. Egy mátrix operátornormája az a legnagyobb szám, amivel egy vektort a mátrix szorzáskor meg tud nyújtani. Azaz $min(\forall x \quad \|Ax\| \leq c\,\|x\|)$.

Műveletek mátrixokon

A szokásos, már ismert dolgokon kívül újdonság lehet a jelölés, illetve pár művelet, köztük a mátrix nyoma, tenzorszorzata.

Pár megjegyzés még előttük, gyakrabban használt műveletekről: A transzponálás komplex számokon a számok konjugálásával is jár. A mátrixok nagyrészt nem kommutatívak, de vannak olyanok, amik kommutálnak, jelölésük $[A, B] = AB - BA = 0$.

Diagonalizálás, invertálhatóság, mátrix rangja:

A mátrix rangja a független bázisvektorainak a száma. Hogy ez mit is jelent? Először nézzük a függetlenséget. Két vektor független, ha egymásnak nem számszorosai - és ezáltal a kettejük által kifeszített sík minden pontja előáll a lineáris kombinációjukként (azaz valami számszor az egyik vektor plusz valami számszor a másik). Ha egy 3 soros és 3 oszlopos mátrixot nézünk, akkor ha minden oszlopvektora (vagy sorvektora) független,

akkor a rangja 3. A vektorok pedig egy hatdimenziós tér bármelyik pontját tudják előállítani. A mátrixok egymástól független vektorai által alkotott rendszer a mátrix egy bázisa. Ezek a vektorok pedig a bázisvektorok. Ha ebből pont annyi van, mint ahány oszlopos a négyzetes mátrixunk, akkor nevezhetjük teljes rangú (full rank) mátrixnak, és invertálható lesz.

Ilyenkor a mátrixot fel tudjuk bontani 3 mátrix $D = PAP^{-1}$ szorzatára, ahol a D az eredeti A mátrixunk sajátértékeit tárolja a főátlójában. Ez kicsit átrendezve $A = P^{-1}DP$.

Mátrix exponenciálás Ez az anyagrész kicsit bonyolultabb elsőre, de lebontjuk. A kurzus során többször fogunk olyan kifejezésekkel találkozni, ahol exponensben mátrixot találunk: e^{-iHt}, fizika nyelvezete miatt. Ugyanis alacsony szinten az egyes rendszerek energiájának leírására használt operátor a Hamilton-operátor (hamiltonian), ami egy hermitikus, PSD[1], de nem unitér mátrix. (Kicsit előrevetítve, a bitjeinknek, ahogy klasszikusan az áramerősség mértéke dönti el az értéküket, itt az energiaszint fog fontos szerepet játszani).

No nézzük akkor mit is jelent egy olyan kifejezés, hogy e^{iA} ahol $A \in \mathbb{C}$ és hermitikus, amennyire lehet programozóbarát szemszögből.

1. Kezdésképp nézzük meg mit jelent, ha van egy függvényünk, aminek az inputja egy négyzetes mátrix és az outputja is az. Hogy is értelmezzük $f(A)$-t?

   Ez nem más, mintha vennénk A sajátértékeit, és azon hajtanánk végre a függvényt, majd összeraknánk újra A-t. Erre pedig az előző pontban néztünk is egy módszert: $f(A) = P^{-1}(f(D))P$.

2. Ezt nézzük meg egy könnyebb példán, ahol A diagonális mátrix, azaz P csak az egységmátrix és $A = D$. Ekkor ugye $f(A) = f(D)$.

   Legyen $f(x) = e^x$ és $A = D = \begin{pmatrix} 2 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 3 \end{pmatrix}$, így $f(A) = \begin{pmatrix} e^2 & 0 & 0 \\ 0 & e^{-1} & 0 \\ 0 & 0 & e^3 \end{pmatrix}$

3. Innen már kitalálható, hogy egy e^{iA} hogy áll össze. Még annyi megjegyzés, hogy a kapuink nekünk mind unitér operátorok. A komplex számok alakjából látható volt,

[1] Pozitív szemidefinit

hogy az e^{ix} egy valós x-re mindig egy egységvektort fog adni (egy 1 abszolútértékű komplex számot, az egységkör egy pontját). A hermitikus mátrixok sajátértékei pedig valós számok, ezért egy H hermitikus mátrixból e^{-iHt} unitér mátrixot csinál, azaz H sajátértékeit "szétszórja" az egységkörön.

Mátrixok tenzorszorzata: a baloldali mátrix minden elemét megszorozzuk a jobboldali mátrixszal, így két 2x2-es mátrixból 4x4-eset kapunk például. A tenzorszorzat vektorokon ugyanígy működik:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \qquad Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$$

$$X \otimes Y = \begin{bmatrix} 0 \times 0 & 0 \times (-i) & 1 \times 0 & 1 \times (-i) \\ 0 \times i & 0 \times 0 & 1 \times i & 1 \times 0 \\ 1 \times 0 & 1 \times (-i) & 0 \times 0 & 0 \times (-i) \\ 1 \times i & 1 \times 0 & 0 \times i & 0 \times 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & -i \\ 0 & 0 & i & 0 \\ 0 & -i & 0 & 0 \\ i & 0 & 0 & 0 \end{bmatrix}$$

A tenzorszorzat nem kommutatív, ahogy mátrixok szorzása sem.

Egy **mátrix nyoma** (trace) az átlójában lévő elemek összege, jelölése: $tr(A) = \sum_i A_{ii}$. Egy négyzetes mátrix nyoma egyenlő a sajátértékeinek az összegével (multiplicitással számolva).

Egy szemléletesebb példa tenzorszorzatra

Az előző rész száraz matekját kicsit gyakorlatiasabb példákban is nézzük meg. Pár matematikai pontatlanság előfordulhat a példa érdekében, de az intuíciót ez nem befolyásolja.

Mire is jó a **tenzorszorzat**? Ha van két pontunk, az meghatároz egy egyenest, ezt általános iskola óta tudjuk. Ugyanakkor mit is csinálunk amikor összekötjük a két pontot? *Kombinálunk két egydimenziós teret.* Na akkor nézzük ugyanezt arra, hogy ha van két síkunk! Két kétdimenziós sík. A két síkot meghatározhatjuk a bázisvektoraik által alkotott mátrixszal (gondoljuk pl az (1,0) és (0,1) által alkotott koordinátarendszer síkjára). Ha ezt a két síkot összeszorozzuk, kapni fogunk egy harmadik (maximum) 2 dimenziós

síkot. Ugyanakkor ha tenzorszorozzuk őket, akkor megkapjuk a két sík "kombinálásával" keletkező 4 dimenziós hiperterünket.

Na ezt ismerősebb adatszerkezetekkel: Ha van egy listánk egész számpárokból A = (List(**Tuple**(int, int)) és egy másik ugyanilyenünk B = (List(**Tuple**(int, int)) akkor a keresztszorzatuk A $\times$ B ezeket pont összekobinálja nekünk minden lehetséges módon, tehát kapunk egy olyat, hogy List(Tuple(**Tuple**(int, int), **Tuple**(int, int))), azaz 2x2 számpárból álló elemeink lesznek. Ezt csinálja több nyelven (pl Pythonban is) a `zip` parancs.

Akkor most tegyük fel, hogy van egy függvényünk, amit A-ra írtunk, és számpárok listáját várja, és másik számpár listát ad vissza. Mondjuk egy `ketszeres()` függvény szorozza be kettővel a kapott számpárok első elemét, és mínusz kettővel a második elemét.

És tegyük fel, hogy van egy `otszoros()` függvényünk, amit B-re írtunk és beszorozza a kapott számpárok listáján a számpárok első felét 5-tel, másik felét -5-tel.

Mit tegyünk, ha mi ezt a két függvényt *egyben* szeretnénk megvalósítani?

Írhatunk pl egy függvényt, ami várja mindkét listát, és végrehajtja a fenti függvényeket egymás után (vagy épp párhuzamosan). De ha nézzük ezt a függvényt, ez már együtt várja a két listát, és együtt is adja vissza őket. Egyszerre végez rajtuk műveletet, egymástól függetlenül.

Ha a `ketszeres()` és az `otszoros()` mátrixok lennének, operátorok, amikkel 1-1 qubitre hatunk (a listáink helyett), akkor a tenzorszorzatuk pont a két qubiten egyszerre ható operációt fogja leírni, ahol az első qubiten a kétszerezés, a másodikon az ötszörözés fog végrehajtódni.

2.3. Számítástudományi alapok

A qubit alapú kvantumszámítás Turing teljes szémítási modell. Ez azt jelenti így elsőre, hogy minden klasszikusan implementálható problémát kvantumszámítógépre is lehet implementálni. Univerzális számítógép a kvantumszámítógép is.

Persze a számítási mód teljesen más, viszont ugyanaz az absztrakció, mint a klasszikus számításnál. Ugyanakkor ez az absztrakció, amit formális nyelveken, bonyultságelméleten, számítástudomány alapjain tanulunk kifejezetten a klasszikus modellre jött létre.

2.3.1. Klasszikus és Kvantum számítási modell

Számítási modellbeli különbségek már az első óra anyagában is feltűntek. A Turing-gép, illetve RAM gép leírására **ez** a jegyzet a 9. oldalától kiváló.

Bár kvantumszámításra is létrehozták a saját "Turing-gépét", ez nem túl széles körben használt, ugyanis az algoritmusok nagyrésze matematikai leírásból jön létre még mindig, azaz nem kísérletezve, nem nagyobb szoftverekben. A matematikai leírás pedig a saját szabályrendszere mellett könnyen ellenőrizhető.

Ugyanakkor az algoritmusok bonyolultságának vizsgálata több problémát vet fel.

2.3.2. Bonyolultságelmélet

A klasszikus tár- és időbonyolultsági definíciók nem igazán állják meg a helyüket a kvantumszámítás területén.

A tárbonyolultság még csak-csak, viszont egy információhalmazt többféle módon lehet elkódolni qubiteken, amelyhez a qubitek mennyisége eltérő, erről majd lesz szó a 9. fejezetben. De alapvetően ez a modell még jellemzően megegyezik a klasszikussal.

Ugyanakkor az időbonyolultság kicsit eltér. Létezik a szekvenciális modell is, de sok kvantumalgoritmus előnye nem mutatkozik meg egy szekvenciális modellen. Ekkor a számítási mennyiséget mérjük, konyhanyelven a kapuk számát, az input méretével arányosan.

Itt viszont belefutunk abba, hogy mit nevezünk egy operációnak? Egy műveletnek? Ha egy olyan művelet az egység, ami pont sokkal hatékonyabb egyik architektúrán, mint a másikon, akkor mi történik?

Itt jön szembe a kvantumszámításban szélesen használt **lekérdezés-modell** (query complexity). Ehhez feltételezzük, hogy van egy adatbázisunk, kulcs-érték párokkal, olyan formában, hogy a kulcs egy n-bit string, az érték pedig egy bit. Ehhez lehet létrehozni egy olyan műveletet, amely végrehajt egy lekérdezést. De ilyen lekérdezést egyszerre több bit szuperpozícióján is lehet végrehajtani.

A lekérdezés-bonyolultság lényege, hogy a lekérdezések számában mérjük az időbonyolultságot. Egy lekérdezés a művelet, az egység. Ez persze megint kérdéseket vet fel,

mert mellette a többi műveletet nem számoljuk (általában nagyságrendileg kisebb, mint a lekérdezés).

2.3.3. Hibajavítás, kódoláselméleti intuíció

A hibajavítás kimarad a kurzusból, mert már nem férne bele, ugyanakkor fontos eleme a területnek. Viszont annyiból perem, hogy szoftverek írásakor most sem foglalkozunk bithibák javításával. Most egy igen sokak által kutatott terület, hogy kvantum hardveres hibajavítást hozzunk létre, és ezután ismét nem lesz olyan mindennapi szempont a szoftverfejlesztés során.

A kvantumszámítógépek nagyon érzékeny rendszerek. Gondoljunk bele, -273 celsius fokon ezredfoknyi eltérésekkel már elromolhat a számításunk. Túl sokáig sem tarthat egy számítás.

Ehhez sokféle módszer létezik. Mivel valami > 0.5 értékkel jó eredményt kapunk sok esetben, elég sokszor megismételni a számítást. De ez a közelítés sem jó mindig, és nem is hardveres.

Hardveresen a qubitek fizikai elrendezéséből adódó struktúráltságot, illetve a qubitek tulajdonságait igyekeznek kihasználni. Illetve az algoritmusok pontosságát is javítani pontosabb, ekvivalens kapusorozatokra való konverzióval, majd ennek a futtatásával.

Qubitek megvalósulhatnak többféle módon is, fotonokból, csapdázott ionokbók és szupravezető áramkörök segítségével is például, ezeknek a hibái pedig különféle módon "kezelhetők".

Bevezetés a programozásba

3.1. A qubit fogalma

A qubit fizikailag sokféle módon megvalósítható - és sokféle módon meg is van valósítva. Itt nem erről lesz szó, hanem arról, hogyan tudjuk elképzelni, és milyen tulajdonságai lehetnek.

Egy qubit hasonlít egy bithez, annyiban, hogy lehet 0 vagy 1. Viszont lehet a qubit bármi a kettő között is. Lehet például félig 1, félig 0. És lehet például 25% eséllyel 1 és 75% eséllyel 0.

Tehát ezek a qubitek számítás során folyamatosan változnak, 0 és 1 között felvesznek értékeket, majd amikor ki szeretnénk olvasni az adatot belőlük, összeesnek, és a végén vagy 0 lesz belőlük mérés során, vagy 1.

A qubiteknek ezen kívül fázisuk is lehet, egymással tudnak "kommunikálni". Ezzel majd később foglalkozunk.

Elsőre érdemes úgy elképzelni, mint egy vektort, ami az egységkör egy pontjára mutat. Viszont mi csak a tengelyeken tudjuk észlelni. Tehát amint megmérjük, odaugrik valamelyik közeli tengelyhez, és oda fog mutatni.

Például ha ez egy 30°-os szöggel mutatna az $x = \frac{\sqrt{3}}{2}$, $y = \frac{1}{2}$ pontra, akkor mérés után 3/4 eséllyel mérnénk az x tengelyre (mondjuk 0-nak) és 1/4 eséllyel az y tengelyre (mondjuk 1-nek). Ha elképzeljük ábrán, akkor látható is, hogy háromszor messzebb van az y tengelytől, mint az x-től.

Különböző, látványosabb reprezentációkat is fogunk nézni nemsokára.

3.1.1. Reprezentációk

Alapvetően kétféle módon látjuk majd reprezentálva a qubiteket, és az állapotaikat. Vektorokként, illetve braket jelöléssel. A kettő ekvivalens, átírhatók egymásra.

Mivel a qubiteknek fázisuk is lehet, ezért az előbbi egységkörös történetet át kell tennünk térbe: egységgömbünk lesz. A mérés logikája nem fog változni.

Vektor reprezentáció

Az első reprezentáció vektoros. Ez talán könnyebben átlátható, de nagyobb rendszerek esetén már problémás tud lenni ha kiírnánk a vektorokat. Lássuk.

Az előző fejezetben láttuk, hogy egy állapot jelölése lehet $|0\rangle$ például.

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Ők egy bázist alkotnak. Merőlegesek egymásra. Különböző bázisokat is tudunk alkotni, ld. Bloch gömbös pont, illetve következő fejezet.

Bitből több is lehet egy rendszerben persze. Míg klasszikusan ha több bitünk van, akkor nekik a tartományukat keresztszorozzuk, őket pedig egymás mellé írjuk, addig a qubitekét tenzorszorozzuk.

Azaz két bitet klasszikusan reprezentálhatunk 00, 01, 10, 11 módon. Qubiteken kicsit máshogy néz ki:

$$|0\rangle \otimes |1\rangle = \begin{bmatrix} 1 \times 0 \\ 1 \times 1 \\ 0 \times 0 \\ 0 \times 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = |01\rangle$$

Pl. ő lenne az "1", mert 01 az 1 bináris alakja. Persze a 000 már egy 8 hosszú vektor lenne, aminek az első eleme 1, többi 0. Érdemes észrevenni, hogy minden ilyen vektor merőleges az összes többire. Így lesznek ők n qubitre a 2^n dimenziós euklideszi tér bázisvektorai, a számításunk alapja.

A vektorok elemei lehetnek komplex számok, de a vektorok hossza 1 kell legyen. Amíg klasszikusan minden diszkréten volt ábrázolva, azaz lényegében csak a "rácspontokon" voltak értékeink, addig kvantumszámítás során az egész térben lehetnek pontjaink, tehát az intervallum folytonos.

Braket reprezentáció

Persze nem csak bázisállapotaink lehetnek, vehetünk például egy qubitet úgy, hogy egyenlő eséllyel legyen 0 és 1. Így qubitek felvehetnek bármit 0 és 1 között, nekünk viszont tudnunk kell ezt jelölni is.

Ezt tudjuk úgy jelölni, hogy minden bázisállapotot megszorzunk az amplitudójával[1]. Ha visszatérünk a körös példához, és a $x = \frac{\sqrt{3}}{2}$, $y = \frac{1}{2}$ pontra mutató helyvektorhoz, akkor láthatjuk, hogy bizony x+y nem "jön ki" 1-re. Pedig azt várnánk, hogy az összes lehetséges eset összes esélyének összege 1 legyen (azaz teljes eseményrendszert alkossanak).

Viszont azt beláttuk, a távolsága a tengelyektől $\frac{3}{4}$ és $\frac{1}{4}$. Mi az összefüggés? Ezek távolságok. A körön. A tengelyen mért értékek *négyzetét* kell vennünk, ha a valószínűségeiket szeretnénk megkapni.

Tehát újabb fontos megállapítást tehetünk: a bázisállapotok amplitudóinak a négyzeteinek összege 1. És ezt hogy jelöljük? Kell egy jelölés, amivel tudjuk azt jelölni, hogy egy állapot *mennyire 0* és *mennyire 1*.

Az előző példán nézve ezt tudjuk jelölni így:

$$\frac{\sqrt{3}}{2}\left|0\right\rangle \; + \; \frac{1}{2}\left|1\right\rangle$$

Hogy ez mit is jelent vektorokkal leírva? A vektorok elemei az amplitudók. Ezek komplex számok is lehetnek, de egyelőre maradjunk a valósaknál:

$$\frac{\sqrt{3}}{2}\left|0\right\rangle \; + \; \frac{1}{2}\left|1\right\rangle \quad = \quad \begin{bmatrix} \frac{\sqrt{3}}{2} \\ \frac{1}{2} \end{bmatrix}$$

[1]A különböző bázisállapotoknak amplitudója (magassága) van: ez egy szinuszoid hullám magasságát jelenti. Itt nekünk ez az amplitudó határozza meg, hogy egy szuperpozícióban, több állapot együttes rendszerében mennyire valószínű, hogy az adott bázisállapotot fogjuk kapni méréskor (ld. mérés).

Példa

Most nézzünk egy másik példát, 2 qubiten, hogy ez mit is jelent. Legyen az egyik qubit biztosan 1, másik qubit pedig legyen qubit 50% eséllyel 0, 50% eséllyel 1. (Azaz legyen egyenlő szuperpozícióban, ahogy majd később hívjuk). Ilyenkor hogy tudunk vele számolni? Mi az állapota ennek a *2 qubites rendszernek*?

Először is kelleni fognak az egyenlő esélyben lévő qubitünk amplitudói. Ez ugye az 50%, azaz az $\frac{1}{2}$ gyöke, tehát $\frac{1}{\sqrt{2}}$. Mindkét bázisállapoté, mivel egyenlők.

Vektorokkal felírhatjuk a két qubitet egyesével, vehetjük a tenzorszorzatukat, és meg is kapjuk az állapotunkat:

$$\begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \otimes \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \times 0 \\ \frac{1}{\sqrt{2}} \times 1 \\ \frac{1}{\sqrt{2}} \times 0 \\ \frac{1}{\sqrt{2}} \times 1 \end{bmatrix} = \begin{bmatrix} 0 \\ \frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \end{bmatrix}$$

Vegyük észre, hogy az első qubit került hátra, tehát az "első" qubit a legkisebb helyiértékű helyre kerül.

Nézzük ez a példa mit jelent braket jelöléssel, bázisvektorok összegeként (lineáris kombinációjaként). Tehát tudjuk, hogy a hátul lévő qubit mindenképp $|1\rangle$. Az elöl lévő pedig $\frac{1}{\sqrt{2}}$ amplitudóval $|0\rangle$ és $\frac{1}{\sqrt{2}}$ amplitudóval $|1\rangle$.

$$0\,|00\rangle \;+\; \frac{1}{\sqrt{2}}\,|01\rangle \;+\; 0\,|10\rangle \;+\; \frac{1}{\sqrt{2}}\,|11\rangle \quad =$$

$$\frac{1}{\sqrt{2}}\,|01\rangle \;+\; \frac{1}{\sqrt{2}}\,|11\rangle \quad =$$

$$\frac{|01\rangle \;+\; |11\rangle}{\sqrt{2}}$$

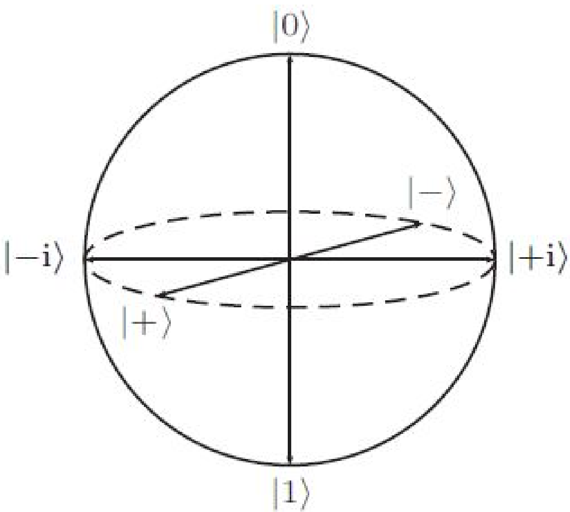
A két jelölés közötti kapcsolatként észrevehetjük, hogy az eredmény vektor elemei pont a bázisállapotok együtthatói a braket jelölésben. Legtöbbször a vektorban nagyon

sok nulla elem van, ezért is könnyebb a braket jelölést használni.

3.1.2. Bloch gömb

Volt arról szó, hogy a qubitet egy gömb felszínére mutató vektorként is el lehet képzelni. Nos, a Bloch gömb pont ezt teszi. Ugyanakkor 3 dimenziós objektum, 3 tengelye van. Mindhárom tengely 1-1 bázisnak felel meg.

Ennek ábrázolásásra a Bloch gömb így néz ki:



Bázisok a Bloch gömbön

Az eddig látott $|0\rangle$ és $|1\rangle$ által alkotott bázis a **Z** bázis, vagy számítási bázis. Legtöbbször ezt fogjuk használni, és általában ez is a szokás, ezért is számítási bázis a neve.

Ugyanakkor van erre merőleges bázis is, ahol a bázisvektorok, ha az amplitudókat nézzük, akkor i és $-i$[1]. Ez talán a legritkábban használt bázis, ez az **Y** bázis. Bázisvektorai braket jelöléssel a $\frac{1}{\sqrt{2}}(|0\rangle + i\,|1\rangle)$ és a $\frac{1}{\sqrt{2}}(|0\rangle - i\,|1\rangle)$. Ezek itt épp pont az előjelben különböznek. Ez a későbbi számításokban fontos lesz.

Emellett van egy harmadik bázis is, ami merőleges mindkettő eddigire a gömbön, ez pedig az **X** bázis, vagy Hadamard bázis. Bázisállapotai a $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ és a

[1] Az i az a komplex i, azaz gyökmínuszegy.

$|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$. A bázisok között váltásról a következő fejezetben bővebben is lesz szó.

3.1.3. Kapuk egy qubiten

A kapukat többféleképpen is el lehet képzelni. A megvalósítások architektúránként nagyon eltérők. Gondoljunk csak bele, például mechanikus számológépből milyen sokfélét alkotott meg az emberiség. A számítás lényege, hogy a kapukkal *hatni* tudunk egy vagy több qubitre.

Ugyanúgy, ahogy egy klasszikus számítógépen megszokhattuk, például egy `NOT` kapu esetében, itt is lehet például negálni a qubitünk értékét. Ezt a *ráhatást* elképzelhetjük úgy is, hogy csak szimplán utasításba adjuk a számítógépnek, hogy negálja a bitet.

Az első kapu amivel megismerkedünk, az tehát az **X kapu**. A klasszikus negálással ellentétben viszont egy kvantum kapunak több dolgot is teljesítenie kell: eleve a qubitek nem csak $|0\rangle$ vagy $|1\rangle$ állapotban lehetnek, hanem ezeknek bármilyen szuperpozíciójában is. Sőt, fázisuk is van, ami a bázisállapotoktól függetlenül számítási lehetőségeket rejt. Tehát a kapunak *kizárólag* a két bázisállapotunk között kell váltania, megőrizve minden egyéb tulajdonságot.

Minden n qubites kapunak van egy $2^n \times 2^n$ mátrixa. (Előző fejezetben láthattuk, hogy egy n qubites rendszer egy 2^n hosszú komplex vektorral írható le, mert tenzorszorzáskor a vektorok elemszámai összeszorzódnak, így ez se olyan meglepő.) Minden kapunak a mátrixa unitér, vagyis 1-1 forgatásnak felelnek meg a hipertérben. Ezen kívül a sorvektoraik/oszlopvektoraik merőlegesek egymásra, és a hosszuk 1.

Tehát az X kapu pont azt éri el, amire gondolnánk: a 0 és az 1 állapot között "vált". Ugyanakkor egy qubit nem csak tisztán 0 vagy tisztán 1 lehet, hanem ezek bármilyen szuperpozíciója is. Ha elképzeljük a Bloch gömböt, akkor a $|0\rangle$ állapot a Z tengelyen (függőlegesen) egy felfele mutató egységvektor. A $|1\rangle$ pedig lefelé mutató. Ebből így sejthető, hogyha minden kapu 1-1 forgatás a bloch gömbön, akkor az X kapu is az lesz: az Y tengely körül π radiánnal (azaz ez a gömbön függőlegesen egy $180°$-os elforgatás). Érdemes észrevenni, hogy bár a kapu neve X (vagy `NOT`), az Y tengely körül forgat.

$$\text{Az X kapu mátrixa: } \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

$$\text{Például: X}\,|0\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle$$

Van egy azonosság kapu is, azaz **I kapu** (Identity). Ez nem csinál mást, csak úgy hat a qubitre, hogy azzal semmi ne történjen. Mátrixa értelemszerűen az egységmátrix.

Következő kapunk a **H kapu**. H, mint Hadamard. Ez a kapu kicsit összetettebb operációt valósít meg: a bázisok között segít váltani. A tisztán $|0\rangle$ állapotból a $\frac{|0\rangle+|1\rangle}{\sqrt{2}}$ vagyis a $|+\rangle$ állapotba viszi át a rendszert. Ez a $|0\rangle$ és $|1\rangle$ állapotok egyenlő szuperpozíciója. A kaput kétszer alkalmazva visszakapjuk az eredeti állapotunkat. A Hadamard kaput nagyon sokszor fogjuk használni a továbbiakban.

$$\text{A H kapu mátrixa: } \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

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

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

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

A következő kapunk a **Z kapu**. A Z kapu pontosan ugyanazt a transzformációt hajtja végre, mint az X kapu, de az X tengelyen. $|0\rangle$-ból $|0\rangle$-t, $|1\rangle$-ből $-|1\rangle$-et csinál. Illetve pl. a $|+\rangle$-ból $|-\rangle$-t - ugyanis pont azon a tengelyen "negál". Megjegyzés: a -1-es amplitudó méréskor ugyanúgy 1 valószínűséget ad, mert $(-1)^2 = 1$. Ez a mínuszjel a qubit *fázisát* határozza meg, nem a bázisállapot amplitudóját.

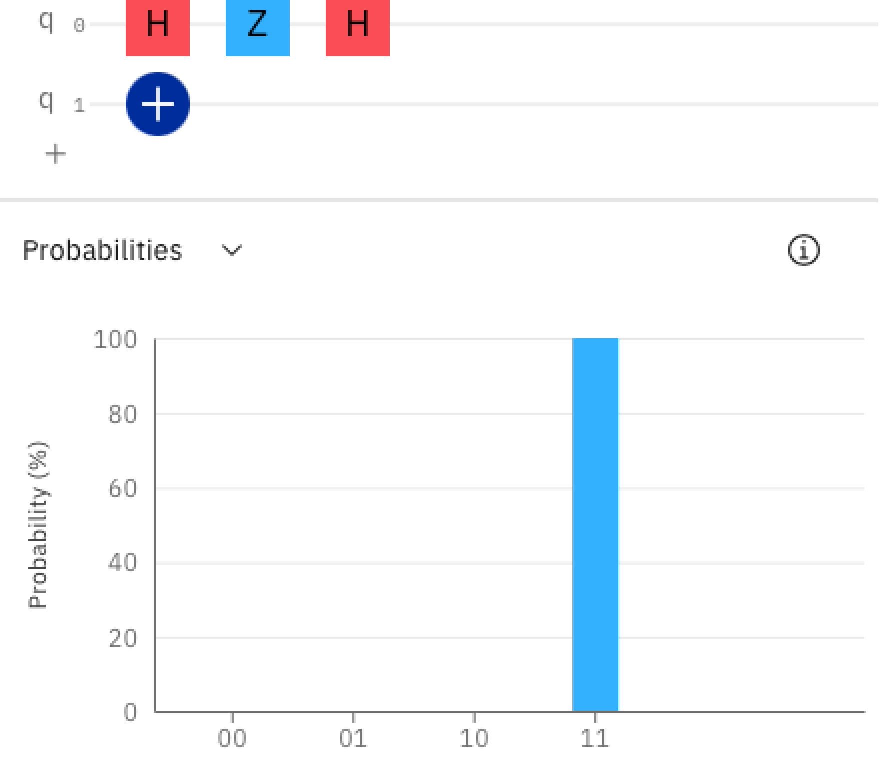
$$\text{A Z kapu mátrixa: } \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

$$\text{Például: } Z\,|1\rangle \;=\; \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} \;=\; \begin{bmatrix} 0 \\ -1 \end{bmatrix} \;=\; -\,|1\rangle$$

Kicsit kísérletezzünk pár ténnyel, amit eddig láttunk: i) a Hadamard kapu kétszeri alkalmazása ugyanaz, mintha az I kaput használtuk volna (oda és vissza a számítási és a Hadamard bázis között), illetve ii) a Z kapu a Hadamard bázisban negál, számítási bázisban pedig a $|1\rangle$ elé tesz egy mínuszjelet.

Feladat: építsünk olyan áramkört 1 biten, ami az X kapu hatását éri el, az X kapu használata nélkül!

Megoldás: Ötlet: váltsunk Hadamard bázisba, negáljunk a Z kapu segítségével, majd váltsunk vissza. $\mathrm{HZH}|0\rangle = \mathrm{X}|0\rangle = |1\rangle$. Ki lehet számolni, de inkább nézzünk egy képet a környezetből amivel dolgozni fogunk:



Az utolsó kapunk itt az **Y kapu**. Az Y kapu a harmadik tengely körül forgat π-vel.

$$\text{Az Y kapu mátrixa: } \begin{bmatrix} 0 & i \\ -i & 0 \end{bmatrix}$$

3.1.4. Mérés

Az algoritmusunk lefutása után szeretnénk valamilyen eredményt kapni. Tehát le kell olvasni a qubitek állapotait. Ez nem olyan egyszerű dolog. A qubiteknek van egy természetes valószínűség alapú állapotuk, amivel számolunk egy algoritmus futása során ("mennyire 1? mennyire 0?"). Ez méréskor elveszik. Viszont az egyes bázisállapotok amplitudói (a "mennyire" 1) alapján kapunk majd egy értéket, a bázisállapotok egyikét, azaz a mi számítási bázisunkban a 0 vagy az 1 klasszikus bitet.

Tehát ha van mondjuk 3 qubitünk, amihez tartozik 8 bázisállapot, a számításaink során lehetnek bármilyen szuperpozícióban, de méréskor egy 3 hosszú klasszikus bitstringet fogunk kapni. Ha vektor reprezentációval számolunk, akkor könnyű a dolgunk: a bázis-bitstringek mérési valószínűségei a vektorunk elemeinek a négyzete, azaz az egyes bázisállapotok amplitudójának a négyzete. Ha braket reprezentációt használunk, akkor pedig eleve a lehetséges állapotok amplitudóival számolunk, így ott is csak az adott állapotok együtthatóinak (amplitudóinak) a négyzetét kell figyelnünk.

A kurzus során a Z bázist, azaz a számítási bázist fogjuk használni. Ugyanakkor a bloch gömb bármely tengelyét lehet mérési bázisnak használni, az X tengelyt (Hadamard bázis) és az Y tengelyt is. Itt észrevehetjük, hogy ha van egy qubitünk a $|0\rangle$ állapotban, akkor az a Z bázisban méréskor 100% eséllyel 0 lesz, ugyanakkor a Hadamard (X) bázisban ez pont a két alapállapot egyenlő szuperpozíciója, azaz $\frac{|+\rangle+|-\rangle}{\sqrt{2}}$, tehát méréskor egyenlő eséllyel kaphatjuk meg a két bázisállapot egyikét, azaz a $|+\rangle$-t *vagy* a $|-\rangle$-t.

3.2. Python 3.6+

Az órákon nagyrészt python libraryket használunk, illetve az IBM rendszereit. Python programozáshoz a Szkriptnyelvek, illetve a Python programozás a gyakorlatban kurzusok elvégzése ajánlott, de nem kötelező. A kurzus algoritmikus gondolkodást elvár, de mély

nyelvismeretet nem.

A Python azért volt optimális választás kvantum könyvtárakhoz, mert az egyik legkönnyebben megtanulható nyelv, és így programozásban teljesen kezdők is használni tudták. Ezután elterjedt. Forduló nyelv pedig csak egy létezik jelenleg szélesebb használatban, ez a Q#.

3.2.1. Szkriptnyelv

Tehát interpretált nyelvünk van, az utasításokat sorról sorra végrehajtja az értelmező. Lényegében akár fel is sorolhatjuk hogy mit szeretnénk végrehajtani. A nyelvről és tulajdonságairól bővebb ismertető olvasható a fenti két kurzus anyagában a Szoftverfejlesztés Tanszék oldalán.

3.2.2. Qiskit library

A kurzus a Qiskitet használja. A Qiskit az IBM által fejlesztett Python library. Folyamatosan és intenzíven fejlődik, változik, mint ahogy kvantumszámítás is, ezért frissen kell tartani, a kódok pedig gyorsan elavulnak, mégha ez nem is túl szerencsés.

Az IBM rendszereinek az eléréséhez szükséges pár dolog:

- Regisztráció az IBM Quantum oldalán.
- A regisztrációs email fontos, végig ezt fogjuk használni. Nem szükséges studos email, lehet privát is. Regisztráció után az oktató tudja hozzáadni a felhasználót a hubhoz, ahol a kurzus is található.
- A lokális telepítéshez a Qiskit dokumentációja segít.

Az oldal és a rendszer használata intuitív. A kurzus során használatban lesz a **circuit composer**: itt áramköröket fogunk építeni, illetve vizsgálni. Alul állapotvektorként és valószínűségekként, vizuálisan is láthatjuk az áramkörünket. Jobb oldalt a hozzá tartozó Qiskit, illetve qasm 2.0 kódokat láthatjuk. A megépített áramkört pedig az IBM eszközein futtatni is tudjuk. A másik fő része amit használni fogunk, az a **quantum lab**. Itt jupyter notebookokat tudunk létrehozni, és algoritmusokat írni. Kicsit egyszerűbb, mint lokálisan feltelepíteni a Qiskitet.

3.3. Qiskit, qasm

A Qiskitnek sok modulja van, edukációs és kutatási céllal. Ezek folyamatosan változnak. A Qiskit jó keretrendszer az algoritmusok vezérlésére, konfigurációjára, futtatására. A Python megfelelő a vezérlésre, modulok építésére, programozásra.

Az qasm pedig az áramkörök építésére. A Qiskitben jelenleg (2021) a 2.0 a kiadott hivatalos verzió, 2022-től már a 3.0 lesz. Nagy különbségek, hogy a 3.0 már turing teljes nyelv lesz, vezérlési szerkezetekkel, paraméteres függvényekkel. Ez a 2-ben még nincs meg, de áramkörök építésére kényelmes.

3.3.1. Opensource quantum assembly (3.0)

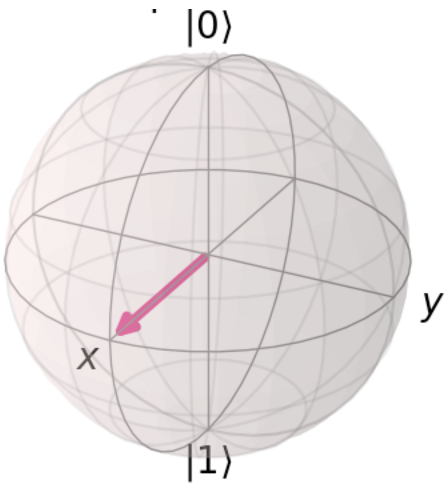
Már a C-hez egy fokkal közelebb álló nyelv lesz. A nyelvtana elérhető már nyilvánosan, antlr segítségével lehet Pythonnal és Qiskittel használni, jelenleg béta verzió. Még minden vezérlést a Python lát el a Qiskitnél, ez változni fog. Egész algoritmusokat lehet majd építeni és futtatni az open qasm 3.0 segítségével.

Szuperpozíció és általános kapuk

4.1. Szuperpozíció

Foglalkozzunk kicsit a szuperpozícióval és a Hadamard kapuval. Az X és Z kapuk klasszikusan könnyedén értelmezhető műveletet hajtanak végre: negálnak. Bizonyos bázisokban. Csak X és Z kapuk használatával mérés előtt teljes biztossággal meg tudjuk mondani a mérés kimenetelét (mérési hibától eltekintve).

Foglalkoztunk a Hadamard kapuval, ami átvált számítási bázisból a Hadamard bázisba - de mi történik ha méréskor mi maradunk a számítási bázisnál? Kis emlékeztetőképp a Bloch gömbön ábrázolva ez az állapotunk:



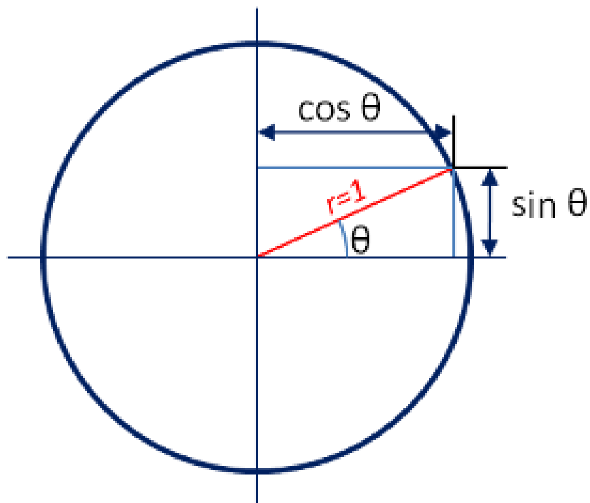
Méréskor ilyenkor kizárólag a két bázisállapot egyikének tudjuk mérni az állapotunkat: 0-nak vagy 1-nek. Az, hogy melyiknek mekkora az amlpitudója, és ezáltal a valószínűsége, a vektorunknak a Z (függőleges) tengellyel bezárt szöge határozza meg.

Itt sajnos kicsi kavarodással kell számolnunk: a bázisállapotok a valóságban derékszöget zárnak be egymással. Ez sok fontos tulajdonságot határoz meg, például az egymással vett vektorszorzatuk nulla, lineáris kombinációik az egész n-dimenziós (Hilbert) tér standard bázisát alkotják stb. A Bloch gömbön viszont egymással szemben, egy tengelyen helyezkednek el ezek. Így amikor a vektorunk számítási bázisban vett, bázisállapotokhoz való szögét szeretnénk meghatározni, a Bloch gömbön a függőleges tengellyel

bezárt szöget kettővel le kell osszuk. Ne zavarjon össze senkit a dolog, csak egy felezésről van szó.

Mekkora is a szög? Nos, a Bloch gömbön π radián mindkét bázisállapothoz (azaz 90-90°, de szokjuk, hogy radiánban kell számolnunk!). Tehát a kérdéses szögünk $\theta = \frac{\pi}{4}$ mindkét bázisállapothoz[1]. Ezzel megkaptuk azt a szöget, amiből az amplitudót és így a valószínűséget is ki tudjuk számolni.

A szögünk megvan, ebből nekünk egy szám kell. Az a kérdés, az állapotunk milyen messze esik a tengelyektől. Minél közelebb, annál nagyobb eséllyel mérjük majd abban az állapotban. Ehhez szükséges egy kis trigonometria:



Az egységkörön mozogva, a szögből a távolságokat kiszámolva megkapjuk a bázisállapotok amplitudóit. A függőleges tengely jelöli a 0-t, a vízszintes az 1-et. Vezessük be a jelenleg egy qubitből álló rendszerünk állapotára a $|\psi\rangle$-t, mint jelölés. Innentől a rendszerünk aktuális állapotát ezzel fogjuk jelölni.

Nézzük a jelöléseket, és helyezzük el a példánkat:

$$|\psi\rangle \;=\; \cos(\theta)\,|0\rangle \;+\; \sin(\theta)\,|1\rangle$$

Megj.: egyelőre a fázistól eltekintünk, a fejezet végén egy általánosabb jelölés is vár majd. Folytassuk, helyettesítsünk be:

[1] Ha nem pont félúton lennénk, és az egyik bázisállapottal θ szöget zárna be, akkor persze a másik bázisállapottal ez $(\pi - \theta)$ lenne.

$$|\psi\rangle = \cos\frac{\pi}{4}|0\rangle + \sin\frac{\pi}{4}|1\rangle$$

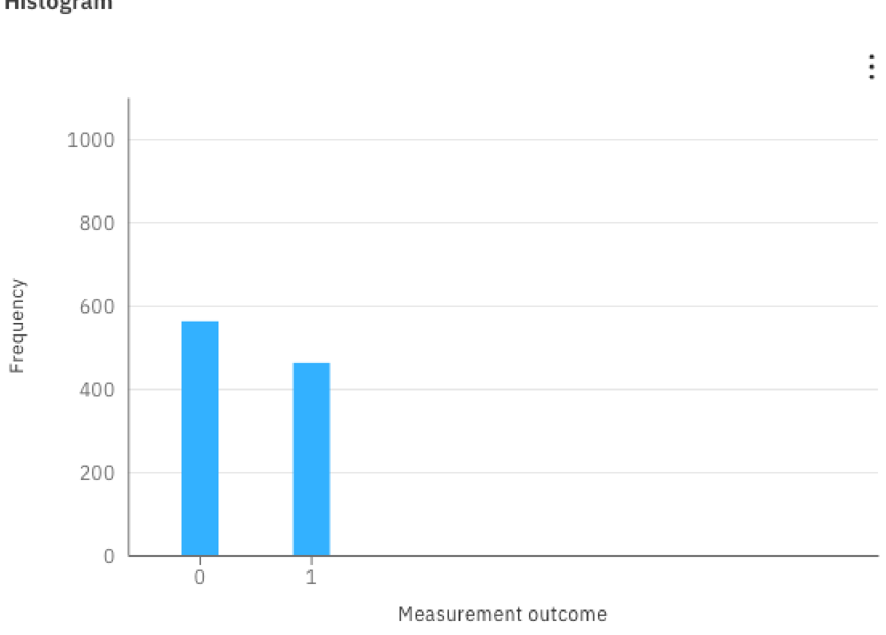
$$|\psi\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$$

Ezzel a jelöléssel már találkoztunk, írhatnánk többféle módon:

$$|\psi\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ 1 \end{bmatrix} = |+\rangle$$

Az amplitudókat, állapotot ismerjük már, a valószínűségek vannak hátra. Gondolkodjunk ismét az egységkörben. A valószínűségek összege 1 kell legyen. Erre is létezik egy trigonometrikus azonosság: $sin^2(\theta) + cos^2(\theta) = 1$. Így tehát a valószínűségeink mindkét bázisállapothoz $(\frac{1}{\sqrt{2}})^2 = \frac{1}{2}$.

Tehát méréskor a számítási bázisban teljesen véletlenszerűen vagy 0-t vagy 1-et fogunk kapni. Ha futtatjuk az 1 qubitből és 1 Hadamard kapuból álló áramkörünket, majd mérünk, ilyen eredményt kapunk nagyjából, 1024 ismétlés után:



A minimális áramkör 1024-szer futott az IBMQ_Athens 5 qubites kvantumszámítógépen. Nem teljesen egyenlő, de ha végtelenszer mérnénk, az lenne. Itt láthatjuk, hogy mérés valóban csak az adott végállapot valószínűségi eloszlásából egy mintavételezésnek felel meg.

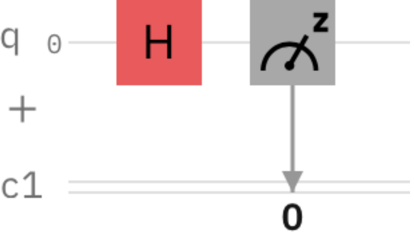
4.2. Érmedobás

Amikor az ember megismerkedik a világgal, minden teljesen kiszámítható. Ha elengedem a föld felett 1,5 méterrel a telefont, akkor le fog esni[2]. Ha írok egy kódot pl. összeadásra, akkor a kód az összeadás eredményét fogja visszaadni. Biztosan.

Sőt, még ha egy randomgenerátort írunk, akkor is teljesen determinisztikusan, kiszámíthatóan fog valamilyen input (idő, clock time, akármi) alapján meghatározott eredményt adni. Nincs valódi véletlen. És most mégis szembesülünk valamivel, ami viszont teljesen véletlenszerű. Kikövetkeztethetetlen.

Részben itt az első teljes algoritmusunk is: egy valódi randomgenerátor. Bármilyen súlyozott véletlent elő tudunk idézni. Legegyszerűbb az egyenletes: ezt láttuk az előző pontban. 1024-szer futtatva az áramkört, kb 50-50% eséllyel kaptunk 0-t és 1-et. Tökéletes, egyenletes véletlen: érmedobás.

Az áramkör hozzá egyszerű, csak egy Hadamard kaput és egy mérést tartalmaz:



[2]... és összetörik, mire a szomszédban felkapcsolódik a villany, és valaki pedig elkezd két literál mentén rezolválni, kiscicák pedig elhullani. Ne dobáljuk a telefont.

Így néz ki Qiskitben implementálva:

```
qr = QuantumRegister(1, 'q')
cr = ClassicalRegister(1, 'c')
circuit = QuantumCircuit(qr, cr)

circuit.h(qr[0])
circuit.measure(qr[0], cr[0])
```

És így néz ki qasm2-ben:

```
1   OPENQASM 2.0;
2   include "qelib1.inc";
3
4   qreg q[1];
5   creg c[1];
6
7   h q[0];
8   measure q[0] -> c[0];
```

4.3. Fázis

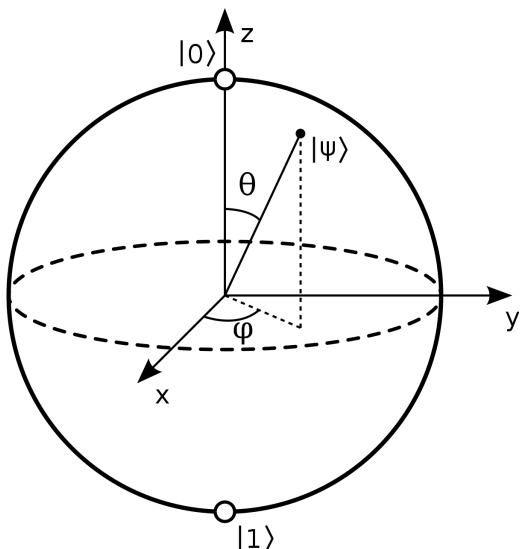
Eddig sokat nem foglalkoztunk az amplitudók előjelével, eltűntek a négyzetre emeléskor. Azonban nagy jelentőségük van. Segítségükkel az amplitudók *kiolthatják* egymást[3] egy szuperpozícióban. Ez egy olyan tulajdonság, amit több a későbbiekben megismert algoritmus felhasznál.

Az első fázis kapunk, tudtunk nélkül a Z kapu volt, amely egy qubiten a $|0\rangle$ fázisát békénhagyva a $|1\rangle$ bázisállapot fázisát negálja. Ez a fázis az a plusz információ, amivel egy qubit több egy klasszikus bitnél. Egy klasszikus bit információtartalma ilyen módon feleannyi, mint egy qubitnek. Kihasználni viszont nem olyan egyszerű, mert az algoritmusunk kezdetekor a kezdőállapot egy klasszikus bitstring, és a mérés eredménye is az, tehát ezt az exponenciális kapacitást kizárólag számítás közben tudjuk kihasználni, vagy szupersűrűen kódolva az információt, bonyolult kezdőállapot létrehozásával.

[3] Azonos amplitudójú, ellentétes fázisok esetén (maximális interferencia).

Az előjel nem elég. Az önmagában nem lenne több információ, mégegy szám kell a fázis leírására, elvégre az sem csak -1 vagy 1 lehet. Nekünk egy számmal, a bázisállapot együtthatójával az amplitudót és fázist egyszerre kell tudnunk jelölni, ráadásul az összefüggéseket megtartva.

Tehát a komplex számok használata elkerülhetetlen. Visszatérve a Bloch gömbhöz:



Thetával már megismerkedtünk, és azzal is, hogy a Bloch gömbön ez a theta valójában kétszerese a szögnek amivel mi számolni szeretnénk. Fí lesz nekünk a másik paraméterünk, és az állapotunk így alakul:

$$|\psi\rangle \;=\; \cos\frac{\theta}{2}\,|0\rangle \;+\; e^{i\varphi}\sin\frac{\theta}{2}\,|1\rangle$$

A fázisból származó együttható φ-től függően lehet valós is - sokszor az. Mégegy fontos ténnyel számolnunk kell: észre lehet venni, hogy $|0\rangle$ nem kapott együtthatót a fázisból. Kaphatna. Viszont csak a bázisállapotok egymáshoz viszonyított fázisa fontos. Csak képzeljünk el egy hullámzó tengert, ahol a hullámok itt-ott összecsapnak, magasabbak, alacsonyabbak. Ezzel a képpel mindenki elképzelhette a vizet, akár a partot, élőlényeket, időt... de a tenger helyét a Földön nem sokan. A hulllámoknak nem az számít, hol a tenger, hanem a saját környezetük, illetve kizárólag a többi hullám, hogyha minden egyéb környezeti hatástól eltekintünk.

Ugyanígy a kvantumállapotoknál. A rendszer globális fázisa nem számít, csak a bázisállapotok egymáshoz viszonyított fázisa. A fenti képletben ha a $|0\rangle$-nak lenne fázisa, azt kivonhatnánk mindkét bázisállapotból anélkül, hogy bármit változtatnánk számítási szempontból.

4.4. Egyéb egybites kapuk

4.4.1. P kapu

Az X, I, H, Z kapuk után kicsit árnyaljuk a szép tiszta képet, de még mindig csak egybites kapukkal. Nem csak egész félgömbnyi forgatásaink (negálásaink) vannak, hanem annak töredéke is.

Az első paraméteres kapunk következik. Ugyanazon a tengelyen forgat, mint a Z kapu. A Bloch gömbön látott φ a paramétere. És ezzel a paraméterrel forgat.

$$\text{A P}(\varphi)\text{ kapu mátrixa:} \quad \begin{bmatrix} 1 & 0 \\ 0 & e^{i\varphi} \end{bmatrix}$$

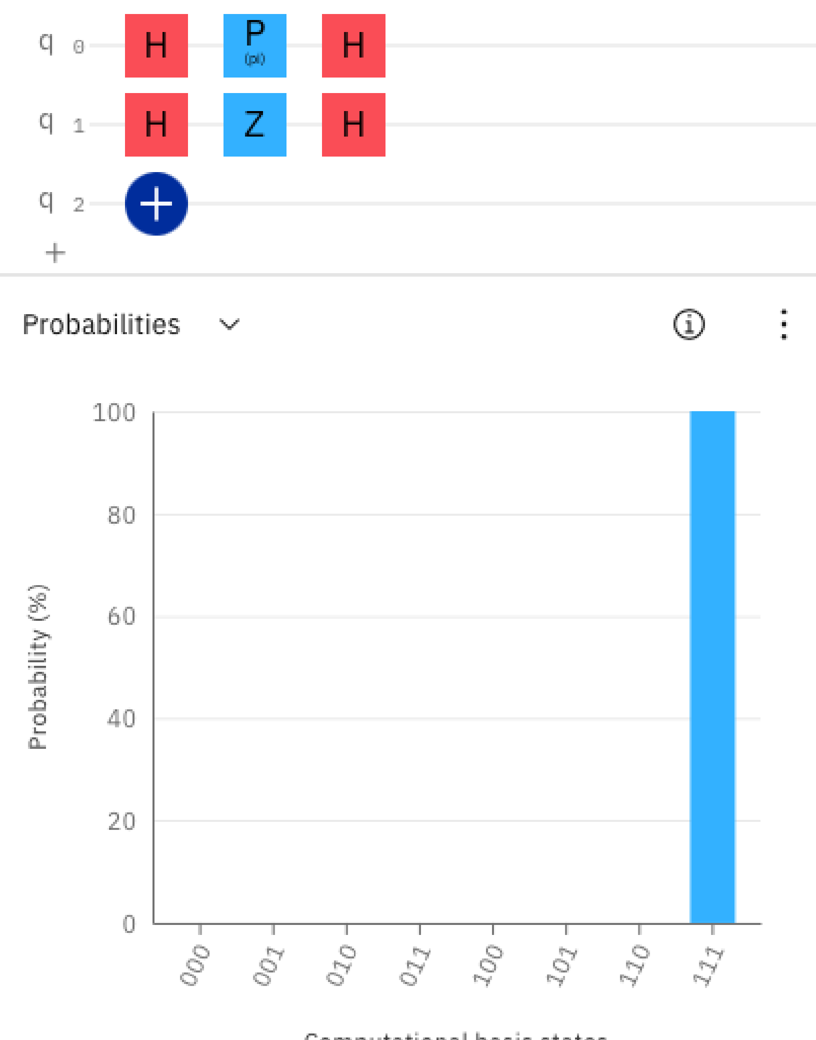
Elsőre kényelmetlennek tűnhet, hogy *paramétere* van egy kapunak, de méginkább az alakja, de ilyen kaput már láttunk: a Z kapu pontosan egy P kapu, $\varphi = \pi$ paraméterrel. Kicsit jártasabbaknak ismerős lehet, hogy Euler azonosságát kapjuk, $e^{i\pi} = -1$.

Kevésbé jártas érdeklődőknek:

$$e^{i\pi} \;=\; cos(\pi) \;+\; i\; sin(\pi) \;=\; -1+0 \;=\; -1.$$

Így elképzelni nem könnyű, nézzük meg a circuit composer segítségével. Mind a P kaput, mind a Z kaput egy-egy Hadamard szendvicsbe téve könnyen vizsgálhatjuk, ugyanis a nemrég látott azonosság segítségével így könnyedén összevethetjük az X kapuval.

$$\text{H(P}(\pi)\text{)H}\,|0\rangle \;=\; \text{HZH}\,|0\rangle \;=\; \text{X}\,|0\rangle \;=\; |1\rangle\,.$$



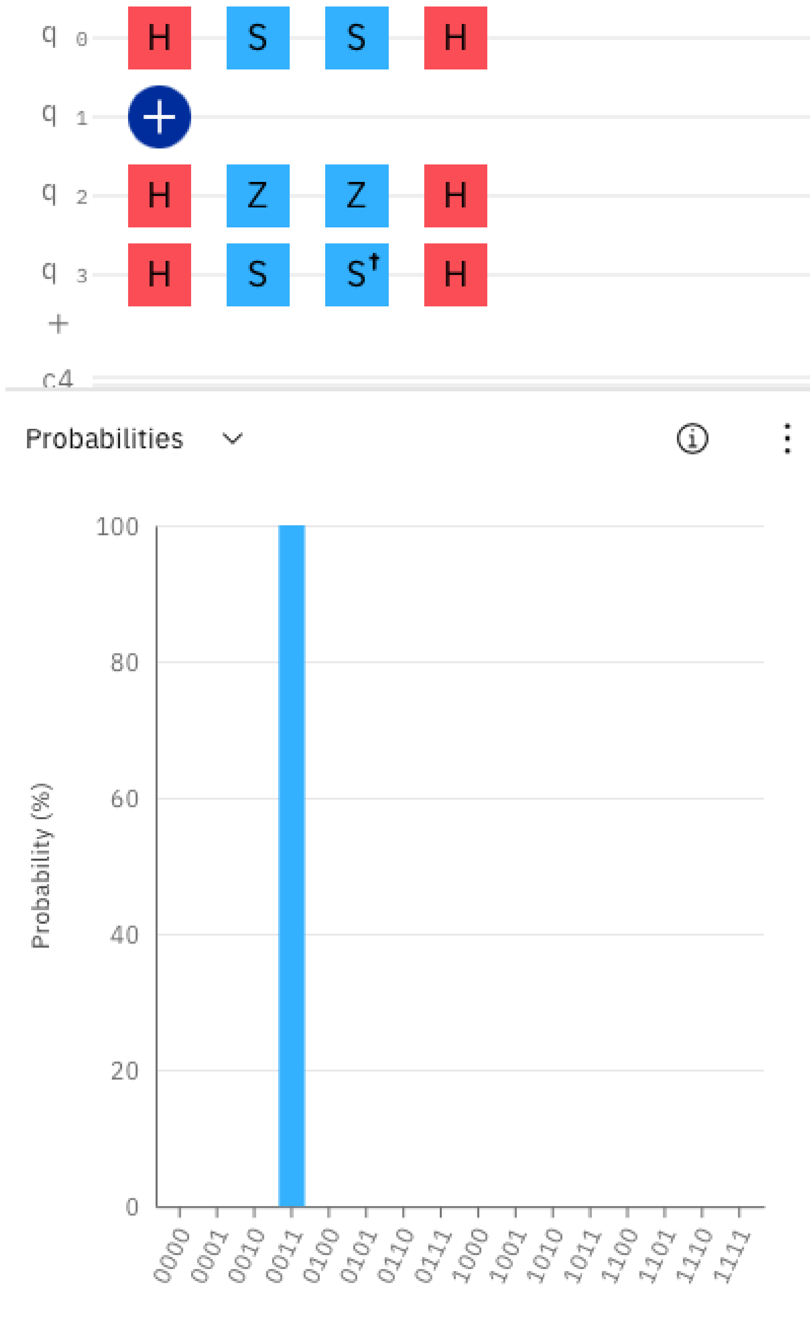
4.4.2. S kapu

Az S kapu nem lesz nagy újdonság. Olyasmi, mintha a Z kaput használnánk, de nem π-vel forgatunk, csak a felével. Ez matematikailag úgy néz ki, hogy $\sqrt{\mathrm{Z}} = \mathrm{S}$, és egy $\frac{\pi}{2}$ szögű forgatás. Ráadásul ismét egy P kapunk van, $\frac{\pi}{2}$ paraméterrel.

Fontos megjegyezni, hogy az eddigi kapuk mind önmaguk inverzei, matematikailag unitér mátrixok. Az S kapu inverze a konjugált transzponáltja, jelölése $\mathrm{S}^{\dagger}$.

$$\mathrm{S} = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{2}} \end{bmatrix} \qquad \mathrm{S}^{\dagger} = \begin{bmatrix} 1 & 0 \\ 0 & e^{-i\frac{\pi}{2}} \end{bmatrix}$$

Az előző példához hasonlóan nézzük most az S kaput: a felső két qubiten láthatjuk hogy SS=Z és HSSH = HZH = X. Alatta pedig hogy a Z önmaga inverze, az S-nek pedig az $S^\dagger$ az inverze. (Note: a kapott bitstring mindig a "0011" lesz, azaz a fenti ábrán "q3q2q1q0").



4.4.3. R_Z kapu

Az R kapuk rotation - forgatás kapuk, alsóindexben jelölve, melyik tengely mentén. Egyparaméteres kapuk, egy szöget várnak paraméterként. Mivel a háromféle forgatással bármilyen lehetséges (tiszta) állapotot elő tudunk állítani 1-1 paraméter segítségével, ezért viszonylag számításilag "olcsó" és pontos kapukról van szó.

A kurzus második felében megismerkedünk majd hibrid klasszikus-kvantum optimalizáló algoritmusokkal. Ezek az optimalizáló algoritmusok jellemzően ilyen egyparaméteres forgatásokból álló paraméteres kvantumáramkörök optimális paramétereit igyekeznek megtalálni. Tehát megértve a különböző forgatások lényegét, és a szuperpozíciót, készen állunk továbblépni, és több qubites rendszerekkel foglalkozni, amelyekből már összetettebb áramköröket, később algoritmusokat tudunk építeni.

Mindhárom tengely felé fel tudunk írni ilyen egyparaméteres forgatási kaput, de most csak egyet fogunk megnézni, a többi triviális[4].

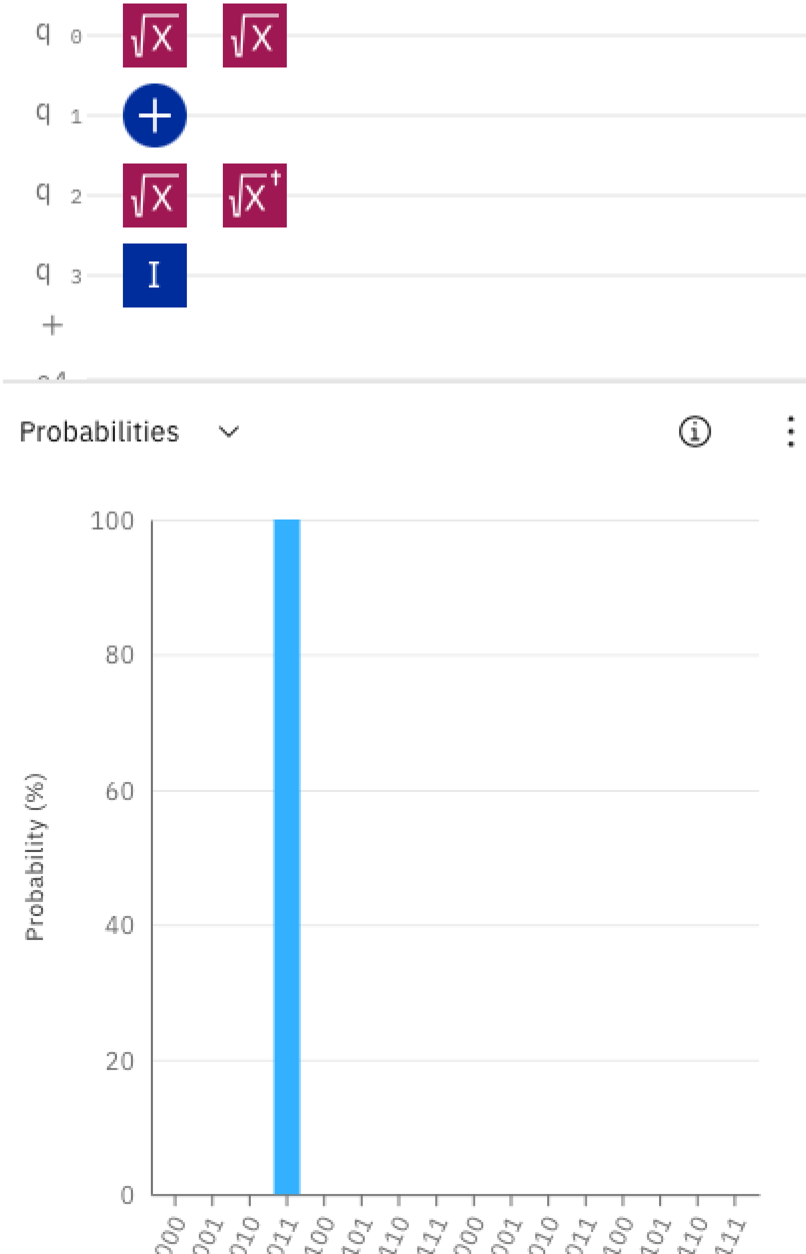
A következő egyparaméteres, egybites kapunk az R_Z kapu. R mint "rotate", Z pedig hogy a Z tengely mentén. A kapu hatása intuitíven nagyon hasonlít a P kapuéra. Viszont itt a paraméter a másik szögünk, theta (θ).

A mátrixa elég bonyolult, majd megismerkedünk vele később. De ismét, van egy paraméteres forgatásunk. Most nem a Z kapu féle irányban, hanem az X kapu félében.

Tehát például definiálhatjuk a $\sqrt{X}$=SX kaput az S kapuhoz hasonlóan. Ez tulajdonképp csak egy $R_Z(\theta) = R_Z(\frac{\pi}{2})$ kapu.

Nézzük tehát a circuit composerben az SX = $R_Z(\frac{\pi}{2})$ kaput. A felső két qubitet negáljuk, két "negyedforgatással" és az X kapuval, azaz "félforgatással". Az alsó két qubiten az SX és az SXdg kapuk vannak szemléltetve, mint egymás inverzei, tehát együtt az I kapu műveletét adják.

[4]egyértelmű, az olvasó magától könnyedén beláthatja, megérteni házi feladat, stb...



q 0
q 1
q 2
q 3
Probabilities
100
80
60
40
20
0
Probability (%)
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
Computational basis states

Több qubites számítások, összefonódás

5.1. Több qubites állapotok

Egész eddig csak 1 qubites rendszerekkel, állapotokkal foglalkoztunk[1]. Az eddigiek alapján már az egész elképesztő, mennyi számítást lehet csak 1 darab qubittel elvégezni, mennyi állapota lehet.

De persze, mint ahogy klasszikusan is, több bittel többre megyünk. Például 3 bittel $2^{\mathbf{3}}$ féle bitstringünk lehet. Ugyanennyi qubittel 8 bázisállapotunk lesz, a rendszerünk pedig ennek valamilyen lineáris kombinációja[2], szuperpozíciója.

5.1.1. Jelölések, bázisállapotok

Tegyük fel, van 3 qubitünk. Kis lineáris algebrát segítségül hívva, miután mindegyik kétdimenziós vektor, ezeket szeretnénk kombinálni úgy, hogy minden dimenziónak "meglegyen a saját helye" a térben. Így hát lesz egy $2^{\mathbf{3}}$ dimenziós vektorunk.

Matematikailag ennek a módja, kétdimenziós terek "kombinálására" a tenzorszorzat. Tegyük fel, hogy van egy $|\mathbf{110}\rangle$ (Note: ez decimálisan 6) állapotunk. Ezt felírhatjuk braket alakban és vektor alakban is. Braket alakban nincs semmi változás: $|\mathbf{110}\rangle$. Minden más bázisállapot amplitudója 0, így azokat nem írjuk ki. Vektor alakban pedig:

$$|110\rangle = \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0\\0\\0\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0\\0\\0\\0\\0\\0\\1\\0\end{bmatrix}$$

[1]Ez nem igaz: 2.2-ben a tenzorszorzatnál volt egy snapshot több qubitről

[2]Ez sem teljesen pontos, de hamarosan kiderül (összefonódás)

Programozó szemmel ránézve rögtön feltűnhet, hogy a vektor utolsó előtti indexe 1, többi 0. Mintha egy állapot[6]=1 lenne, és valóban, decimálisan 6-ot takar a bitstringünk.

5.1.2. Jelölések

Pár jelölést muszáj bevezetnünk. Elsőre ijesztőnek tűnhetnek, de nem bonyolultak. Állapotok jelöléséről van szó - összegképletekkel.

A Qiskit 0-ra inicializálja alapból a qubitjeinket. És a standard számítás alapja is ez, tehát a kiinduló állapotunk a $|\mathbf{000...0}\rangle$. Ha pl. egy 5 qubites rendszerrel dolgozunk, akkor ez $|\mathbf{00000}\rangle$, azaz $|\mathbf{0}\rangle\,|\mathbf{0}\rangle\,|\mathbf{0}\rangle\,|\mathbf{0}\rangle\,|\mathbf{0}\rangle$. És amit ez jelent persze, az az egyes állapotok tenzorszorzata, azaz amivel valójában számolunk, az a $|\mathbf{0}\rangle \otimes |\mathbf{0}\rangle \otimes |\mathbf{0}\rangle \otimes |\mathbf{0}\rangle \otimes |\mathbf{0}\rangle$. Ezt mind kiírni mondjuk 20 qubit esetén nagyon sok lenne. Tehát bevezetjük a $|\mathbf{0}\rangle^{\otimes \mathbf{5}}$ jelölést.

Általánosabban, egy $\boldsymbol{n}$ qubites rendszer esetén a $|\mathbf{0}\rangle^{\otimes \boldsymbol{n}}$-t.

A másik jelölés a braket jelölésre egy összegképlet. Például vegyük először a $|+\rangle$, azaz a $\frac{1}{\sqrt{2}}\,|\mathbf{0}\rangle + \frac{1}{\sqrt{2}}\,|\mathbf{1}\rangle$ állapotot. Ez 1 qubit. Ugyanez - tehát egyenlő szuperpozíció - két qubiten: $\frac{1}{\sqrt{2^2}}\,|\mathbf{00}\rangle + \frac{1}{\sqrt{2^2}}\,|\mathbf{01}\rangle + \frac{1}{\sqrt{2^2}}\,|\mathbf{10}\rangle + \frac{1}{\sqrt{2^2}}\,|\mathbf{11}\rangle$.[3] Illetve egy jelölés, ami formális nyelvekről ismerős kell legyen: $\{0{,}1\}^{\boldsymbol{n}}$ jelölje az $\boldsymbol{n}$ hosszú 0-ból és 1-esekből álló összes bitstringet. Tehát az összegképlet $\boldsymbol{n}$ qubitből álló egyenlő szuperpozícióra:

$$\frac{1}{\sqrt{2^n}} \sum_{i \in \{0,1\}^n} |i\rangle$$

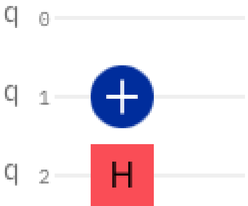
5.1.3. Példák

Persze, felmerülhet, hogy mi szükség egy ekkora vektorra. Az alapvető oka, hogy az elemei nem csak binárisak lehetnek. Mi történik, ha az elöl lévő bit a 0 és 1 egyenlő szuperpozíciójában van? Ekkor már braket alakban is szerepel két bázisállapot:
$\frac{1}{\sqrt{2}}\,|\mathbf{010}\rangle + \frac{1}{\sqrt{2}}\,|\mathbf{110}\rangle$. Ezt bitenként fel lehet írni: $|+\rangle\,|\mathbf{1}\rangle\,|\mathbf{0}\rangle$. Fontos, hogy ilyenkor nem írhatjuk egybe őket, mint a bázisállapotoknál. Számoljuk ki vektor alakban is, majd megnézzük a circuit composerrel is.

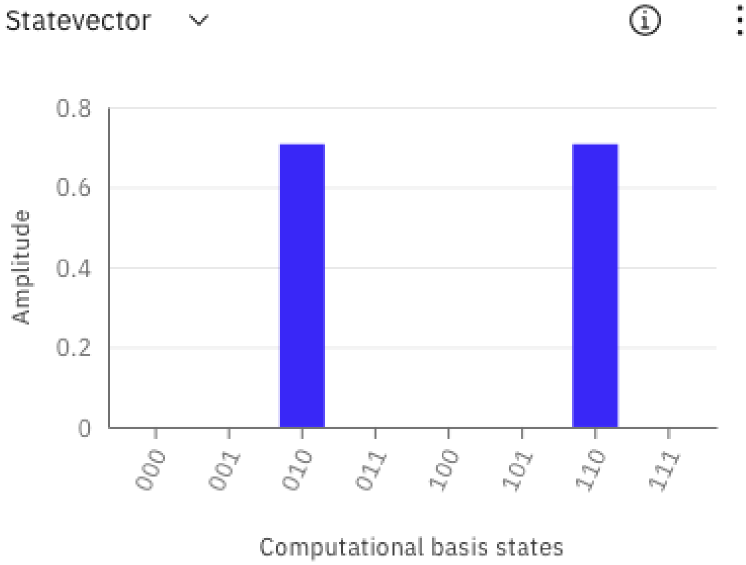
[3] A kis négyzetek szándékosan lettek kiírva, de persze ott $\frac{1}{2}$ szerepel így.

$$\begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \otimes \begin{bmatrix} 0 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ \frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

Decimálisan számolva most megkaptuk a 2 és a 6 egyenlő szuperpozícióját, pontosan ahogy a bitstringekből is gondoltuk volna. No nézzük az implementációt, állapot előállítását. A kiinduló állapot $|\mathbf{00...0}\rangle$. Ahhoz, hogy az első qubit a $|+\rangle$ állapotba kerüljön, egy Hadamard kapuval kell rá hatni. A második qubit $|\mathbf{1}\rangle$ lesz, neki egy negálás kell. A harmadik bitet békén hagyhatjuk. majd a kapuknál megnézzük ezt matematikailag is, egyelőre csak nézzük a példát. Tehát az áramkör (bitsorrend: q2q1q0):



Most már kicsit ismerősebben mozgunk, a mérési esélyek helyett a diagrammon inkább nézzük az állapotvektort:



5.2. Egybites kapuk sorrendje, alkalmazása

Az előző példákon láttuk, hogyan lehet párhuzamosan két qubiten különböző műveletet végrehajtani. Nézzük most meg hogyan tudunk számolni vele. Alapvetően kétféle módon tudjuk a kapukat felpakolgatni: sorban, egymás mögé, azonos qubiten végrehajtva őket, illetve párhuzamosan, egymás alá, egyidőben több qubiten végrehajtva őket. Mindkettőben van egy kis csavar, de nem megy szembe a tanult logikánkkal.

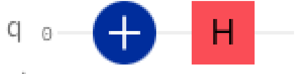
5.2.1. Kapuk sorozata

Kezdjünk egy egyszerűbbel: $\mathrm{HX}|\mathbf{0}\rangle$. Először végigszámoljuk, utána megnézzük composerrel. Két kapu sorozata esetén elég *összeszorozni* őket. A szorzás sorrendje fontos, a legtöbb mátrix esetén a szorzás nem kommutatív - viszont unitér mátrixok szorzata is unitér mátrix lesz, tehát elég balról jobbra összeszorozgatnunk a mátrixokat, majd beszorozni ezzel az állapotvektorunkat.

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

Matematikailag. Bár a kapuk közül az első, amelyik hatni fog a qubitünkre, az a felírt sorozat utolsó kapuja. Tehát először negáljuk az állapotunkat, utána hajtjuk végre

a Hadamard kaput. Tehát ha ebből áramkört szeretnénk építeni, akkor az így fog kinézni composerben:



És így néz ki qiskitben és qasm2-ben:

```
1   from qiskit import QuantumRegister,
    ClassicalRegister, QuantumCircuit
2   from numpy import pi
3
4   qreg_q = QuantumRegister(1, 'q')
5   creg_c = ClassicalRegister(1, 'c')
6   circuit = QuantumCircuit(qreg_q, creg_c)
7
8   circuit.x(qreg_q[0])
9   circuit.h(qreg_q[0])
```

```
1   OPENQASM 2.0;
2   include "qelib1.inc";
3
4   qreg q[1];
5   creg c[1];
6
7   x q[0];
8   h q[0];
```

5.2.2. Párhuzamos végrehajtás

Most lássuk, hogy tudunk számolni akkor, ha van pl. 2 qubitünk valamilyen szuperpozícióban, és hatni szeretnénk az egyik qubitre mondjuk egy Hadamard kapuval, másikra pedig egy X kapuval. Ezt meg tudjuk közelíteni úgy, hogy bitenként végrehajtjuk, azaz: $(\mathrm{H}\,|0\rangle) \otimes (\mathrm{X}\,|0\rangle)$. De egy fokkal elegánsabb az egész bázisállapotra ható operátort kiírni, és ezt meg is tudjuk tenni, ugyanis a tenzorszorzat disztributív a (direkt)szorzás felett, tehát:

$$(\mathrm{H}\,|0\rangle) \otimes (\mathrm{X}\,|0\rangle) \;=\; (\mathrm{H} \otimes \mathrm{X})\,(\;|0\rangle \otimes |0\rangle \;=\; (\mathrm{H} \otimes \mathrm{X})\,|00\rangle$$

$$\text{Kiszámolva pedig:} \qquad \frac{1}{\sqrt{2}}\,|01\rangle + \frac{1}{\sqrt{2}}\,|11\rangle$$

Ez most braket jelöléssel szép, látszik is az eredményen, hogy az elöl lévő qubit vagy 0 vagy 1, a hátul lévő mindig 1. És az amplitudókból az is, hogy az elöl lévő egyenlő szuperpozícióban van.

Nézzük meg vektoros jelöléssel is. Ez kicsit bonyolultabbnak tűnhet, de nem rejt el semmit (mint például a kapuk mátrixait, elemeit a braket). A Hadamard kapuból kiemelem az $\frac{1}{\sqrt{2}}$-t, hogy átláthatóbb legyen:

$$(\mathbf{H} \otimes \mathbf{X})\,|00\rangle = \left(\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} \otimes \begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}\right)\begin{bmatrix}1\\0\\0\\0\end{bmatrix} =$$

$$\frac{1}{\sqrt{2}}\begin{bmatrix}1 \times \begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix} & 1 \times \begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}\\ 1 \times \begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix} & -1 \times \begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}\end{bmatrix}\begin{bmatrix}1\\0\\0\\0\end{bmatrix} =$$

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

$$\frac{1}{\sqrt{2}}\begin{bmatrix}0\\1\\0\\1\end{bmatrix} = \begin{bmatrix}0\\ \frac{1}{\sqrt{2}}\\ 0\\ \frac{1}{\sqrt{2}}\end{bmatrix}$$

Az áramkör pedig:

5.3. CNOT kapu, kontrollált operátorok

És elértünk az első kétbites kapuhoz. De még mielőtt elkezdjük használni, egy pár szó a szerepéről. Egy nagy különbség a klasszikus és a kvantumszámítás között, hogy míg a klasszikus kapuk járhatnak információveszteséggel, a kvantum kapuk nem, ráadásul a kvantum kapuknak reverzibilisnek kell lenniük, azaz az operációt "visszafele" is végre kell tudni hajtani. Ebből az is következik, hogy ha 2 biten végrehajtunk egy kaput, akkor kizárólag az eredményből meg kell tudjuk mondani, melyik bitnek pontosan mi volt az állapota a kapu előtt.

Ha belegondolunk, klasszikusan az AND és az OR kapuk például 2 bitet várnak inputnak, és 1-et adnak vissza. Elveszítünk 1 bitnyi információt. Ráadásul ha egy AND kapu után kapunk egy 0 bitet, akkor van 3 lehetséges input, amelyikről nem tudjuk eldönteni, melyik volt a valódi. Ilyen az összeadás egy biten, a XOR (kizáró vagy) kapu is.

Amivel most megismerkedünk, az lényegében egy XOR műveletet hajt végre, de információveszteség nélkül. A kapu a CNOT (Controlled NOT) vagy CX. A hatása, hogyha a control bit 1, akkor negálja a target bitet (azaz felcseréli a 0 és az 1 amplitudóit). A mátrixa is egészen kikövetkeztethető:

$$\text{CNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

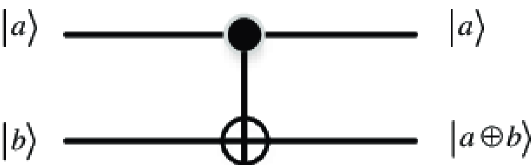
Nézzük meg, hogy egy általános kétbites rendszeren mit is csinál:

$$|\boldsymbol{a}\rangle = \begin{bmatrix} \boldsymbol{a_{00}} \\ \boldsymbol{a_{01}} \\ \boldsymbol{a_{10}} \\ \boldsymbol{a_{11}} \end{bmatrix} \qquad \text{CNOT}\,|\boldsymbol{a}\rangle = \begin{bmatrix} \boldsymbol{a_{00}} \\ \boldsymbol{a_{01}} \\ \boldsymbol{a_{11}} \\ \boldsymbol{a_{10}} \end{bmatrix}$$

Az $|\boldsymbol{a}\rangle$ állapot együtthatóit jelöljük itt $\boldsymbol{a_{xy}}$-nal az xy bázisállapothoz.

5.3.1. Példa

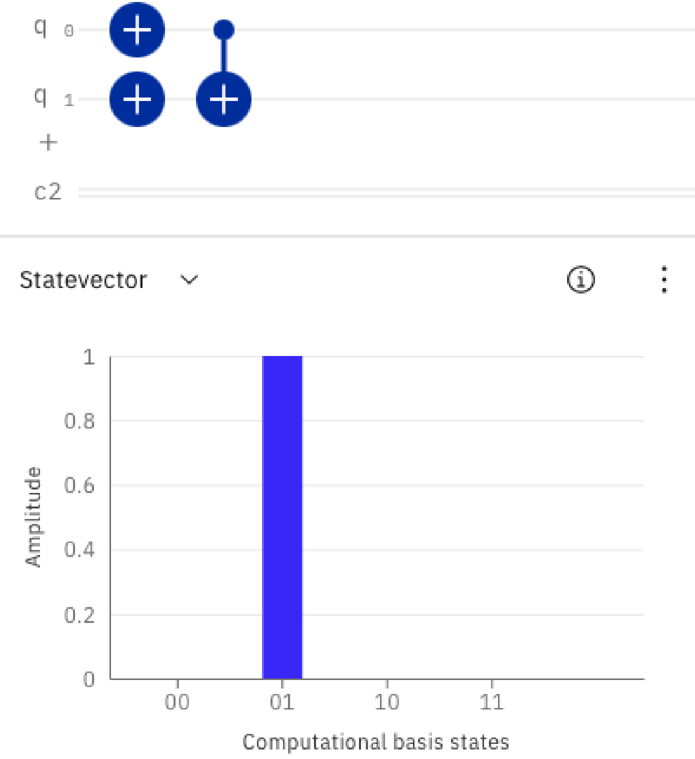
Először is, aritmetikailag nézzük meg, a jelölésével együtt:



Az $|\boldsymbol{a}\rangle$ a control bit, az állapota nem változik. A $|\boldsymbol{b}\rangle$ a target bit, ehhez binárisan hozzá lesz adva a. Azaz egy XOR művelet valósul meg.

Nézzünk egy áramkört, amely előbb előállítja a $|\mathbf{00}\rangle \xrightarrow{X^{\otimes 2}} |\mathbf{11}\rangle$ állapotot. Majd ha ez megvan, akkor végrehajtja rajta a CNOT kaput: $|\mathbf{11}\rangle \xrightarrow{\text{CNOT}} |\mathbf{01}\rangle$ (megj.: a q0 a control bit, ez a hátsó!).

Az áramkör, és az amplitudók pedig itt láthatók:



5.4. Összefonódás

Elérkeztünk a szuperpozíció után a másik, talán még jelentősebb jelenséghez a kvantumszámítás területén. Fizikailag képesek vagyunk összefonni[4] qubiteket. Ekkor azután ők közösen hordoznak valamennyi információt.

Az állapotokból aktuális, kiolvasható információt mérés után tudunk kapni. Persze nem csak az összes qubitet lehet megmérni egy rendszerből, hanem 1-et is magában például. Ugyanakkor ha megmérünk 1 qubitet, az magában nem hordoz információt a többi qubit állapotáról - kivéve ha összefonódott állapotban volt más qubitekkel.

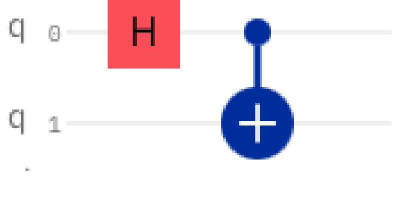
Ha van 2 összefonódott qubitünk, és megmérjük az egyik állapotát, akkor mérés nélkül meg tudjuk mondani a másikét. És ez még fizikai távolságtól sem függ. Ennek fontos szerepe van majdnem az összes hatékony kvantum algoritmusban. Sőt, külön kutatások irányulnak afelé, hogy ez-e *kizárólag* az a jelenség, ami az összes aszimptotikus kvantum gyorsulásért felelős.

Egy példával élve: gondoljuk végig a következő leegyszerűsített képet: van 2 atommagunk, 1-1 elektronnal körülötte. Az elektronoknak saját pályájuk van. Mi történik, ha az atommagok olyan közel kerülnek, hogy a két elektron már nem tudná tartani az önálló pályáját. Melyik magot választaná? A válasz elég intuitív: *mindkettőt*. Egyszerre.

Az összefonódást előidézni algoritmikusan nem nehéz. Az előző pontban foglalkoztunk a CNOT kapuval, amelyik negálja a target bitet, ha a control 1. De mi történik, ha a control bit egyenlő szuperpozícióban van? Akkor a target bit is abban lesz? Lássuk.

Először most példát nézünk: $|00\rangle \xrightarrow{H \otimes I} \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|10\rangle \xrightarrow{CNOT} \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$

Ekkor ha megmérjük csak az első qubitet mondjuk, 50-50% eséllyel lesz 0 vagy 1, de a másik qubit viszont 100% eséllyel ugyanaz lesz. Az áramkör pedig ez:



[4] angolul "entangle"

5.4.1. EPR-pár, tulajdonságai

A példa után lássuk a matekot, absztrakciót. Mitől összefonódott ez az állapot. Az atomos példában hasonló történik. A fejezet eleje táján szerepel, hogy *minden* állapot felírható különálló qubitek tenzorszorzataként. Előtte pedig, hogy a Bloch gömb *felszínén* helyezkednek el az állapotaink. Semelyik dőlt betűs szó nem igaz. És látni is fogjuk, miért - belátjuk, hogy a fent látott állapot, az EPR-pár nem áll elő qubitek tenzorszorzataként, azaz nem szétbontható két külön bitre.

Állítás: Az EPR-pár nem áll elő qubitek tenzorszorzataként.

Bizonyítás: Indirekt módon: tegyük fel, hogy a $\frac{|00\rangle+|11\rangle}{\sqrt{2}}$ állapot előáll qubitek tenzorszorzataként. Elsőként írjuk fel vektor alakban az állapotunkat, az egyes bázisállapotok amplitudójával:

$$|\psi\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \end{bmatrix} = \begin{bmatrix} \alpha_0 \times \beta_0 \\ \alpha_0 \times \beta_1 \\ \alpha_1 \times \beta_0 \\ \alpha_1 \times \beta_1 \end{bmatrix} = |\alpha\rangle \otimes |\beta\rangle$$

Mivel léteznie kell mindkét qubitnek, teljes valószínűséggel, ezért $\alpha_0^2 + \alpha_1^2 = 1$ és $\beta_0^2 + \beta_1^2 = 1$ kell legyen. Ennek a következményei, hogy

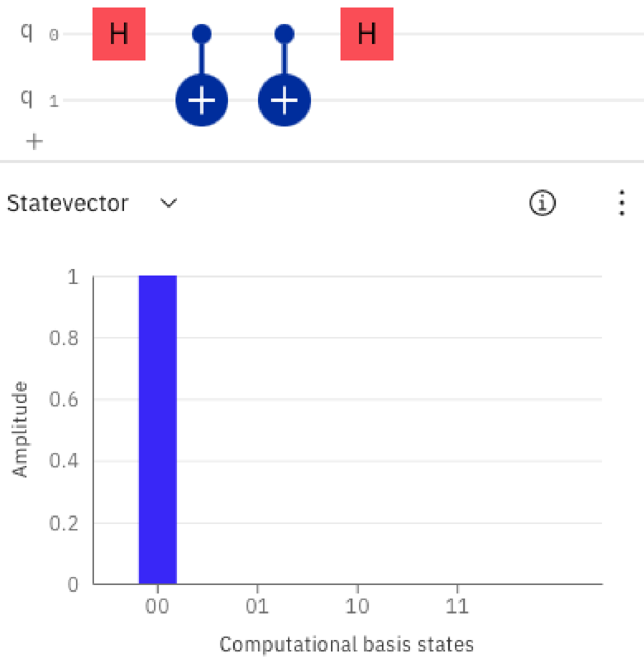
1. $\alpha_0 \neq 0$ és $\beta_0 \neq 0$, különben a szorzatuk 0 lenne.
2. $\alpha_0 = 0$ vagy $\beta_1 = 0$, mert a szorzatuk 0.
3. $\alpha_1 = 0$ vagy $\beta_0 = 0$, mert a szorzatuk 0.
4. $\alpha_1 \neq 0$ és $\beta_1 \neq 0$, különben a szorzatuk 0 lenne.

Az 1. és a 4. állítás kimondja, hogy minden együttható nemnulla. Ez ellentmond a 2. és 3. állításnak is, tehát az állapot valóban nem bontható szét két különálló qubitre.

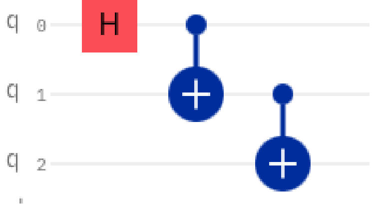
□

5.4.2. További összefont állapotok, szétválasztás

Nem csak ez az egy összefonódott állapot létezik. Létezhet többféle, alapvetően ha bármilyen szuperpozícióban lévő qubitet használunk a CNOT kapu control bitjének, akkor összefonjuk a két bitet. Emellett - ahogy minden kvantum áramkör - ez az áramkör is visszaállítható, az összefonódás pedig megszüntethető ugyanezen kapusorozat fordított alkalmazásával. Ilyenkor persze csak az összefonódott állapot szűnik meg, a két qubit "kiszabadul", önálló állapota lesz, de az összefonódás alatt szerzett számítás nem vész el.



És - ha valaki ilyesmire gondolt - több qubitet is össze tudunk fonni, tulajdonképp akármennyit. De minden ilyen gyengíti a rendszert, növeli a hibák lehetőségét. A következő állapot például a GHZ (Greenberger–Horne–Zeilinger) állapot, 3 qubit teljesen összefonódott rendszere: $\frac{|000\rangle+|111\rangle}{\sqrt{2}}$ állapot:



Univerzális kapu, áramkörök építése

6.1. U kapu

Miután láttuk, hogy az R kapukkal a Bloch gömb bármelyik tengelye körül tudunk forgatni, jogos lenne a kérdés, hogy miért kell 3 kapu rá? A válasz: nem kell.

Létezik egy univerzális kapu, az U kapu, amelynek 3 paramétere van, és képes kifejezni bármely egybites kaput. Minden kvantumszámítógépnek van egy saját, teljes kapukészlete. Ez nem az összes kapu, hanem csak egy olyan részhalmazuk, amellyel kifejezhető az összes. Tehát amikor futtatjuk a kódunkat, bár a python interpretált nyelv, és az utasításokat klasszikusan az interpreter értelmezi, a kvantum áramkörünk *fordul*. A fordítás egyik lépése egy szintaktikai művelet az IBM rendszerében, amely az áramkörünket, kapusorozatainkat azonosságok segítségével a bázis kapukra fordítja.

A bázis kapuk általában az IBM eszközein: CNOT, I, R_Z, SX, X. Természetesen létezik egy optimális, minimális áramkör az adott feladat elvégzésére, minden báziskapuhalmazra. Ez az 5 kapu egy nagyon szűk halmaz[1], így ez a művelet[2] általában bőven növeli az áramkörünk hosszát[3]. Erről a folyamatról bővebben majd a tervezés pontban tanulunk.

Minden áramkör lefordítható a {CNOT, U} kapuhalmazra, és fordítás során először U kapukra bomlik az áramkörünk, majd a báziskapukra az U kapukról.

Az U kapu mátrixa egy általános forgatási mátrix, a fázis együtthatóival:

$$\begin{bmatrix} cos\left(\frac{\theta}{2}\right) & -e^{i\lambda} sin\left(\frac{\theta}{2}\right) \\ e^{i\phi} sin\left(\frac{\theta}{2}\right) & e^{i(\phi+\lambda)} cos\left(\frac{\theta}{2}\right) \end{bmatrix}$$

Elsőre nagyon bonyolultnak tűnhet, de nem lesz ez olyan csúnya, mindjárt lebontjuk, hogy mi mit jelent, és nézünk példákat is.

[1]tekinthetjük utasításkészletnek

[2]transpilation

[3]és ezzel a klasszikus időbonyolultságát is a programunknak, de ez a növekedés nem lehet aszimptotikusan nagyobb, mint polinomiális

6.1.1. Paraméterezés

A fenti mátrixban 3 paraméter szerepel: θ-t és ϕ-t már láttuk. A harmadik, λ pedig a fáziseltolás paramétere.

Nézzünk pár példát, amit már láttunk, csak nem ilyen alakban:

$$\mathrm{U}(\theta, \phi, \lambda) = \mathrm{U}(0, 0, \pi) = \begin{bmatrix} cos(0) & 1 \times sin(0) \\ 1 \times sin(0) & e^{i\pi} cos(0) \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = \mathrm{Z}$$

$$\mathrm{U}(\theta, \phi, \lambda) = \mathrm{U}(0, 0, \lambda) = \begin{bmatrix} cos(0) & 1 \times sin(0) \\ 1 \times sin(0) & e^{i\lambda} cos(0) \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{bmatrix} = \mathrm{P}(\lambda)$$

$$\mathrm{U}(\theta, \phi, \lambda) = \mathrm{U}\left(\frac{\pi}{2}, 0, \pi\right) = \begin{bmatrix} cos\left(\frac{\pi}{4}\right) & 1 \times sin\left(\frac{\pi}{4}\right) \\ 1 \times sin\left(\frac{\pi}{4}\right) & -1 \times cos\left(\frac{\pi}{4}\right) \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} = \mathrm{H}$$

$$\mathrm{U}(\theta, \phi, \lambda) = \mathrm{U}(\pi, 0, \pi) = \begin{bmatrix} cos\left(\frac{\pi}{2}\right) & 1 \times sin\left(\frac{\pi}{2}\right) \\ 1 \times sin\left(\frac{\pi}{2}\right) & -1 \times cos\left(\frac{\pi}{2}\right) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \mathrm{X}$$

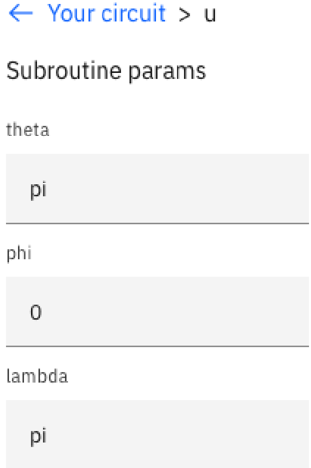
Ismerni kell hozzá az alapvető trigonometrikus azonosságokat, és a komplex számok exponenciális alakját, ha ez nem ismerős, a 2. fejezet segíthet.

6.1.2. Példák

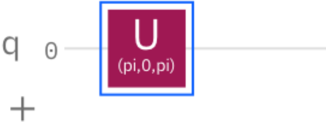
Nézzük meg ezeket a kapukat pár példán a gyakorlatban is:

1. példa: $\mathrm{X}|0\rangle$ kapu, kizárólag U kapuk felhasználásával.

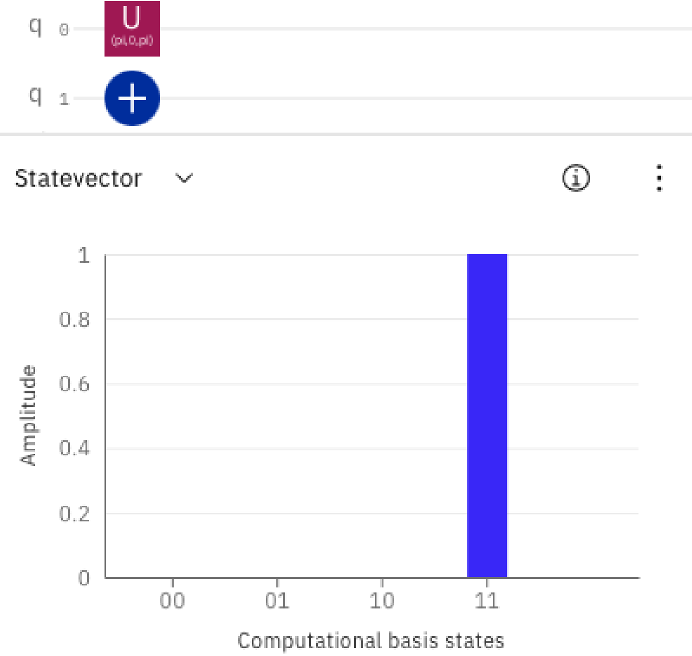
Az X kapuról láttuk, hogy ugyanaz, mint az $\mathrm{U}(\pi, 0, \pi)$ kapu, tehát ezutóbbit kell használnunk. Ezt a példát a composer segítségével fogjuk megnézni. A kapu áramkörre helyezése után rákattintva be tudjuk állítani a paramétereit. Ahogy az összes kvantumos fejlesztői környezetünkben, a "pi" string itt is a π-t takarja.



Az áramkörön pedig így néz ki ezután:



És hogy verifikáljuk, valóban csak negálást végez, hasonlítsuk össze - most utoljára ilyen módon - az X kapuval.



2. példa: HZH$|0\rangle$ kizárólag U kapuk felhasználásával.

Ezt is ismerjük már, de nem U kapukkal. Tehát az U kapuink:

$$\mathrm{H} = \mathrm{U}\left(\frac{\pi}{2}, 0, \pi\right)$$

$$\mathrm{Z} = \mathrm{U}\left(0, 0, \pi\right)$$

Composerben így néz ki:

A composer jó kisebb áramkörök kipróbálására, de ideje megismerkednünk közelebbről a **Quantum Lab**bel, Qiskittel, és megírni ugyanezt programkóddal is. Egyelőre futtatás nélkül, csak az áramkört megépítve. Kell nekünk egy QuantumCircuit object, benne egy egybites kvantum regiszterrel, és egy egybites klasszikus regiszterrel. Utóbbit nem fogjuk még használni, de szükséges lépés az inicializáláshoz, és majd ide kerül mérés után a kvantumregiszter bitjeinek értéke.

Lássuk tehát a kódot:

```
from qiskit import QuantumCircuit
from numpy import pi
from qiskit.quantum_info import Operator

circuit = QuantumCircuit(1,1) # 1 qubit, 1 bit

circuit.u(pi/2, 0, pi, 0) # theta, phi, lambda, qubit_index
circuit.u(0, 0, pi, 0)    # theta, phi, lambda, qubit_index
circuit.u(pi/2, 0, pi, 0) # theta, phi, lambda, qubit_index
```

Az U kapu paraméterezése kommentben olvasható. A képen látható, végén importált Operator pedig hasznunkra lesz a továbbiakban, ugyanis ez az áramkör ekvivalens (globális fázistól eltekintve) az X kapuval. Hogy ezt be is bizonyítsuk, példányosítanunk kell egy

operátort az áramkörünkből, illetve meg kell építsük a másik áramkört, egy darab X kapuval. Ezután már, mintha csak egy szokásos objektumorientált nyelv összehasonlításánál hívnánk az `equals` metódust, hívjuk az `equiv`-et.

A teljes kód tehát így néz ki:

```
from qiskit import QuantumCircuit
from numpy import pi
from qiskit.quantum_info import Operator

circuit = QuantumCircuit(1,1)
circuit.u(pi/2, 0, pi, 0)
circuit.u(0, 0, pi, 0)
circuit.u(pi/2, 0, pi, 0)
op_circ = Operator(circuit)

c_cmp = QuantumCircuit(1,1)
c_cmp.x(0)
op_cmp = Operator(c_cmp)

print(op_circ.equiv(op_cmp))
```

6.2. Kontrollált forgatások

A CNOT kapu után felmerülhet, hogy ha egy X kaput lehet kontrollálni, akkor lehet-e mást is. És lehet, lényegében bármilyen kaput. A mátrixuk pedig általánosan:

$$\mathrm{CU} = \begin{bmatrix} \mathrm{I} & 0 \\ 0 & \mathrm{U} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \mathrm{U_1} & \mathrm{U_2} \\ 0 & 0 & \mathrm{U_3} & \mathrm{U_4} \end{bmatrix}$$

Ilyen kapuk alkalmazásánál észnél kell lenni, mikor mennyire fonja össze a qubiteket. Ha akaratlanul összefonunk qubiteket, annak sok kéretlen következménye lesz. Másrészt

ezeket a kapukat fizikailag megvalósítani "nehezebb" - pontatlanabbak lesznek a számítások, és jelentősen lassabb is a futás.

Kvantum algoritmusok tervezése és írása

7.1. IBM Quantum Experience használata

IBM account kezelése

Elsőként szükségünk lesz az első órákon regisztrált IBM accountra. Ha saját gépen, környezetben szeretnénk programozni, szükség lesz a `qiskit.IBMQ` libre. A hozzáféréshez szükséges egy API token, ami egyedi mindenkinek. Ezt első alkalommal el kell mentsük az IBMQ.save_account(API_KEY) paranccsal.

Aki inkább maradna a Quantum Labnél, amit a kurzus folyamán végig használunk, annak nem szükséges külön mentenie. Innentől viszont közös a folytatás: a betöltéshez, ha valamelyik IBMQ eszközt szeretnénk használni, szükséges az IBMQ.load_account() parancs. Ennek, ha előtte mentettünk vagy az online felületet használjuk, nem kell paraméter.

Providerek

Ha ezzel készen vagyunk, következő lépés egy provider szerzése. Az IBM a felhasználók csoportjait ennek a segítségével különbözteti meg, ez határozza meg, hogy utána milyen eszközöket tudunk használni, milyen paraméterekkel és milyen mennyiségben.

A provider kiválasztásához 3 dolgot kell megadnunk. Nagyjából úgy érdemes elképzelni, mintha egy könyvtárrendszer lenne a számítógépek felett, amin végig kell menni, hogy a fájlokhoz (itt backendekhez) férjünk. A három "mappája" pedig:

1. hub: lehet "ibm-q’ a mindenki számára elérhető eszközökhöz, illetve a kurzus során mi az education program segítségével hozzáférünk más eszközökhöz is, ehhez az "ibm-q-education" a hub neve.

2. group: a két csoport szerint lehet "open", vagy az egyetemihez "uni-szeged-1".

3. project: a fenti tagolásban az általános hozzáféréshez a "main" használatos, a kurzus pedig a "quantum-program" nevet használja.

Qiskitben így néz ki:

```
provider = IBMQ.get_provider(
    hub='ibm-q',
    group='open',
    project='main'
)
```

```
provider = IBMQ.get_provider(
    hub='ibm-q-education',
    group='uni-szeged-1',
    project='quantum-program'
)
```

Az IBMQ.load_account() parancs az előbbi providerrel tér vissza. A hozzáférhető eszközök (backendek) listáját a provider.backends() segítségével ki tudjuk listázni.

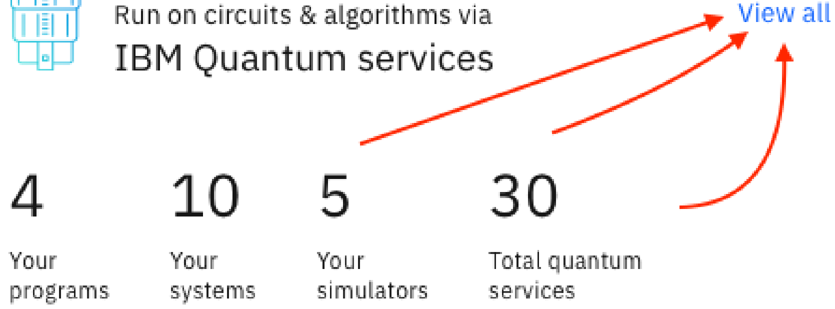
Backendek

A provideren keresztül elérhetünk bizonyos backendeket, eszközöket, amelyeken futtathatjuk a kvantum programjainkat. Többféle backend van:

vannak különböző **szimulátorok**, amelyek igyekeznek realisztikusan szimulálni a valódi QPU-t. A szimulátorok jellemzően robosztusabbak, és gyorsabbak, pontosabbak egy valódinál. Többféle szimulátor van: ibmq_qasm_simulator 32 qubiten szimulálni realisztikus precizítással a programunkat, simulator_statevector szintén több qubiten zaj, és pontatlanság nélkül tesztelni a megírt kódunk helyességét. Emellett a hibajavító kódolásra, és egyéb dolgokra is van külön szimulátor.

és vannak a **valódi kvantumszámítógépek**. Mind szupravezető qubitekkel, de különböző processzorokkal, különböző kapu-megvalósításokkal, és különböző méretben.

Ezeket tudjuk listázni a dashboardról



A lista táblázatos nézetben, saját eszközökre szűrve, a kurzushoz való hozzárendelés után valahogy így fog kinézni:

Name	Qubits ↓	QV	Status	Total pending jobs	Processor type	Features
ibmq_16_melbourne	15	8	Online	15	Canary r1.1	-
ibmq_casablanca	7	32	Online	1	Falcon r4H	
ibmq_santiago	5	32	Online	371	Falcon r4L	-
ibmq_manila	5	32	Online	123	Falcon r5.11L	-
ibmq_bogota	5	32	Online	27	Falcon r4L	
ibmq_quito	5	16	Online	11	Falcon r4T	-
ibmq_belem	5	16	Online	0	Falcon r4T	-
ibmqx2	5	8	Online	22	Canary r1	-
ibmq_lima	5	8	Online	34	Falcon r4T	-
ibmq_armonk	1	1	Online	3	Canary r1.2	

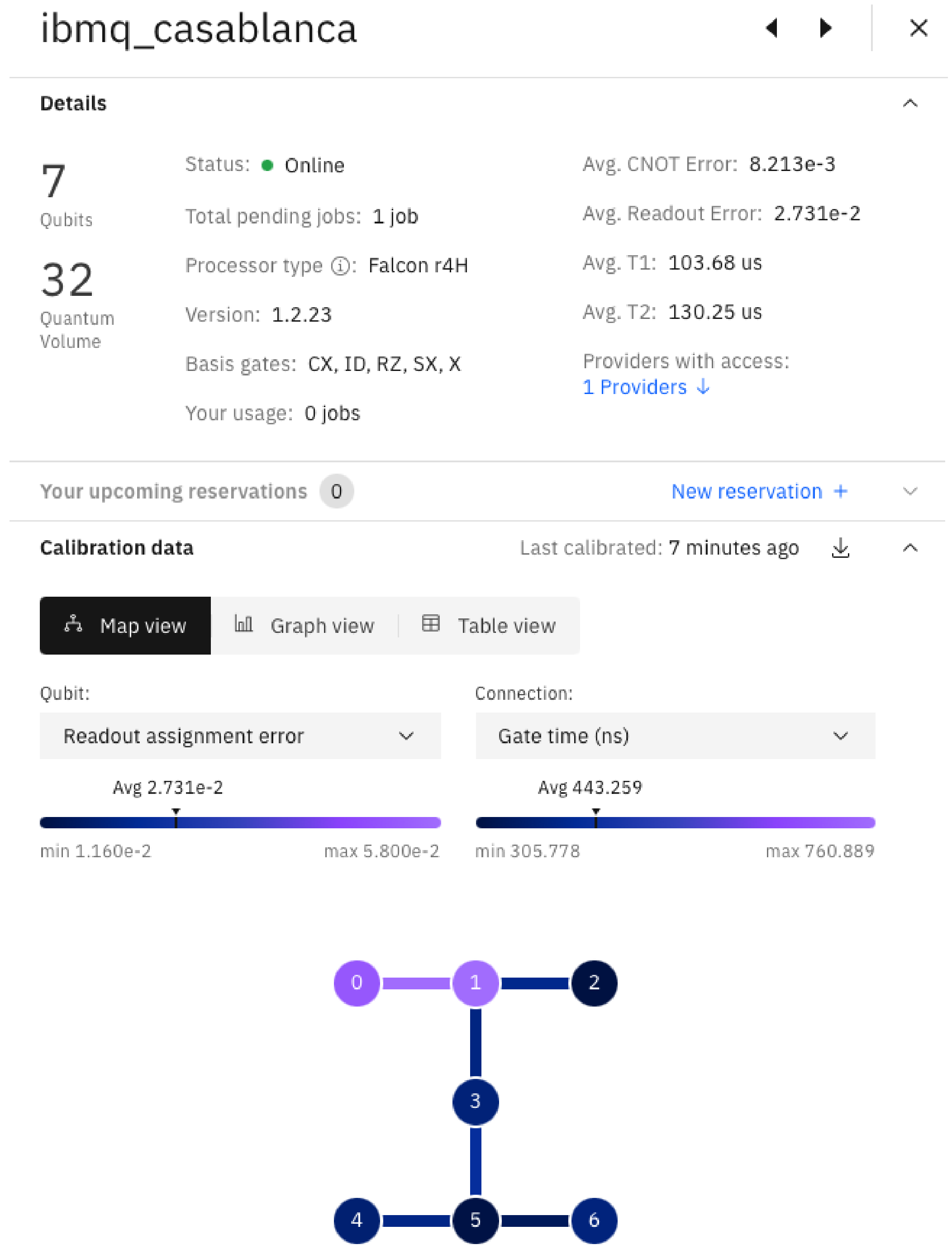
A lista idővel változhat, ahogy az IBM fejleszteni a számítógépparkját, nyugalmaz bizonyos kvantumszámítógépeket, és épít újakat. A kurzus kezdetére például ibmq_16_melbourne and ibmqx2 már nem lesz nyilvános üzemben.

A listán szereplő eszközök közül ibmq_rome és ibmq_casablanca eszközöket fogjuk órán többet használni, ezek ugyanis nyilvánosan nem elérhetők, a kurzus idejére nekünk lesz kizárólag elérhető.

Ez fontos információ, ugyanis úgy működnek ezek az eszközök, mint régen a lyukkártyával programozós kurzusok: az ember megírja a programot, beáll a sorba a géphez, majd amikor sorra kerül, odaadja a kártyát (programját) egy adminisztrátornak, aki futtatja neki, majd két emelettel arrébb egy másik embertől átveheti az ember az eredményt.

A futkosáson, fizikai programon, lyukkártyán, és emberi adminisztrátorokon kívül pontosan ugyanígy működik az IBM Quantum Cloud. A programunkat beküldjük, kivárja a sorát, majd lefut, és visszakapjuk az eredményt. Persze ez sok várakozással járhat, főleg nyilvános eszközökön.

Az eszközök egyes adatai, például a qubitek elhelyezkedése, és kapuk hibái is sokat számítanak, ezeket is meg tudjuk nézni:



Kiolvashatjuk például, hogy egy kapu átlagos végrehajtási ideje 443 nanoszekundum, ami 0.000443 másodperc. Kiolvashatjuk azt is (a qubitek elhelyezkedéséből), hogyha például egy hatbites kvantumregiszterünk van a 0,1,2,3,5,6 indexű qubitekkel[1], és mi a 2-es és 4-es qubitre ráteszünk egy CNOT kaput, akkor az fizikailag nem megoldható. Kicsit dolgozni kell még vele, felcserélni megfelelő qubitek értékeit. A transpiler ezt is végrehajtja helyettünk, de nem feltétlenül optimálisan.

Ha nem akarunk olyan sokat várakozni, akkor az eszközök tulajdonságait le tudjuk kérni, és az alapján keresni egy megfelelő számítógépet:

```
from qiskit.providers.ibmq import least_busy
backend = least_busy(
    provider.backends(
        filters=lambda x:
        x.configuration().n_qubits >= 5
        and not  x.configuration().simulator
        and x.status().operational==True
    )
)
```

Itt például visszakapjuk azt a backendet, amelyiken a legkisebb a sor, feltéve, hogy legalább 5 qubites, valódi kvantumszámítógép, és üzemben van, működik is.

Áramkörök importálása, vezérlése

Láttunk kódot qasm-ben és Qiskitben is, de futtatni csak utóbbi segítségével tudunk. Áramköröket írni viszont - főleg majd qasm3-tól - sokkal kényelmesebb lesz kvantum assemblyben, és sokkal elegánsabb is az egyes áramköröket külön qasm állományokba szervezni.

Kvantum assembly (*.qasm) fájlt nagyon egyszerű Qiskit QuantumCircuit objectté konvertálni, ugyanis az osztálynak van egy from_qasm_file("path") metódusa, amit elég meghívni.

[1] Meg lehet adni ilyeneket is a programnak

Jelenleg Qiskitben, pythonnal könnyebben lehet nagyobb áramköröket kezelni, erre egy példa, hogyha 5 qubiten szeretnénk Hadamard-transzformációt végrehajtani[2], akkor egy for ciklussal lényegesen könnyebb és elegánsabb is odapakolni az 5 Hadamard kaput.

És persze pythonban tudunk elágazni, ha megmérünk 1 qubitet, annak függvényében el tudjuk dönteni, hogy a többivel mi legyen. Tudunk függvényeket definiálni - ez majd főleg a hibrid optimalizálásnál lesz fontos.

Futtatás

Ha megvan az áramkörünk, állhatunk is be a sorba a fent látott, legkevésbé lefoglalt géphez. Futtatáshoz összesen annyi kell, hogy a backend-re ráhívunk egy "run"-t, paraméterként az áramkörrel. job = backend.run(circuit).

Az áramkör, a futtatás és a plot kommentezve van, de pár megjegyzés mielőtt megnézzük: az áramkör egy EPR-párt készít, annak egy párját, amit már néztünk. A két EPR semmilyen módon nem különböztethető meg egymástól, még az egyik qubit mérésével sem, ha nem tudjuk az összefonó kapusorozatot. Ez is egy érdekes tulajdonsága a teljesen összefonódott állapotoknak.

Gyorsan nézzük a matekját, aztán a kódot, és az eredményeket is:

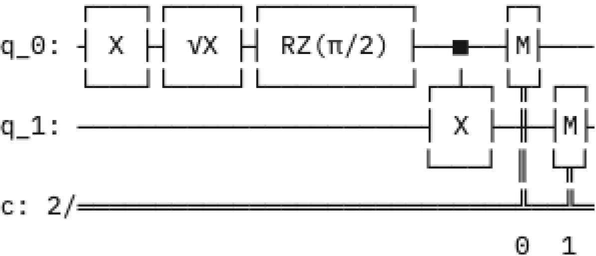
$$|00\rangle \xrightarrow{\text{X}\otimes\text{I}} |10\rangle \xrightarrow{\text{H}\otimes\text{I}} \frac{1}{\sqrt{2}}|00\rangle - \frac{1}{\sqrt{2}}|10\rangle \xrightarrow{\text{CNOT}} \frac{1}{\sqrt{2}}|00\rangle - \frac{1}{\sqrt{2}}|11\rangle$$

A báziskapukra való fordítást némely backenden nekünk kell megtennünk, ezért a Hadamard kaput helyettesítenünk kell az $\text{SX} \times \text{R}_\text{Z}\left(\frac{\pi}{2}\right)$ kapusorozattal. Aki szeretné kiszámolhatja, hogy egy globális fázistól eltekintve ez ekvivalens a H kapuval.

Eredmények

Futtatás után az eredményeket is szeretnénk látni, ezt vissza is adja az eszköz. Illetve mielőtt beküldjük a kódot, érdemes vetni egy pillantást az áramkörre is. Az online notebookban az áramkört egyszerűen printelni tudjuk. Offline kell egy circuit.draw('mpl') és egy plt.show(), hogy a matplotlib kirajzolja. Tehát az áramkör jelenleg így néz ki:

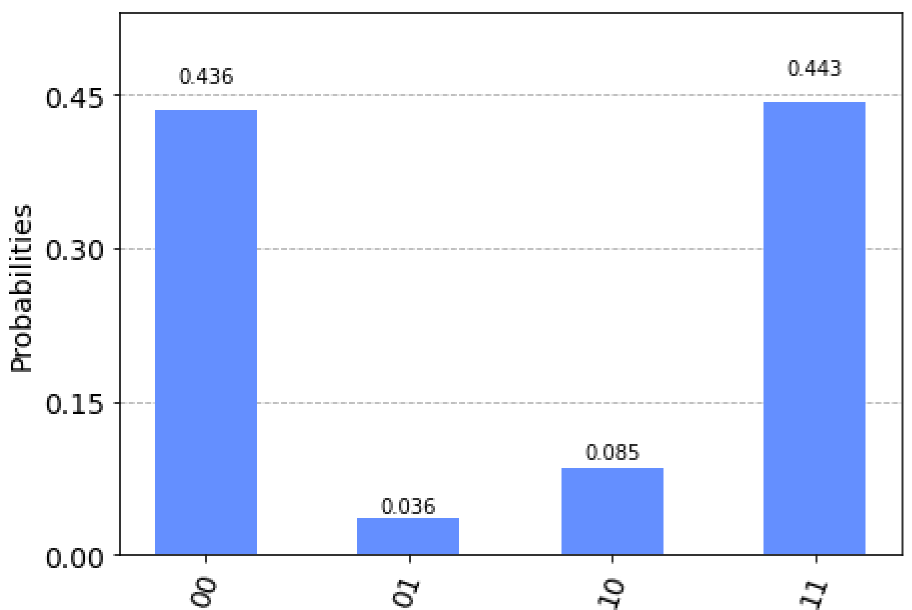
[2]azaz mindegyiken egyszerre végrehajtani a H kaput



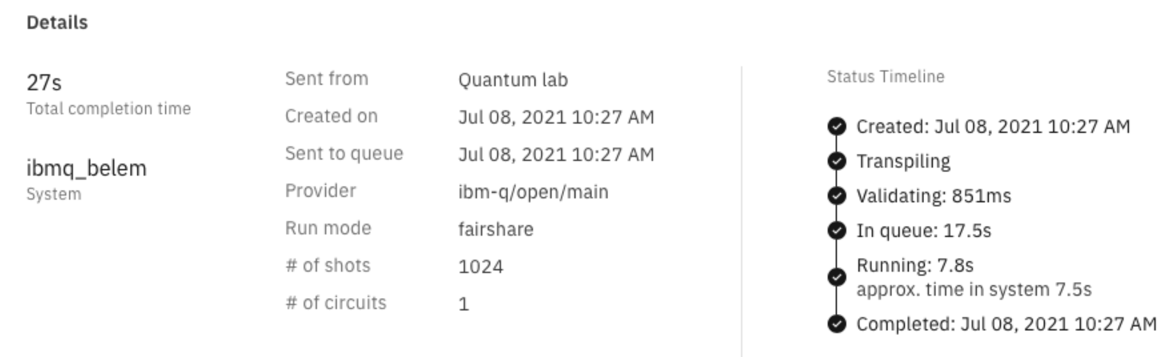
A kódunkhoz már (kicsit fentebb) kerestünk egy backendet, az azutáni rész valahogy így fest:

```
from qiskit import QuantumCircuit
from numpy import pi

circuit = QuantumCircuit(2,2) # 2 qubites regiszter
circuit.x(0)
circuit.sx(0)        # bázis kapuk: ['id', 'rz', 'sx', 'x', 'cx', 'reset']
circuit.rz(pi/2, 0)  # sx * rz(pi/2) = h  (mert a h nem báziskapu!)
circuit.cx(0,1)      # control, target
circuit.measure(0,0)
circuit.measure(1,1)

print(circuit) # nézzük meg az áramkört

job = backend.run(circuit) # beküldjük
result = job.result()      # sync hívás, erre várni fogunk
plot_histogram(result.get_counts()) # nézzük az eredményt!
```

Mivel valódi kvantumszámítógépen futtattuk, ebben valamennyi futási és mérési hiba is lesz, de alapvetően arra számíthatunk, hogy 50-50%-ban nullát (00) és hármat (11) fogunk kapni.



A "Jobs" menüben egyéb adatokat is láthatunk róla:



Tehát 27 másodperc alatt végigment a soron, az ibmq_belem rendszeren, általános licensz alatt futott, 7.8 másodperc alatt 1024-szer[3]. És az is, hogy 1 áramkört küldtünk be, mert lehet többet is, illetve többször futtatni szükség esetén.

[3]Órán megnézegetjük ezeket, a pici kép gyenge felbontással ne zavarjon senkit

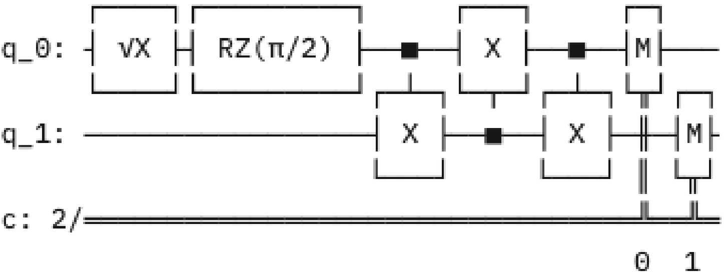
7.1.1. Példa: SWAP

Írjunk áramkört, amely felcseréli két qubit állapotát. Qubit állapotok felcserélésére nagyon sok szükség van, ugyanis két-három qubites kapuk alkalmazásához teljes fizikai összeköttetés, csatorna szükséges, viszont a szupravezető chipek esetén ez az összeköttetés nem túl sűrű. A 67. oldal alján lévő ábra mutatja, melyik qubit melyikkel szomszédos, köztük lehetséges kontrollált kaput végrehajtani. De mi történik ha nem ilyen bitek közt szeretnénk? Másolni nem lehet qubiteket!

Egészen addig kell cserélgetni az állapotokat, amíg egymás mellé nem kerülnek. És ez rendkívül költséges művelet. Gondoljunk bele, nagyjából tízszer annyi idő szükséges egy jóval pontatlanabb CNOT kapu végrehajtásához, mint egy egybites kapuhoz.

A szemléltető áramkör először egyenlő szuperpozícióba teszi az egyik bitet, utána megcseréli a másikkal, majd mér.

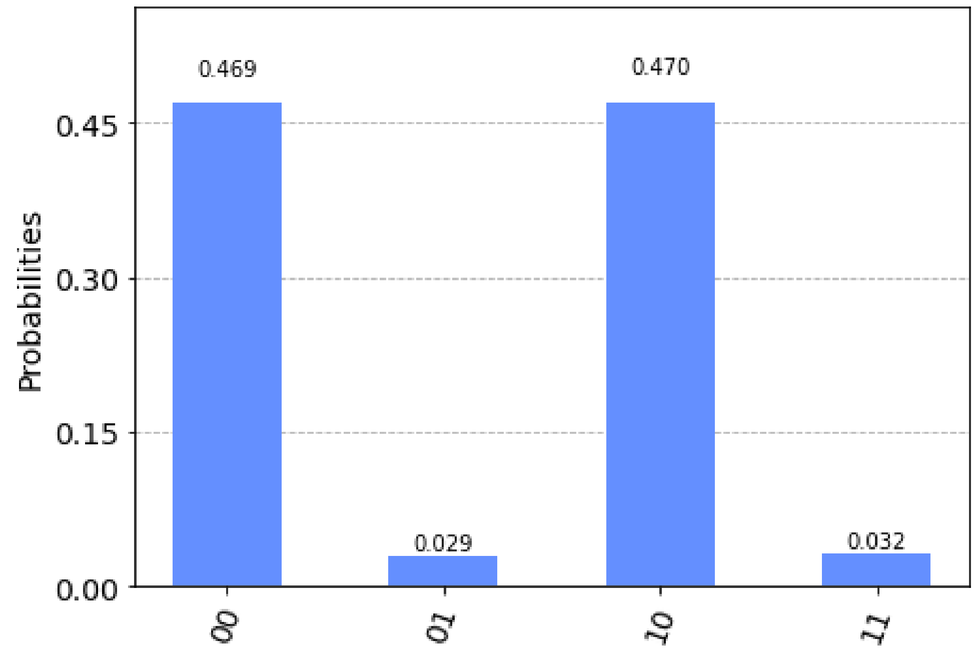
```
from qiskit import QuantumCircuit
from numpy import pi

circuit = QuantumCircuit(2,2)
circuit.sx(0)
circuit.rz(pi/2, 0)  # sx * rz(pi/2) = h
circuit.cx(0,1)      # control, target
circuit.cx(1,0)
circuit.cx(0,1)      # Ez a 3 CNOT kapu a csere
circuit.measure(0,0)
circuit.measure(1,1)

print(circuit)

job = backend.run(circuit)
result = job.result()
plot_histogram(result.get_counts())
```



Braket jelöléssel levezetve:

$$|00\rangle \xrightarrow{\text{I}\otimes\text{H}} \frac{\mathbf{1}}{\sqrt{\mathbf{2}}}\,|\mathbf{00}\rangle + \frac{\mathbf{1}}{\sqrt{\mathbf{2}}}\,|\mathbf{01}\rangle \xrightarrow{\text{SWAP}} \frac{\mathbf{1}}{\sqrt{\mathbf{2}}}\,|\mathbf{00}\rangle + \frac{\mathbf{1}}{\sqrt{\mathbf{2}}}\,|\mathbf{10}\rangle$$

A mérési eredmények:



7.2. Tervezési stratégiák

Egy kvantum algoritmus írásakor sok mindent figyelembe kell vennünk. A félév második fele, és innen a jegyzet is kvantum algoritmusokról szól, implementációról, működési elvekről. Látni fogjuk, hogy hogyan lehet hatékony egy kvantum algoritmus, hogyan lehet kihasználni az eddig tanultakat, hogy klasszikusnál hatékonyabb megoldást találjunk 1-1 problémára.

Az eddig látottak - bár mélységében nem mentek bele a matematikába, absztrakcióba - átadtak egy modellt, egy felfogást. Egy új rendszert, egy dobozt, amiben gondolkodhatunk, alkothatunk. Ennek a doboznak nézzük most meg pár további tulajdonságát, algoritmus tervezési szempontból.

7.2.1. Elkódolás és kiolvasás, dekoherencia

Az amplitudó és fázis minden qubitnek kétszer annyi információ tárolására ad lehetőséget, mint amennyit egy klasszikus bit tud. Tehát ha van 5 qubites rendszerünk, abban egyszerre $\mathbf{2^5}$ bitnyi információt tudunk tárolni. Könyvek, videók, újságok ki szokták emelni, hogy egy koherens 400 qubites rendszerben kicsit több (bitnyi) információt lehet eltárolni, mint amennyit tárol az egész univerzum.

Az első probléma amibe belefutunk, hogy ezt nehéz kihasználni, mert az igaz, hogy ennyit el tud tárolni, de kiolvasáskor, méréskor csak egy klasszikus bitstringet kapunk. Számolhatunk exponenciális mennyiségű információval, kezelhetjük, transzformálhatjuk, de amint megmérjük, az információ nagyrésze elveszik. Akkor mire jó egyáltalán? Számításokat tudunk vele végezni. *Nekünk* kell úgy tervezni az algoritmust, hogy futás közben kihasználjuk a lehetőségeket, de mérés előtt olyan állapotot kell előállítanunk, amit megmérve a kapott eredmény megfelel a szándékainknak. Lényegében csak a return value kell logaritmikusan kicsi legyen az algoritmus futás közben rendelkezésre álló tárigényéhez képest.

Másik oldalról a kezdőállapotunk is egy klasszikus bitstring, itt épp a 000...0. Ez sem tárol túl sok információt. Viszont létezik módszer, aminek a segítségével a kezdőállapotba már exponenciálisan sok információt tudunk elkódolni. Persze ez is egyfajta előfeldolgozó algoritmus, ami pontatlan lehet, és a fő algoritmusunktól veszi el a QPU idejét. Ahogy fut az algoritmusunk, egyre veszít a pontosságából, és idővel szétesik a kvantum

rendszerünk, ez a dekoherencia. Ezért a kapusorozataink mélysége (hossza) nem lehet túl nagy, és a több qubites kapuk száma még nagyobb jelentőséget kap.

Az algoritmusaink nagyrésze így is fog kinézni: klasszikus állapotból indulva, a kvantumjelenségeket kihasználva eljutunk egy olyan állapotba, amiből klasszikus állapotokat kapva (akár egy szuperpozícióból) választ kapunk a kérdéses problémára.

7.2.2. Kvantumszámítógépek

A jelenlegi kvantumszámítógépek nem túl nagyok. Tesztelni lehet rajtuk az algoritmusainkat, kicsiben. Valódi alkalmazásokra sok esetben várni kell még nagyjából 2026-2028-ig.

Addig pedig nagyon sok fejlődés vár még az eszközökre. Hibajavítás terén 2021. májusában történt egy nagyobb áttörés, az az egyik legjobban kutatott terület.

A szupravezető qubites számítógépek nem a legpontosabbak, skálázni sem feltétlenül könnyű őket, de nagy remények vannak afelé, hogy sikerül. Mellette viszont fejlődik több egyéb architektúra, a legígéretesebb közülük a fotonikus kvantumszámítás. Sokan gondolják úgy, hogy az lehet a jövő. Akármi is legyen, programozóként a jelenségek megértése, és az architektúrafüggetlen kvantumprogramok írása a feladatunk.

Ez viszont alacsony szinten még nem teljesen megoldható. Megoldható, de hatékonytalan. Az algoritmusok, amiket nézünk, univerzálisak, és nem is fogjuk egyes architektúrákon próbálgatni őket folyamatosan, de tisztában kell legyünk a határokkal, amit például a báziskapukészlet, több qubites kapuk hibája, qubitek elhelyezkedéséből adódó csere műveletek jelentenek.

7.2.3. Időbonyolultság

A kvantum világában maga a futásidő nem annyira releváns mérték, főleg, mert nem lehet túl hosszú, ha kicsi pontatlanságot szeretnénk. A klasszikus időkomplexitás, kapusorozat hossza persze fontos egy darabig, de futásidőt és hatékonyságot nehezen lehet vele pontosan mérni. Valamekkora hibahatárral számolva viszont lehet, és mérjük is.

Számításaink során találkozni fogunk két fontosabb fogalommal, jelöléssel[4]. Az első a **BQP** bonyolultsági osztály. Ez nagyjából a klasszikus P osztály kvantum megfelelője (de nem esik egybe vele, nem szabad keverni őket). A jelentése Bounded-error Quantum Polynomial time. A kvantum algoritmusaink valamekkora hibával dolgoznak, a mérés is egy folytonos eloszlásból vett mintavételezésnek fogható fel. Ennek az osztálynak az a jelentősége, hogy az algoritmus hibája méréskor véges, azaz sokszor ismételve a futtatást és mérést, egyértelműen megkapjuk a probléma megoldását - polinomidőben.

A BQP-ben P benne van, illetve több NP-beli probléma is, például a faktorizálás. Ha NP-teljes probléma is lenne benne, az implikálná, hogy P=NP, és ilyet még senkinek sem sikerült találnia. Olyannyira, hogy a poszt-kvantum, teljesen klasszikus titkosítási protokollok alapjául jellemzően 1-1 NP-teljes probléma NP-teljessége mögötti ötlet lapul. Ilyen például az LP - ILP problémája kapcsán, hogy egy ismeretlen rácson teljesen véletlenszerűen letett ponthoz megtalálni a legközelebbi rácspontot nagyon nehéz.

A másik fontos fogalom a **lekérdezés bonyolultság**, vagyis query complexity. Vegyük a következő problémát: van egy függvényünk, ami minden inputra azonos választ (igazságértéket) ad, vagy az inputok felére ilyet, felére olyat. Például az előbbi lenne az [1,10] számhalmazon a f(x) = x>0 függvény. És a másodikra példa az f(x) = x>5. Kérdés, hányszor kell kiszámolnunk a függvény értékét, hogy megmondjuk, a kettő közül melyik a függvényünk? Klasszikusan legalább hatszor. Egy kvantum algoritmussal elég egyszer.

Ez a lekérdezés bonyolultság fő kérdése: hányszor kell "megkérdeznünk" (futtatnunk) a függvényt/adatbázist, hogy biztos választ kapjunk.

[4]Akinek nem volt még bonyája, vagy bővebben utánanézne, javaslom Iván Szabolcs bonyolultságelmélet jegyzetét.

Kvantum gyorsítás példákon

8.1. Lekérdezés-modell

Az előző fejezet végén láthattuk, mi is a lekérdezés-bonyolultság. Szükségünk van rá, mert a klasszikus időbonyolultság nem igazán jól írja le az egyes problémák tulajdonságait a kvantumvilágban. Erre fogunk most példát látni.

Az első két algoritmusunk pont a kvantumszámítás lekérdezés-alapú előnyét használja ki. Persze olyat nem mondhatunk, hogy kizárólag a lekérdezések száma a fontos, ez nem mondana magában semmit a klasszikus időbonyolultsággal való összehasonlításnál. Miért? Gondoljunk bele, legyen adott egy SAT probléma. Ezt egy klasszikus algoritmus ki tudja számolni minden esetben pl. értékadások sorozatával, szintaktikusan kiértékelve (Pl. logikáról emlékezhetünk a DPLL algoritmusra), $O(2^n)$[1] időben, miközben a formulát egyszer számolja végig, lényegében 1 darab lekérdezést hajt végre.

A formula méretétől függetlenül. Meg persze meg lehet oldani úgy is, hogy minden lehetséges inputnak adunk egy próbát, és ha kielégítő értékadást találunk, adjuk vissza hogy kielégíthető. Ez $O(2^n)$ lekérdezés, de egy-egy lekérdezés hatékonyan kiszámítható[2].

Tehát nem mindegy azért az sem, hogy egy probléma lekérdezés komplexitása és időbonyolultsága milyen viszonyban áll egymással.

8.2. Kvantum párhuzamosítás

Elsőként kicsit foglalkoznunk kell az elmélettel is. Bár sokaknak kényelmetlen először az elméletet megnézni, ebben az esetben elengedhetetlen. Próbáljuk követni, a példáknál vissza fogunk térni rá, kiemelve az elméleti részt is. Lassan haladva végigmegyünk a lekérdezések lényegén. Aki szeretné kihagyni, ugorjon a Deutsch-Jozsa algoritmushoz. Ez az elméleti rész azért hosszú, hogy ne legyen tömény, de érthető, követhető maradjon végig.

[1] n a literálok száma by def, de itt a DPLL-től lehetne a klózoké is. (És egyébként $(2^n)^1 = 2^n$)

[2] Persze, ettől NP-beli, a tanú polinomidőben számítható

Nézzük a jelöléseket. Tegyük fel, hogy van egy (sima, klasszikus) algoritmusunk, amely négyzetre emelést végez, azaz

$$\boldsymbol{f(x) = x^2}$$

Ezt felírhatnánk úgy is, hogy

$$\boldsymbol{f \,:\, x \mapsto x^2, \quad x} \in \mathbb{N}$$

Most tartsuk meg a műveleteinket, és szimplán a természetes számokat jelöljük el, mint számjegyek sorozata:

$$f \in \{0, 1, 2...9\}^n \mapsto \{0, 1, 2...9\}^m : x \mapsto x^2$$

Például $32^2 = 1024$, azaz valamilyen $n = 2$ számjegyű szám négyzete lett valamilyen $m = 4$ számjegyű szám. A négyzetreemelés csak egy példa volt, ideje általánosítani bármilyen függvényre a természetes számokon. Most, hogy $\boldsymbol{f}$ lehet bármi, így fog kinézni a jelölésünk:

$$f \;:\; \{0, 1, 2...9\}^n \mapsto \{0, 1, 2...9\}^m$$

Fontos megjegyezni, hogy míg $\boldsymbol{f}$ eddig egy darab függvényt jelentett, most bármilyen függvény lehet, amelyik kielégíti a "szabályt", azaz a leképezésünket. Na de nekünk qubitjeink vannak, térjünk is át bináris jelölésre:

$$f \;:\; \{0, 1\}^n \mapsto \{0, 1\}^m$$

Ha bármennyi bitünk lehet, és rövidebb bitstringet feltöltjük vezető nullákkal, lényegében van egy akármilyen függvényünk, amit kettes számrendszerben ki tudunk számolni. Itt most bár "átírtuk" a tizes alakokat kettesre, akár a természetes számokról is leléphetünk, mivel ábrázolni tudjuk (akár végtelen biten) a valós számokat is.

Elérkeztünk tehát egy általános bináris függvénydefinícióhoz. Mivel a függvényünk nem feltétlenül injektív[3], viszont a kvantumszámítás unitér és reverzibilis, ezért pontosan ahogy a CNOT kapunál láttuk, kell egy másik qubit, jelen esetben másik regiszter az eredmény tárolására. Ilyen formán bármilyen $\boldsymbol{f : \{0,1\}^n \mapsto \{0,1\}^m}$ függvényhez meg tudunk adni egy kvantum áramkört[4], ami megvalósítja ezt a függvényt ilyen formában:

$$\boldsymbol{|x\rangle\,|0\rangle \;\mapsto\; |x\rangle\,|f(x)\rangle}$$

[3] Azaz nem biztos, hogy 1:1 leképezést valósít meg, pl. lehetne $\boldsymbol{f}$ a konstans 5 függvény is, az pedig minden számhoz az 5-öt rendeli

[4] Akár kizárólag Toffoli kapukból!

Sőt, akkor már felírhatjuk matematikailag az áramkör unitér operátorával (mondjuk U-val) is:

$$\mathrm{U}_f|\boldsymbol{x}\rangle\,|0\rangle \;=\; |\boldsymbol{x}\rangle\,|\boldsymbol{f}(\boldsymbol{x})\rangle$$

Kicsit visszautalva a négyzetes példához, nézzük 6 (110) négyzetét:

$$\mathrm{U}_f|\mathbf{110}\rangle\,|\mathbf{000...0}\rangle \;=\; |\mathbf{110}\rangle\,|\mathbf{100100}\rangle$$

Azaz[5]:

$$\mathrm{U}_f|\mathbf{110000000}\rangle \;=\; |\mathbf{110100100}\rangle$$

Ez most kiszámol nekünk bármilyen függvényt. Lényegében U_f bemutatja, hogy a kvantumszámítás valóban Turing-teljes számítási modell. De pl. a Brainfuck nyelv is Turing-teljes programnyelv, mégse használja senki. Sőt, még PHP-t is használnak emberek, szóval ettől még nem lett hirtelen se jó, se párhuzamos semmi.

Itt jön képbe a szuperpozíció, és a Hadamard transzformáció. Erre tanultunk már egy összegképletet itt. Most már kicsit ismerősebben jön tehát szembe a képlet, hogy $\boldsymbol{n}$ bit egyenlő szuperpozíciója:

$$\mathrm{H}^{\otimes \boldsymbol{n}}\,|\mathbf{0}^{\boldsymbol{n}}\rangle \;=\; \frac{\mathbf{1}}{\sqrt{\mathbf{2}^{\boldsymbol{n}}}} \sum_{\boldsymbol{z}\in\{\mathbf{0},\mathbf{1}\}^{\boldsymbol{n}}} |\boldsymbol{z}\rangle$$

Ami azt jelenti, hogy az összes $\boldsymbol{n}$ hosszú bitstring $\boldsymbol{z}$ 1-1 egyenlően pici valószínűséggel lehet megmérve, egyszerre vannak benne ebben a szuperpozícióban. Folytatva ezt, persze meg lehet azt is csinálni, hogy egy $\boldsymbol{m}$ hosszú bitstringnek csak az első $\boldsymbol{n}$ bitjét tesszük egyenlő szuperpozícióba (és a maradék $\boldsymbol{k}$-t nem):

$$\mathrm{H}^{\otimes \boldsymbol{n}}\,|\mathbf{0}^{\boldsymbol{m}}\rangle\,\left|\mathbf{0}^{\boldsymbol{k}}\right\rangle \;=\; \frac{\mathbf{1}}{\sqrt{\mathbf{2}^{\boldsymbol{n}}}} \sum_{\boldsymbol{z}\in\{\mathbf{0},\mathbf{1}\}^{\boldsymbol{n}}} |\boldsymbol{z}\rangle\,\left|\mathbf{0}^{\boldsymbol{k}}\right\rangle$$

Ez pedig azt jelenti, hogy például az előző példát nézzük, a négyzetest, ahol 3 input bitünk volt és 6 output, összesen 9, ez a szuperpozíció ott úgy nézne ki, hogy

$$\frac{\mathbf{1}}{\sqrt{\mathbf{8}}} \sum_{\boldsymbol{z}\in\{\mathbf{0},\mathbf{1}\}^{\mathbf{3}}} |\boldsymbol{z}\rangle\,|\mathbf{000000}\rangle$$

[5]Note: bázisállapotok közt $|1\rangle\,|0\rangle = |10\rangle$, tenzorszorzatról van szó

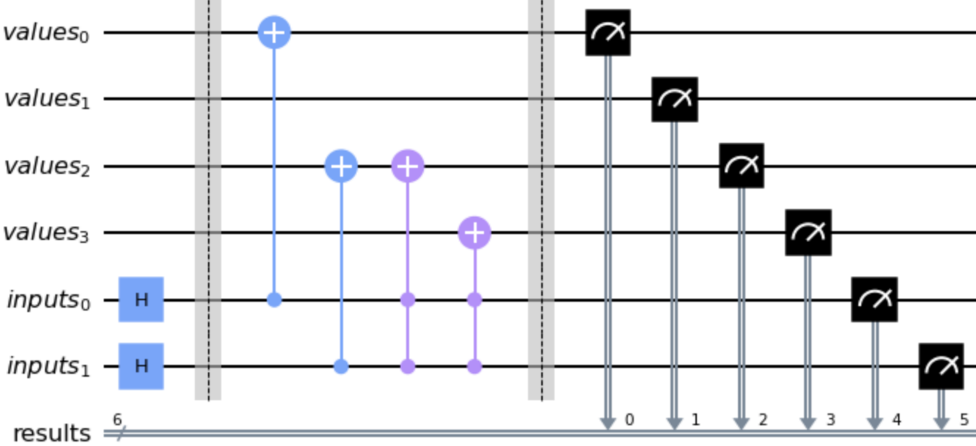
Azaz egyenlő eséllyel a {000000000, 001000000, 010000000, 011000000, 100000000, 101000000, 110000000, 111000000} halmazból valamelyik lesz az eredmény, ha megmérjük.

Most, hogy lépésről lépésre végighaladtunk a jelöléseken, matekon, elérkeztünk a lényeghez. Nézzük, mit csinál U_f akkor, amikor az inputunk az összes lehetséges input egyszerre, azaz azoknak a szuperpozíciója:

$$\mathrm{U}_{\boldsymbol{f}}\left(\frac{\mathbf{1}}{\sqrt{\mathbf{2}^{\boldsymbol{n}}}}\sum_{\boldsymbol{x}\in\{\mathbf{0},\mathbf{1}\}^{\boldsymbol{n}}}|\boldsymbol{x}\rangle\,|\mathbf{00}\ldots\mathbf{0}\rangle\right) = \frac{\mathbf{1}}{\sqrt{\mathbf{2}^{\boldsymbol{n}}}}\sum_{\boldsymbol{x}\in\{\mathbf{0},\mathbf{1}\}^{\boldsymbol{n}}}|\boldsymbol{x}\rangle\,|\boldsymbol{f}(\boldsymbol{x})\rangle$$

Ha eddig sikerült követnünk mindent, akkor - bár ránézésre bonyolult kifejezésünk van, minden része érthető kell legyen. Nézzük a példánkat tehát, miről is van szó, ahol $\boldsymbol{f}$ a négyzetre emelés. A példában csak 2 bites inputokkal foglalkozunk, így is kellően nagy lesz az áramkör.

Lássuk tehát először az áramkört:



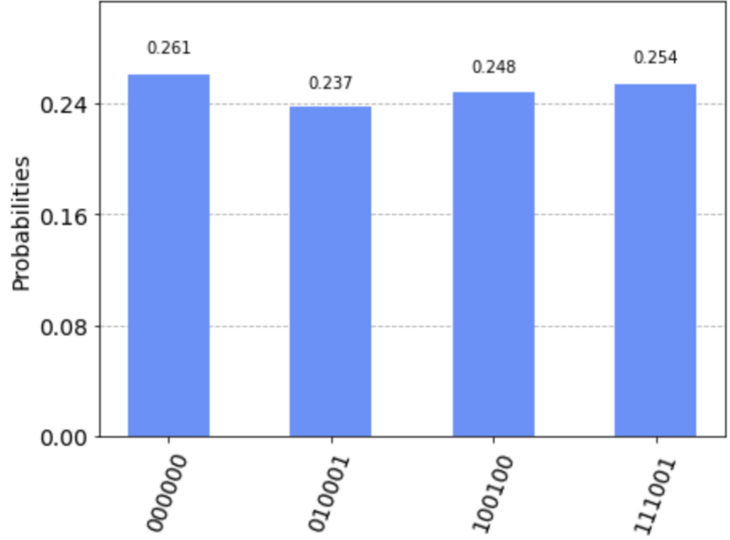
A regiszterek számozása a Qiskit indexelése miatt ilyen. Tehát az input regiszterünket előkészítjük egyenlő szuperpozícióba, majd jön U_f a két barrier között, ami kiszámolja nekünk a négyzetreemelés függvényt[6]. Utána pedig mérés. Nem bonyolult áramkör, de 6 biten már ennek is lehet 64 féle kimenetele, amiből még éppen látható amit szeretnénk.

[6]Ez az áramkör hasból keletkezett oda, de bármilyen függvényre írhat bárki ilyet, kicsiben nem olyan nehéz

Lássuk mindezt kóddal:

```
# Init
x_reg = QuantumRegister(2, 'inputs')
fx_reg = QuantumRegister(4, 'values')
m_reg = ClassicalRegister(6, 'results')

c = QuantumCircuit(fx_reg, x_reg, m_reg)

# circuit
c.h([x_reg[0], x_reg[1]]) # Inputok egyenlő szuperpozícióban
c.barrier()
c.cx(x_reg[0], fx_reg[0]) # Négyzetre emelés
c.cx(x_reg[1], fx_reg[2]) # Négyzetre emelés
c.ccx(x_reg[0], x_reg[1], fx_reg[2]) # Négyzetre emelés
c.ccx(x_reg[0], x_reg[1], fx_reg[3]) # Négyzetre emelés
c.barrier()
c.measure([0,1,2,3,4,5], m_reg)
```

Ha mindent jól csináltunk, akkor az első két bit (valóban első kettő, ezért a felcserélt regiszterek) az összes lehetséges kombinációban kell legyenek, mögöttük pedig négybiten a négyzetük. Mindez pedig a 4 lehetséges input-output pár egyenlő szuperpozíciójában.



Fontos, hogy nem kapjuk meg az *összes* lehetséges megoldást azonnal. Minden mérésnél a négy lehetséges kimenetel egyikét fogjuk kapni (ha teljesen hibátlan, zajtalan a kvantumszámítógépünk). Tehát a függvény *egy* lehetséges megoldását.

8.3. Standard lekérdezés

Meg kell jegyezni itt, hogy a "négyzet" függvény, tehát az U_f operátor (áramkör) adott volt, tekinthetjük blackboxnak is. Kap egy inputot, azon kiszámol egy meghatározott outputot *valahogy*. Tekinthetjük a függvény lehetséges input-output párjait egy adatbázis elemeinek is, amit 1-1 lekérdezéssel érünk el. Innen ered a query complexity elnevezés is. Az ilyen valamit bárhogy kiszámító blackbox "eszközt" nevezzük orákulumnak, vagyis hogy a terület terminológiájánál maradjunk, oracle-nek.

A standard query lényege az, hogy van egy valamilyen inputunk, és mi annak az i-edik bitjét szeretnénk megtudni. "Megkérdezzük a mindenttudó Orákulumot, tessék szépen mondani, mi áll ott a hatodik helyen" jellegűen. Formálisan már ismerős lehet a képlet:

$$O_x \;:\; |i, 0\rangle \mapsto |i, x_i\rangle$$

Itt az első regiszter, i egy n bites szám (bitstring, az index amire kíváncsiak vagyunk), a második regiszter 1 bites, és x i-edik bitje.

A standard query első, n-bites regiszterét address (cím) regiszternek, az utolsó egybitest pedig target (cél) regiszternek hívjuk, elég beszédes a nevük.

8.4. Fázis lekérdezés

Kérdés, mi történik, ha az address regiszterben szuperpozícióban lévő qubitek vannak. Ez is egy érdekes koncepció, kvantum ram kellene hozzá, és hogy az adatot is kvantumrendszerekben tároljuk[7]. Ez jelenleg még nem valóságos és felhasználható dolog.

Másik kérdés, mi történik, ha a target regiszterben már van a qubitnek valamilyen $|b\rangle$ állapota. Ekkor a standard query pontosan ugyanazt csinálja, mintha nulla lenne ott; binárisan hozzáadja egybiten a megtalált bitet. Lényegében egy xor műveletet hajt végre

$$O_x \;:\; |i, b\rangle \mapsto |i, b \oplus x_i\rangle$$

Ebben nincs semmi fázis. A fázis például egy mínuszjellel tud megmutatkozni egyes bázisállapotok előtt. Phase query (fázis lekérdezés) egy ilyen dolog, a formája:

$$|i\rangle \mapsto (-1)^{x_i}\,|i\rangle$$

[7]Ehhez egy óriási előrelépés történt 2021. augusztus elején, létrehozták az időkristályokat sikeresen.

A működése olyan, mintha csak egy standard query target regiszterét a $|-\rangle$ állapotba téve hajtanánk végre a lekérdezést. Viszont fontos, hogy itt a target bitnek csak segítő szerepe van, sem az inputnak, sem az outputnak nem része, de az algoritmus használja (és nem változtatja meg!).

Lássuk, hogyan működik elméletben. Erre fogunk látni később gyakorlati alkalmazást is. Elsőként a target regisztert a $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ állapotba kell állítsuk. Ezt egy negálással és egy Hadamard kapuval meg is tudjuk tenni, mert $|-\rangle = \mathrm{HX}\,|1\rangle = \mathrm{H}\,|1\rangle$. Ekkor tehát a lekérdezés így néz ki:

$$O_x(|i\rangle\,|-\rangle) \;=\; |i\rangle\,\frac{1}{\sqrt{2}}\,(|x_i\rangle - |1 - x_i\rangle) \;=\; (-1)^{x_i}\,|i\rangle\,|-\rangle.$$

Ez kicsit nehezen megérthető, de gondoljuk végig, ha $x_i = 0$, akkor a target regiszterünk marad ahogy volt. Ha $x_i = 1$ akkor pedig csak egy mínuszjelet kap az állapot, az egész target regiszter, amit kiemelhetünk előre. Ez lesz a phase query, és úgy fogjuk jelölni, hogy $O_{x,\pm}$, jelölve hogy a fázist kérdezzük csak le.

8.5. Deutsch-Jozsa algoritmus

Na tehát akkor alkalmazzuk az eddigieket. A problémánk adott, legyen egy oracle-ünk, ami vagy minden inputra ugyanazt adja vissza (azaz konstans függvény), vagy az inputok felére 1-et, felére 0-t (balanced).

Nézzünk például ilyet, legyen 4 bitünk, azaz a [0,15] intervallumunk. Az oracle-ben lehet mondjuk az $f : \{0,1\}^4 \mapsto \{0,1\}$ függvényünk az $f_b(x) = x \bmod 2$ balanced függvény. Ez az inputok felén 0-t ad, felén 1-et. És lehet mondjuk a másik az $f_c(x) = 1$ konstans függvény.

A problémánk az, hogy kitaláljuk, pontosan melyik függvényt rejti az oracle[8]. Erről már volt szó, klasszikusan legalább 9-szer kellene futtatnunk az oracle-t hogy biztosan megmondjuk, f_b vagy f_c van benne. Most nézzünk egy kvantum algoritmust, aminek elég 1 lekérdezés, egyszer futtatni az oracle-t ennek az eldöntéséhez.

[8] Általános esetben nem tudjuk pontosan, melyik ez a két függvény

8.5.1. Implementáció

Most az implementációval fogunk először foglalkozni, utána pedig a matematikai magyarázatával. Az implementáció tetszőleges számú bitre általánosítva van, mi most 4 bites konstans vagy kiegyensúlyozott függvénnyel fogjuk megnézni.

Kezdjünk rögtön az oracle-lel. Ez elvileg adott az algoritmusunk futtatásakor, sőt, nem is tudjuk pontosan hogyan működik. De sajnos vagy nem sajnos, ha odamegyünk a portás nénihez vagy lakótársunkhoz, családtagjainkhoz hogy légyszi adjanak nekünk egy balanced oracle-t akkor nincs nagy esélye, hogy kapunk ilyet, szóval nekünk kell most összeraknunk.

```python
# n = qubitek száma
def oracle(constant = True, n=4):
    oracle = QuantumCircuit(n+1)
    if constant:
        # Mindenképp 1
        oracle.x(n)
    else:
        # Esetek felében 1,
        # esetek felében 0
        for i in range(n):
            oracle.cx(i,n)

    oracle.to_gate()
    oracle.name = "Oracle"
    return oracle
```

Az oracle számunkra láthatatlan kell legyen, így egy új, (n+1)-bites kaput definiálunk, amelyben elrejtjük az oracle-t alkotó áramkört. Bár a függvényünk 4 bites, az oracle 5 bites kell legyen ilyenkor, ugyanis tartalmaznia kell a 4 input bitet és az 1 output, target bitet is. Bár a phase oracle nem változtat a target biten, számol vele, ezért kelleni fog mind az 5 bit.

Kontans esetben csak negáljuk a target bitet, tehát minden lehetséges inputon 1-et ad outputként az oracle[9]. Balanced esetben az inputbeli 1-esek számát adjuk vissza (mod 2) most.

Folytassuk az áramkörünk implementációjával. Ahogy a párhuzamosításnál láthattuk, elkészítjük az összes lehetséges inputunk egyenlő szuperpozícióját, illetve mivel phase

[9] Az oracle igen, az áramkörünk nem, a target bit nem alap (0) állapotban érkezik majd ide

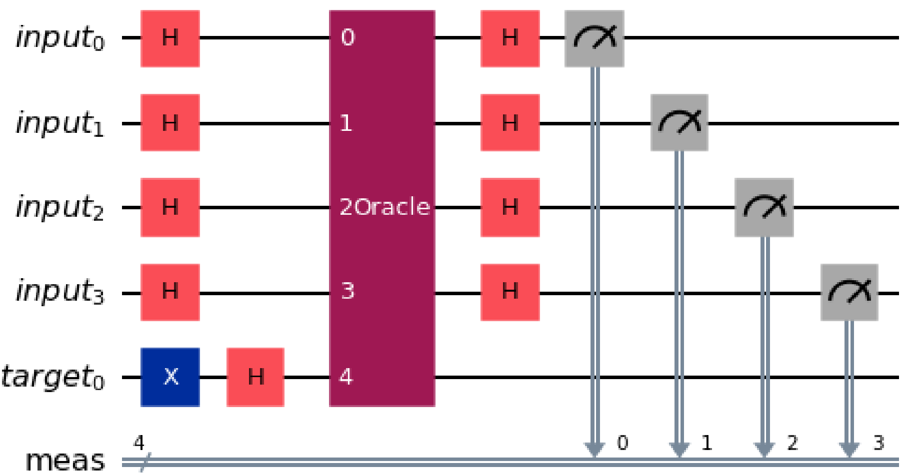
queryt szeretnénk végrehajtani az oracle segítségével, a target bitünket a $|-\rangle$ állapotba készítjük elő.

A queryhez szükségünk lesz még egy Hadamard transzformációra a végén is, amelyik a kezdő szuperpozícióból ekészíti nekünk az állapotot, amit megmérve ki tudjuk olvasni a megoldást.

Saját magunk által definiált kaput a QuantumCircuit.append() metódussal tudunk az áramkörhöz adni, illetve meg kell neki adni, hogy melyik qubitekre szeretnénk rátenni.

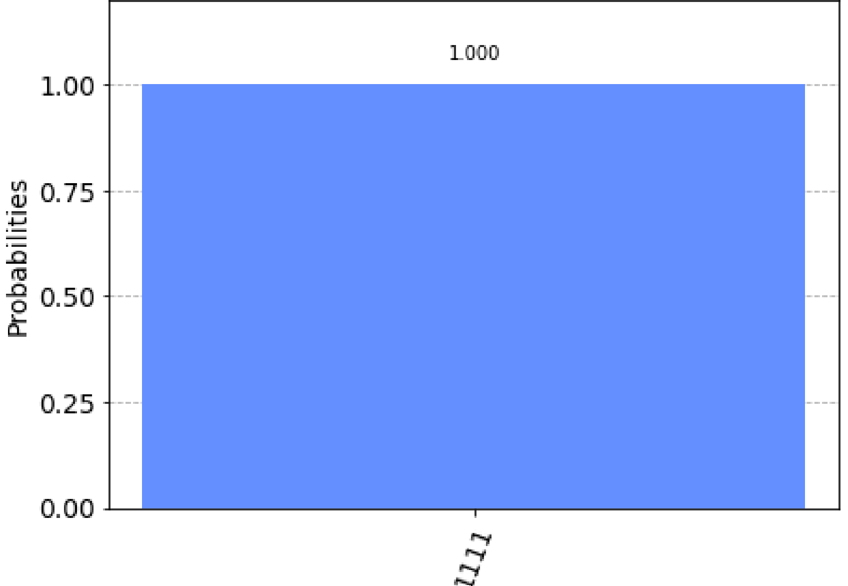
Lássuk tehát:

```
n=4

iqr = QuantumRegister(n, "input")
tqr = QuantumRegister(1, "target")
mcr = ClassicalRegister(n, "meas")
c = QuantumCircuit(iqr, tqr, mcr)

c.h(range(n)) # szuperpozíció
c.x(n)
c.h(n) # |-> állapot

c.append(oracle(constant=False, n=n), range(n+1))

c.h(range(n)) # szuperpozíció megszüntetése
c.measure(range(n), mcr)

c.draw('mpl', style='iqx')
```



Szimulátoron futtatva pedig meg tudjuk nézni az eredményeket. Ha jól csináltuk, amikor a függvény konstans, akkor a $|0000\rangle$ bitstringet kell kapnunk, minden egyéb esetben valami mást. Hogyha a függvény balanced, akkor nem kaphatunk csupa 0 bitstringet. Tehát az eredmények, balanced esetben, a csupa 0 inputból kiindulva:

```
from qiskit.providers.aer import QasmSimulator
from qiskit import transpile

backend = QasmSimulator()
c_t = transpile(c, backend)
counts = backend.run(c_t).result().get_counts()
plot_histogram(counts)
```



Note: a futtatás előtt kell egy transpile parancs, ugyanis sem a szimulátor, sem a kvantumszámítógépek nem tudják maguktól értelmezni az általunk definiált új utasítást, az oracle-t. A transpile parancs ezt (és az egész áramkört) lefordítja a paraméterben kapott eszköz utasításkészletére.

És valóban nem a csupa 0 bitstringet kaptuk. A következőkben láthatjuk ezt tisztán matematikailag is. Már csak a formalizmus miatt is, előtte érdemes átnézni a fejezet korábbi részeit.

8.5.2. Magyarázat

Vezessük tehát le, mi történik. Elsőként egy fontos megjegyzés. Eddig nem foglalkoztunk azzal, mi történik, ha egy valamilyen általános bitstringen hajtunk végre Hadamard transzformációt - ez pedig kelleni fog nekünk az algoritmus végén.

Legyen $i \in \{0,1\}^n$ bitstring. Abban az esetben, amikor $|i\rangle = |0^n\rangle$, már láttuk hogy néz ki:

$$\mathrm{H}^{\otimes n}|0^n\rangle = \frac{1}{\sqrt{2^n}} \sum_{j \in \{0,1\}^n} |j\rangle$$

Tehát minden n hosszú bitstring egyenlő szuperpozíciója. Most nézzük mi történik, ha nem a csupa 0 állapottal indulunk:

$$\mathrm{H}^{\otimes n}|i\rangle = \frac{1}{\sqrt{2^n}} \sum_{j \in \{0,1\}^n} (-1)^{i \cdot j} |j\rangle$$

Ahol $i \cdot j$ a két bitstring belső szorzata[10]. Nem annyira bonyolult, végülis csak a Hadamard és a számítási között váltunk át, tehát például ha $|i\rangle = |011\rangle$ akkor a Hadamard transzformáció eredménye $\mathrm{H}^{\otimes 3}|i\rangle = |+\rangle \otimes |-\rangle \otimes |-\rangle = |+--\rangle$, mintha a biteken egyesével hajtottuk volna végre a kaput.

No tehát akkor számoljuk végig a Deutsch-Jozsa algoritmust:

$$\begin{aligned} |0^n\rangle & \xrightarrow{\mathrm{H}^{\otimes n}} \frac{1}{\sqrt{2^n}} \sum_{i \in \{0,1\}^n} |i\rangle \\ & \xrightarrow{O_{x,\pm}} \frac{1}{\sqrt{2^n}} \sum_{i \in \{0,1\}^n} (-1)^{x_i} |i\rangle \\ & \xrightarrow{\mathrm{H}^{\otimes n}} \frac{1}{2^n} \sum_{i \in \{0,1\}^i} (-1)^{x_i} \sum_{j \in \{0,1\}^n} (-1)^{i \cdot j} |j\rangle \end{aligned}$$

Nézzük az amplitudóját a $|j\rangle = |0^n\rangle$ állapotnak a végén. A második összeg minden esetben[11] 1 együtthatót ad, mert $i \cdot 0^n = 0$ és $(-1)^0 = 1$. Így a végén azt kapjuk, hogy ez az amplitudó

$$\frac{1}{2^n} \sum_{i \in \{0,1\}^i} (-1)^{x_i} = \begin{cases} 1 & \text{ha } x_i = 0 \text{ minden } i\text{-re}, \\ -1 & \text{ha } x_i = 1 \text{ minden } i\text{-re}, \\ 0 & \text{ha } x \text{ kiegyensúlyozott.} \end{cases}$$

[10]elemenkénti szorzatok összege, vektorszorzata

[11]minden i esetén

Mérésnél az amplitudó négyzete lesz a valószínűségünk, tehát konstans függvény esetén 1 valószínűséggel a csupa 0 bitstringet fogjuk kapni. Balanced esetben pedig 0 valószínűséggel fogjuk ezt kapni.

Az algoritmusnak ebben a leírásában nem szerepelt a phase query által target bitnek használt regiszter, de a queryhez szükség van rá, az operáció magába rejti ezt. Explicit ki lehetne írni oda az utolsó bitet, de az algoritmus leírása szempontjából nem fontos.

Nagy sűrűségű kódolás, hibajavítás

9.1. Teleportálás

A kvantum teleportálás kicsit kifinomultabb dolog, mint azt az ember elsőre gondolná a neve alapján. Egy egyszerű kvantum-klasszikus hibrid kommunikációs protokollról van szó. Szükség van hozzá klasszikus adatátvitelre is, tehát maga a teljes kommunikáció nem "teleportálás" valójában.

A kvantum adatátvitel része viszont az. Kvantum állapotot klasszikus úton átküldeni nagyon bonyolult, például meg se mérhetjük a rendszert, fenntartani a koherens kvantumrendszert az egész úton pedig nagyon nehéz, elvégre a számítógépek belsejében sem sikerül jelenleg annyi ideig. És ezen kívül végtelen sok klasszikus bit kellene a kellő pontossághoz. A kommunikáció alapja, hogy ha például van két qubitünk, összefonódott állapotban, és fizikailag elvisszük őket távolra egymástól, attól még a kapocs megmarad köztük, az összefonódás nem szűnik meg. Viszont mérés esetén szétesik a rendszer, elárulva *mindkét* qubit állapotát.

A protokoll úgy néz ki, hogy:

0. Alaphelyzet: Alicenek van egy valamilyen $|q0\rangle$ qubitje aminek az állapotát szeretné átküldeni Bobnak klasszikus csatornán. Alice és Bob ezen kívül osztozik egy EPR páron, azaz két összefonódott qubiten, ezek legyenek $|q1\rangle$ és $|q2\rangle$. Tehát Alicenál van $|q0\rangle$ és $|q1\rangle$. Utóbbi összefont állapotban a Bobnál lévő qubittel, $|q2\rangle$-vel.

1. lépés: Alice egy CNOT kaput alkalmaz, amin a saját állapot bitje a control, az összefont pedig a target. Majd egy Hadamard kaput tesz a saját, átküldésre szánt bitjére.

2. lépés: Alice megméri a 2 qubitjét, majd a két klasszikus bitet (legyen c_0 és c_1), amit kapott eredményként, átküldi Bobnak.

3. lépés: Bob a kapott két klasszikus bit alapján kapukat tesz a saját bitjére:

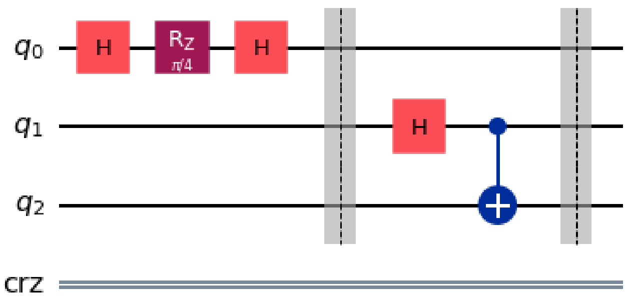
- Ha $c_0 = 1$, akkor Z kaput
- Ha $c_1 = 1$, akkor X kaput

Ezzel Bob qubitjének pontosan ugyanaz lesz az állapota, mint Alice-é volt. Note: kvantumállapotokat másolni nem lehet, Alice qubitje ekkorra már összeomlott, meg lett mérve, tehát az információ fizikailag is átkerül Bobhoz, és megszűnik létezni Alice-nál.

9.1.1. Implementáció

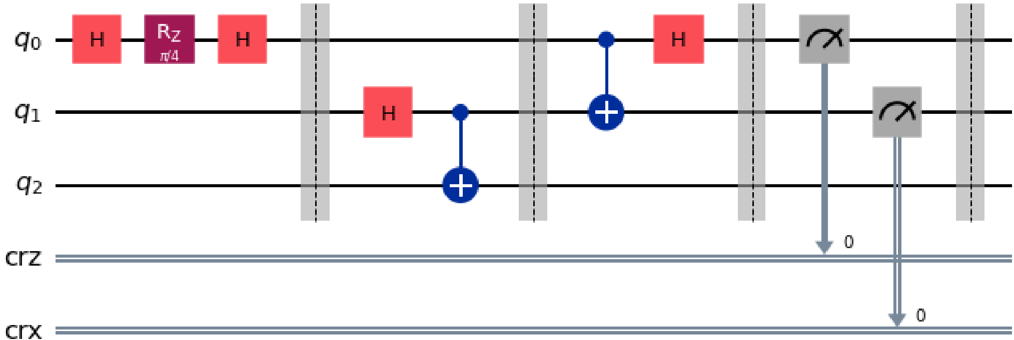
Kezdjük a 0. lépéssel, adjunk valamilyen állapotot Alice qubitjének, és állítsuk össze a Bell-állapotot a második két qubiten:

```
qr = QuantumRegister(3, 'q')
crz = ClassicalRegister(1, 'crz')
crx = ClassicalRegister(1, 'crx')
qc = QuantumCircuit(qr, crz, crx)

# Előállítunk valamilyen állapotot
def alice_state():
    qc = QuantumCircuit(1)
    qc.h(0)
    qc.rz(pi/4, 0)
    qc.h(0)
    return qc

# Áramkör, a 0. lépésnek megfelelően
def preparation(qc):
    qc.compose(alice_state(), [qr[0]], inplace=True)
    qc.barrier()
    qc.h(qr[1])
    qc.cx(qr[1],qr[2])
    qc.barrier()
    return qc

qc = preparation(qc)
qc.draw('mpl', style='iqx')
```



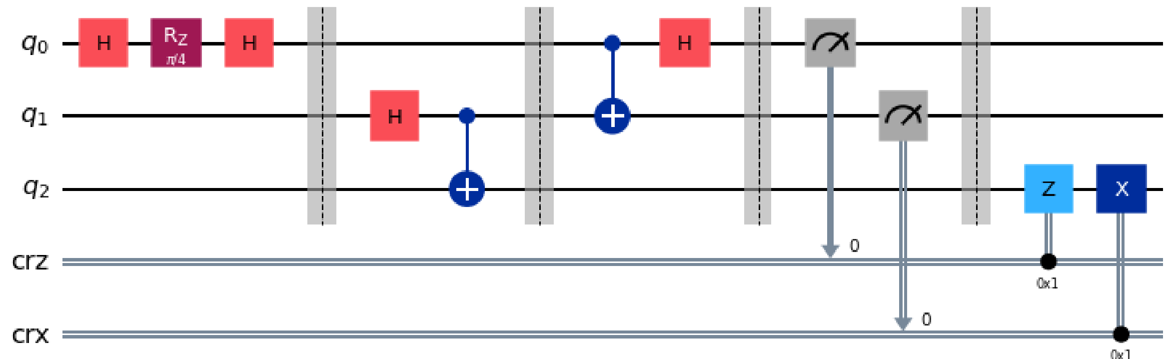
Utána az első lépésünk alapján tegyük rá Alice kapujait, illetve a méréseket. A méréseknek két külön regiszter van fenntartva, ezek klasszikus úton lennének fizikailag átküldve Bobhoz.

```
def alice_actions():
    qc = QuantumCircuit(2)
    qc.cx(0,1)
    qc.h(0)
    return qc

qc.compose(alice_actions(), [0,1], inplace=True)
qc.barrier()
qc.measure([0,1], [0,1])
qc.barrier()
qc.draw('mpl', style='iqx')
```



Ekkor már minden adott, ha átküldjük a két klasszikus bitet Bobnak. Ő ezután a saját bitjét módosítja a klasszikus bitek alapján. Ezt Qiskitben úgy tudjuk megtenni, hogy kontrollált kaput használunk, klasszikus control bittel. Ennek kicsit más a szintaxisa, ez látszik a következő képen:

```
# Bob actions
def bob_actions(qc, crz, crx):
    qc.z(2).c_if(crz, 1)
    qc.x(2).c_if(crx, 1)
    return qc

qc  = bob_actions(qc, crz, crx)
qc.draw('mpl', style='iqx')
```



9.1.2. Magyarázat

Legyen Alice qubitje, amit át akar küldeni valamilyen $|\psi\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle$ állapotban. Ezen kívül Alice és Bob osztozzanak egy összefont EPR páron, ami most $(\text{CNOT})(\text{H} \otimes \text{I})|00\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ legyen. Tehát a teljes rendszer állapota, fizikailag bárhol is legyenek a qubitek

$$(\alpha_0 |0\rangle + \alpha_1 |1\rangle) \otimes \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle).$$

Ez volt a nulladik lépés, az előkészület. Jöjjön tehát az első, ahol Alice egy CNOT kaput tesz a saját, illetve a megosztott qubitekre, majd egy Hadamard kaput tesz a saját qubitjére:

$$(\text{H} \otimes \text{I})(\text{CNOT})\left((\alpha_0 |0\rangle + \alpha_1 |1\rangle) \otimes \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)\right) =$$

$$\begin{aligned}
&\tfrac{1}{2}|00\rangle(\alpha_0|0\rangle + \alpha_1|1\rangle) + \\
&\tfrac{1}{2}|01\rangle(\alpha_0|1\rangle + \alpha_1|0\rangle) + \\
&\tfrac{1}{2}|10\rangle(\alpha_0|0\rangle - \alpha_1|1\rangle) + \\
&\tfrac{1}{2}|11\rangle(\alpha_0|1\rangle - \alpha_1|0\rangle) \quad .
\end{aligned}$$

Az első két qubit Alice-nál van, a második, az alpha amplitudókkal Bobnál. Tehát átkerültek az amplitudók Bobhoz, viszont ki is kell tudnunk olvasni. Ha Alice megméri a qubitjeit, a 4 kombináció valamelyikét kapja. Bob pedig a $|\psi\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle$ állapotot szeretné kapni. Tehát ha Alice mér, és 00-t kap, akkor Bobnál eleve ez a qubit lesz. Hogyha Alice az első qubitjét 1-nek méri, akkor - ahogy az állapoton is látszik - egy Z kapu fog kelleni, eltüntetve a negatív fázist. Ha Alice a második qubitjét méri 1-esnek, akkor pedig X kapu kell, tehát fel kell cserélni a $|0\rangle$ és $|1\rangle$ állapotok amplitudóit. Persze ha 11-et mér, akkor mindkét kapura szükség van.

9.2. Nagy sűrűségű kódolás

Egy másik lehetséges kommunikációs forma (vagy épp információ elkódolási forma) a superdense coding, aminek a segítségével most megnézzük, hogyan lehet 1 qubit átvitelével 2 klasszikus bitnyi információt átküldeni.

A teleportálásnál 2 klasszikus bitet használtunk 1 qubit teleportálásához, itt 1 qubitet használunk 2 klasszikus bithez.

Ehhez pedig összefonódást fogunk használni. Két összefont qubit egyikét megkapja Alice, másikat Bob. Alice végrehajt valamit a saját qubitjén, utána átküldi Bobnak. Bob feloldja az összefonódást és megméri mindkét bitet. Tehát lényegében 2 qubitre van szükség, de kommunikáció csak az egyiken folyik, a másik pedig az összefonódás segítségével "beszáll" a kommunikációba, így 2 bitnyi klasszikus információt lehet elkódolni abba az egy átküldött qubitbe.

Kiindulásnál elkészítjük a $\frac{|00\rangle+|11\rangle}{\sqrt{2}}$ állapotot, majd:

Az elkódolás kísértetiesen hasonlít a teleportáláshoz, de most Alice kezében vannak az X és Z kapuk a Qiskit textbook ábráján:

Intended Message	Applied Gate	Resulting State ($\cdot\sqrt{2}$)
00	I	$\|00\rangle + \|11\rangle$
10	X	$\|01\rangle + \|10\rangle$
01	Z	$\|00\rangle - \|11\rangle$
11	ZX	$-\|01\rangle + \|10\rangle$

Majd ezután átküldi a saját qubitjét is Bobnak. Bob kapui pedig így oldják fel az elkódolt információt:

Bob Receives:	After CNOT-gate:	After H-gate:
$\|00\rangle + \|11\rangle$	$\|00\rangle + \|01\rangle$	$\|00\rangle$
$\|01\rangle + \|10\rangle$	$\|11\rangle + \|10\rangle$	$\|10\rangle$
$\|00\rangle - \|11\rangle$	$\|00\rangle - \|01\rangle$	$\|01\rangle$
$-\|01\rangle + \|10\rangle$	$-\|11\rangle + \|10\rangle$	$\|11\rangle$

Az implementáció nagyon egyszerű, az eddigiek alapján 2 qubiten csak egy összefonódást kell felépíteni, majd a control biten Alice operációit, utána pedig a két biten Bobét.

9.3. Hibajavítás, problémák

Az egyik legjelentősebb területéről van szó a korai kvantumszámításnak, ahol jelenleg tartunk. "Tudunk-e teljesen hibamentes kvantumszámítást létrehozni?". Klasszikus számítógépeken hozzá vagyunk szokva, hogy under the hood, minden hardveresen előforduló hibát kijavítanak, és nekünk nem kell azzal foglalkoznunk, hogy például egy változó inicializálása után más lesz az értéke[1]. De ez nem annyira triviális dolog! Ahhoz hogy így működhessenek a rendszereink, nagyon sok fejlődés kellett, mérnöki, fizikai és matematikai oldalról is. Gondoljunk csak bele, mennyivel kisebbek és pontosabbak is a tranzisztoraink mint régen, a hibajavító kódolások mennyire hatékonyak.

Van egy nagy előnye a klasszikus számításnak, amikor hibajavításról beszélünk, és ez a diszkretizáció. Klasszikusan bitjeink vannak, amik vagy 0-k vagy 1-ek. Hiba vagy történik egy biten, vagy nem. Vagy átfordul, vagy nem. A kvantumszámítás nem ilyen. Itt folytonosan lehet 0 és 1 bármilyen szuperpozíciója a bitünk állapota, folytonosan lehet fázisa is. Ha használunk egy kaput, például egy egyszerű X kaput, ott is felmerül a kérdés, hogy negáljam - oké, de *mennyire*?

[1]Ha a két szobával arrébb ülő Józsi nem force pusholt fel valamit utánunk ami megbabrálja pont azt a memóriaterületet...

9.3.1. Hibák fajtái

Persze többféle hardveres hiba létezik. És amellett, hogy a hardverek is folyamatosan fejlődnek, a hibák se mind folytonos és láthatatlan állapotátmenetek. Mi most három nagyobb csoportra osztva fogjuk vizsgálni ezeket a hibákat.

Első csoportunk lesz a **mérési hiba**. Ide tartozik az inicializálási hiba is. Mindkét esetben "külső segítséggel" kell állítanunk valamit a rendszeren, akár beállítani, akár megfigyelni. És ez a külső hatás az érzékeny rendszerre már deformálja az állapotunkat.

Második csoportunk a **kapuhiba**. A kapuk is, hasonlóan az előző ponthoz, külső hatások a rendszerünkre. Teljesen hibátlanul fizikailag implementálni őket nagyon nehéz (lehetetlennek tartott) feladat. A szupravezető qubitek kapui összetett folyamatok a klasszikus logikai kapukhoz képest. Képzeljünk el egy hintát, kistesónk éppen beleült és kéri, hogy hajtsuk. Megtehetjük, hogy időnként játékból megfogjuk a hintát, várunk egy kicsit, majd visszalökjük. Ha nézzük a klasszikus analógiáját, meglökjük azon a magasságon valamennyire, kábé a régivel azonos lendülettel folytatja a hinta az útját. Ezt a tranzisztorunk simán lefordítaná ugyanarra a bitre, elvégre neki teljesen mindegy, hogy 4,5V vagy 4,2V éri, abból 1-es lesz. Ha vesszük a kvantum analógiáját, nincs ilyen diszkretizációt végrehajtó eszközünk. Sőt, még a hintának az se mindegy, hogy ha megfogtuk, utána mikor engedjük el, az előzőhöz képest mennyire marad meg az inga jellegű ritmusa (mozgásának a fázisa). Tehát pontosan, precízen éreznünk kell, mekkora erővel és mikor szabad kistesónkat újra pályára tennünk. Sok gyakorlással sikerül, ugye? Nem. Egyszer sikerülhet tökéletesen, pár alkalommal még lehet mázlink, de folyamatosan sosem fogjuk tudni tökéletesen a megcélzott erővel, megfelelő időben elindítani.

A harmadik csoportunk a **zaj**. Egyéb hibája a hardvernek, amit nem szándékos külső behatással idézünk elő. Visszatérve az előző példára, minden lökéssel módosítunk a hinta pályáján, belenyúlunk. A lökéseinknek van valamilyen pontossága, hibája. De ha nem lökjük meg a hintát, akkor egy idő után meg fog állni[2]. Nagyon hasonlóan működik a qubittel is, egy idő után szépen lassan elveszíti az állapotát, vagyis átalakul "magától". Ahogy a hinta se tesz azért hogy megálljon, a kvantum rendszerünk se tesz azért, hogy ez megtörténjen. De ahogy a hintának az alkatrészei is súrlódnak, hat rá a gravitáció és a légellenállás, a kvantum rendszerünknek is vannak fizikai problémái ami miatt ez történik.

[2]Tegyük fel, hogy az utasa nem hajtja

9.3.2. Vegyes állapotok

Szép magyar fordítás, de az irodalom, így mi is az eredeti, "mixed states" kifejezést fogjuk használni. Az állapotoknak egy szép idealizálása, matematikai formalizálása az állapotvektorok formája. Ugyanakkor a természet nem ennyire "tökéletes", egy vektorral nem tudjuk teljesen pontosan leírni 1-1 qubitünk pontos állapotát. Az állapotokat viszont leírhatjuk sűrűségi mátrixszal (density matrix). Ez egy pozitív szemidefinit mátrix, nagyon hasonlít a neurális hálók tévesztési (confusion) mátrixára. Sőt, ha sokszor mérünk, akkor a szokásos négyzetre emeléstől eltekintve ugyanolyan mátrixnak is kell kijönnie, tehát főátlóban annak az esélye lesz, hogy pont az adott (oszlop által meghatározott) állapotban mérjük a rendszert. Máshol pedig, hogy mekkora az esélye, hogy adott állapot helyett valamelyik másikban mérjük.

Logikus lenne, hogy minden sor és oszlop négyzetösszege 1 legyen, elvégre valószínűségekről van szó. Viszont (valamekkora mértékig) összefonódott állapotok esetén ez sérül, jellemzően a valóságban ezek 1-nél kisebb számok.

Sűrűségi mátrixokkal ugyanúgy lehet számolni, mint állapotvektorokkal, csak egy 2^n dimenziós vektor helyett[3] $2^n \times 2^n$ mátrixokkal kell számolni. A kurzus bevezető jellegű, ezért mi végig tiszta állapotokkal (pure states) foglalkozunk. Viszont jó látni, hogy ennél kicsit árnyaltabb a valóság. És ha például hibajavítással szeretne valaki foglalkozni, elkerülhetetlen, hogy vegyes állapotokkal számoljon.

9.3.3. Hibajavítás

Haladjunk végig a csoportokon egyesével, és nézzünk egy-egy kis ízelítőt az egyes javítási módszerekről. Végig szupravezető áramkörös kvantumprocesszorokkal foglalkozunk. Léteznek másfélék is, és a javítási módszerek teljesen eltérők más-más architektúrákon.

A mérési hibákra egy ismételt méréseken alapuló error mitigation nevű módszer. Ez azon alapul, hogy a mérésekből próbáljuk visszakövetkeztetni az áramkör hibáját, majd ezt a hibamátrixot invertálva klasszikusan lényegében leradírozni a hibát. Ez nem tökéletes, mert többszöri mérésen és experimentálisan működik, nem feltétlenül hatékony a mátrix invertálás miatt, de a mérési, és egész áramkör alatti futási hibát nagyon le tudja csökkenteni. A Qiskit textbookban bővebben szó van erről: Error mitigation.

[3] n a qubitek száma

A kapuhibákat többféle módon is javíthatjuk. Az egyik típusú hiba a klasszikus bitflip jellegű. Ebből lehet bitflip, azaz X típusú, ami felcseréli a $|0\rangle$ és $|1\rangle$ amplitudóját, és lehet phase flip, azaz Z típusú, ami - mint egy Z kapu - negálja a $|1\rangle$ fázisát. Az ilyen típusú hibára nyújtanak egy megoldást a surface kódok. Ezek úgy működnek, mint a klasszikus hibajavítás, lényegében X és Z típusú parity qubiteket hoznak be, megfelelő, négyzetrácsos elrendezésben (innen jön a surface) kötik össze őket fizikailag. Ezzel a redundanciával ez a két fajta hiba hatékonyan detektálható(!), és utólag javítható.

A hardveres hibákra, ahol exponenciálisan csökkenő a pontosság, a qubitek fizikai javítása nyújthat megoldást, legalábbis folyamatos fejlődést. Ebben a témában Zlatko Minev (IBM) vezet egy heti rendszerességű podcastet, a linkje itt található.

Kvantum Fourier Transzformáció

10.1. Fourier transzformáció

A Fourier transzformáció egy nagyon jelentős, sok helyen használt dolog. Röviden, intuitívan az a motiváció mögötte, hogy egyes függvénytranszformációkat nehéz végrehajtani egyféle "elkódolásban" és könnyű másféle "elkódolásban".

Egy analógia: ezt az absztrakt fejezetet nehéz megérteni egy buliban leülve a sarokba, könnyű lehet például egy csendes helyen egyedül, elmélyedve. A fő motiváció, hogy mindent hatékonyan csináljunk. Tehát ebben az esetben menjünk el erre a csendes helyre, tanuljuk meg, majd menjünk vissza bulizni a tudással - ez még mindig időhatékonyabb, mint adott esetben a buliban megérteni az anyag egy kis részét is[1]. És függvényeknél ugyanígy: ha van egy nehéz transzformáció, akkor "vigyük át" egy másik bázisba, rendszerbe, ahol könnyű végrehajtani, majd "hozzuk vissza" a megoldást.

Ez az "átvitel" sok esetben a Fourier transzformáció. Analízis szempontjából a periodikus függvények nagyrészt nehéznek számítanak. Viszont Fourier térben könnyű számolni velük. De például zajos jelek szétbontásánál is hasznos tud lenni.

10.1.1. Matematikai alapok

Tegyük fel, hogy van egy akármilyen egyváltozós függvényünk $g(x)$, ami folytonos minden pontban, alaphalmaza a komplex számok. A Fourier transzformáció felbontja a függvényt szinuszoid függvények összegére.

Elsőre nehéz lehet elképzelni. Nézzünk egy kisebb példát. Tegyük fel, hogy az alaphalmazunk nem $\mathbb{R}$, hanem csak a [0,1,2,3,4] halmaz. A függvényünk az alaphalmaz minden eleméhez rendel egy értéket. A Fourier terünk alaphalmaza a szinuszoid hullámok. Tehát az új függvényünk $G(y)$ minden lehetséges szinuszoid függvényhez rendel egy értéket, hogy a függvények összege pont az eredeti függvényünk legyen. Az inverz transzformáció pedig ezt hajtja végre visszafele, amivel visszakapjuk az eredeti függvényünket.

[1] *Ha ez bármikor fontos lenne bárkinek ilyen helyzetben...*

Fontos megjegyezni, hogy ez nem 1:1 leképezés. Itt a kiindulási függvényünk 1 pontjához a Fourier tér összes pontját rendeljük. Tehát "adott pontban milyen szinuszoid függvények mekkora súllyal építik fel a függvényünk értékét".

Sőt, komplex számaink vannak, mert egy szinuszoid hullámot két dolog definiál: az amplitudója ("milyen magas"), és a fázisszöge ("mennyire szinusz, mennyire koszinusz?"). Tehát minden függvényértékünkhöz hozzárendelünk végtelen sok komplex számot, amiket össze kell adjunk, hogy visszakapjuk az eredeti függvényértékünket. Ezeknek a komplex számoknak a fázisa jelöli ki, melyik szinuszoid függvényről van szó, az amplitudója pedig hogy mennyire járul hozzá az adott ponthoz (mennyit ad hozzá az összeghez, "mennyire fontos függvény ő abban a pontban").

Lássuk a matekot. Legyen az előbbi jelölésből $g(x) = x_j$ és $G(y) = y_k$.

$$y_k = \int x_j e^{-2\pi ijk} dj$$

Inverz számításkor csak a mínusz jelet kell elhagynunk és normalizálnunk. Itt a g(x)-ek az eredeti függvényértékeink, az exponenciált tagok pedig a szinuszoid komponensek az új függvényünkben. Az integráltól nem kell megijedni, csak annyit jelent, hogy "minden j-re összeadva". Hamarosan el is dobjuk az integrál jelölésünket.

Az exponenciális tagot tehát jelöljük el ω-val. Továbbá diszkretizálni fogjuk az egyenletet, tegyük fel, hogy N függvényértékünk van összesen. Így tehát legyen $\omega_N^{jk} = e^{-2\pi ij\frac{k}{N}}$. Így tehát a transzformációnk így néz ki:

$$y_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j \omega_N^{jk}$$

Ehhez pár megjegyzés. A sok betű ellenére szép interpretációnk van. ω_N pontosan az N. egységgyök, j és k pedig egész számok. Tehát ω_N^{jk} az N. egységgyök $j \cdot k$-adik hatványa. Komplex egységgyökök hatványainak pedig egy szép tulajdonsága, hogy mind az egységkörön helyezkednek el, egyenlő távolságra egymástól.

Visszautalva a második fejezet mátrix exponenciálós részéhez, a Fourier trafó egy mátrixon is értelmes, és azt teszi vele, hogy a sajátértékeit "szétszórja" az egységkörön. Ennek a geometriai tulajdonságnak érdekes következményei vannak, ezekről még lesz szó.

10.2. QFT

Definiálni tudjuk a FT-t egy N-dimenziós mátrixon is, a következő módon:

$$\boldsymbol{F_N} = \frac{1}{\sqrt{N}} \begin{bmatrix} & \vdots & \\ \dots & \omega_N^{jk} & \dots \\ & \vdots & \end{bmatrix}$$

Ez a mátrix unitér, így kvantum kapuként is tudjuk használni. Ez lesz a QFT, vagyis kvantum Fourier transzformáció. Ezt így még nehezen lehet értelmezni, nézzük meg az $\boldsymbol{N = 2}$ esetben, mit kapunk.

$$\boldsymbol{F_2} = \frac{1}{\sqrt{2}} \begin{bmatrix} & \vdots & \\ \dots & \omega_2^{jk} & \dots \\ & \vdots & \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} \omega_2^{0\cdot 0} & \omega_2^{0\cdot 1} \\ \omega_2^{1\cdot 0} & \omega_2^{1\cdot 1} \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \mathrm{H}$$

Tehát 1 qubiten a QFT csak egy Hadamard kapu. Bár több qubiten kicsit bonyolultabb az implementáció, lényegében a számítási bázis tere, és a fázistér között "válthatunk" egy ilyen transzformációval, ami egy igen hasznos művelet.

A transzformáció egy $|x\rangle = |x_1\rangle\,|x_2\rangle \dots |x_n\rangle$ n bites állapoton[2] pedig így néz ki:

$$\begin{aligned} F_N\,|x\rangle &= \frac{1}{\sqrt{N}} \bigotimes_{k=1}^{n} \left(|0\rangle + e^{2\pi i x/2^k}\,|1\rangle\right) \\ &= \frac{1}{\sqrt{N}} \left(|0\rangle + e^{\frac{2\pi i}{2}x}|1\rangle\right) \otimes \left(|0\rangle + e^{\frac{2\pi i}{2^2}x}|1\rangle\right) \otimes \dots \otimes \left(|0\rangle + e^{\frac{2\pi i}{2^n}x}|1\rangle\right) \end{aligned}$$

Magyarul, a Bloch-gömbön most az x-y síkon (vízszintes) vagyunk a körvonalon, és ha transzformáljuk például a $|\mathbf{101}\rangle = |5\rangle$ állapotot, akkor az első qubiten, 5/8 $\times 2\pi$, azaz $\frac{10}{8}\pi$ fázisunk lesz, és a qubit egyenlő szuperpozícióban lesz a számítási bázisban. A második qubiten 5/4 $\times 2\pi$, azaz $\frac{10}{4}\pi$, azaz 2π/4 fázisunk lesz, a harmadik qubiten pedig 5/2 $\times 2\pi$, azaz $\frac{10}{2}\pi$, azaz π fázisunk lesz, mert a fázis 2π periodikus.

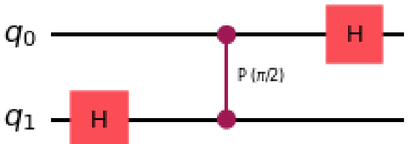
[2] x_1 a "most significant", tehát egy "$\mathbf{100}$" bitstringben ő az $\mathbf{1}$

10.3. QFT implementációja

Egy qubiten már láttuk, hogy a QFT csak egy Hadamard kapu. Intuitívan az oka egyszerű, egyrészt szuperpozícióba teszi a qubitet, másrészt ha a qubit értéke 1 volt, akkor a fázisa π lesz[3], ha 0 akkor pedig nem változik.

Két qubiten a QFT így néz ki:

```
F4 = QuantumCircuit(2)

F4.h(1) # 2. bit
F4.cp( pi/2, 0, 1) # CP kapu, az aktuális bit a target(!)

F4.h(0) # 1. bit
F4.draw('mpl', style='iqx')
```



Két qubiten már egy kicsit bonyolultabb a dolog, F_4-et kellene valahogy felépítenünk. Méghozzá úgy, hogy "skálázzuk" fel a meglévő áramkört.

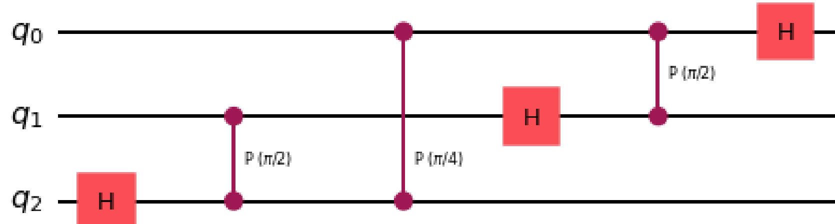
A második qubiten is kelleni fog majd a H kapu, ahogy az előző pontban néztük, mit csinál a transzformáció, lényegében egyenlő szuperpozícióban lévő qubiteknek ad fázist. És kelleni fog egy fázis kapu, méghozzá mivel a fázist összeadjuk lényegében a többi qubit (relatív) fázisával, ez egy CP, azaz kontrollált fázis kapu kell legyen. Olyan, mint a CNOT, ami az első bitet békén hagyja, a második bithez pedig hozzáadja az első értékét (modulo 2). A CP kapu az első bit értéke alapján fázist ad a másodiknak, méghozzá paraméteresen.

A paraméter pedig π lesz, leosztva a kettő hatványaival. Itt épp a qubit indexére emelt kettőhatvánnyal, tehát ebben az esetben ez $\frac{\pi}{2^1}$. Fontos kiemelni, hogy a CP kapu target bitje az aktuális bit, és nem a control bitje.

[3]A relatív fázisa, tehát a $|0\rangle$ fázisa változatlan marad, a $|1\rangle$ fázisa pedig π lesz

Három qubiten pedig ez lesz az áramkör:

```
F8 = QuantumCircuit(3)

F8.h(2) # 3. bit
F8.cp(pi/2,1,2) # CP kapu a 2. bitről
F8.cp(pi/4,0,2) # CP kapu az 1. bitről

F8.h(1) # 2. bit
F8.cp( pi/2, 0, 1) # CP kapu az 1. bitről

F8.h(0) # 1. bit

F8.draw('mpl', style='iqx')
```



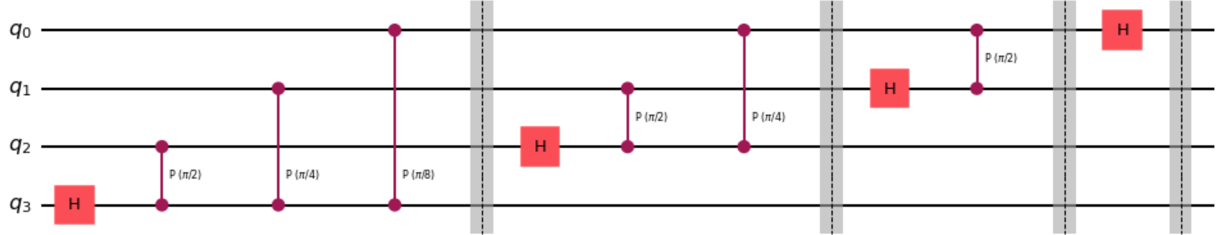
Az első két qubiten meghagytuk hátul a kétqubites verziót, csak skálázunk felfelé. A harmadik bitet úgy tesszük hozzá, hogy kell rá egy Hadamard kapu, és utána végigiterálunk a többi qubiten a CP kapuval, úgy, hogy a fázist egyre csökkentjük, tehát a paraméter $\frac{\pi}{2^s}$ lesz, ahol s azt jelöli, hanyadik qubit a control.

Persze ezt akárhány bitre ki lehet terjeszteni, és az is látszik, hogy a kontrollált kapuk száma négyzetesen fog skálázódni az új bitek számával. Ugyanakkor az n+1-ik bit már csak exponenciálisan kicsi plusz precizitást ad. Pont úgy, mintha mi egy tizedestörtet float vagy double típusban tárolunk el, a double "dupla" olyan pontos, de az a másik 32 bit exponenciálisan egyre kisebb értékeket ad hozzá, egyre kisebb mértékben növelve a pontosságot.

Meg fogjuk nézni általánosan is a transzformációt, illetve pár tulajdonságát utána. Fontos, hogy mivel a bitek száma az eredeti szám kettes alapú logaritmusával arányos, ezért a négyzetes skálázás még mindig logaritmikus marad a futásidőben, míg a legjobb klasszikus algoritmus, a Fast Fourer Transform (FFT), $O(n\ logn)$ időben fut.

Most lássuk paraméteresen, n biten:

```
def QFT(n):
    qc = QuantumCircuit(n)
    for i in range(n-1, -1, -1): # for ciklus n-1-től 0-ig
        qc.h(i)
        for j in range(i-1, -1, -1): # for ciklus i-1-től 0-ig
            qc.cp(pi/(2**(i-j)),j,i)
            # j a control, i a target
            # 2^(i-j)-vel osztunk
        qc.barrier()
    return qc

n = 4
qc = QFT(n)
qc.draw('mpl', style='iqx')
```

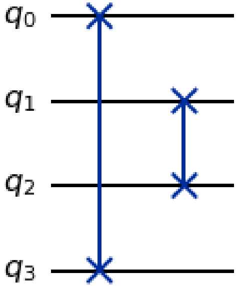


A fenti CP kapukon se látszik explicit, hogy melyik a control és melyik a target bitjük. Mindig a "felső", a kisebb indexű qubit a control, és a nagyobb indexű, amelyiken előtte a Hadamard kapu van, az a target.

A probléma ezzel a számábrázolásban rejlik. Miközben számítjuk a fázist, megfordítjuk a bitek sorrendjét. Persze ki lehet cserélgetni őket egyesével, és erre szükség is van, tehát szépen párosával, cserénként 3 CNOT költséggel meg kell cserélgetni a qubitek sorrendjét.

Tehát, hogy teljes legyen a QFT, a fenti kódrészlethez még hozzátartozik a teljes kvantumregiszter megfordítása:

```python
def SWAP(n):
    qc = QuantumCircuit(n)
    for i in range(floor(n//2)):
        qc.swap(i, n-1-i)
    return qc

n=4
qc = SWAP(4)
qc.draw('mpl', style='iqx')
```



Másik hátránya, hogy a kvantumszámítógépeken nincs minden qubit fizikailag összekötve a másikkal. Hogyha két nemszomszédos qubiten szeretnénk kétbites kaput végrehajtatni, akkor szépen egymás mellé kell tologatni az állapotokat, amíg egymás melletti fizikai qubiten nem lesznek. Ez is felfújja a CNOT költségét az áramkörnek.

Az inverz QFT sem különbözik sokban, csak a CP kapuk szögeit kell negálni. A következő algoritmusokban főként az inverz verziót fogjuk látni - elkészítjük a megoldásunkat a fázis térben, majd áttranszformáljuk a számítási bázisunkba és megmérjük.

Grover algoritmusa, Fázis becslés

11.1. Bevezetés: Grover algoritmusa

Az utunk a szubexponenciális gyorsítás felé újabb nagy lépcsőhöz érkezett. Az intuíció szerencsére nem bonyolult, így példával, implementációval kezdünk, majd belenézünk a matekjába is.

Grover kereső algoritmusáról van szó, ami talán a második legjelentősebb kvantum algoritmus. Kvadratikus gyorsítást nyújt.

Adott a probléma: van egy rendezetlen adatbázisunk, és meg szeretnénk benne találni egy elemet. Kicsit absztraktabban, legyen az adatbázis az első 100 természetes szám, 0-tól 99-ig. Benne is van mind a száz, de teljesen véletlenszerű sorrendben. Kérdés itt is, hogy hányszor kell lekérni 1 elemet az adatbázisból, hogy megtaláljuk amelyiket keressük?

Legrosszabb esetben persze végig kellene mennünk az egészen. De átlagos esetben is 50 számot le kellene kérnünk, ami $O(n)$[1] időt ad neki klasszikusan. Grover algoritmusa mindezt átlagosan $O(\sqrt{n})$ időben tudja megoldani, az amplitudók változtatgatásával.

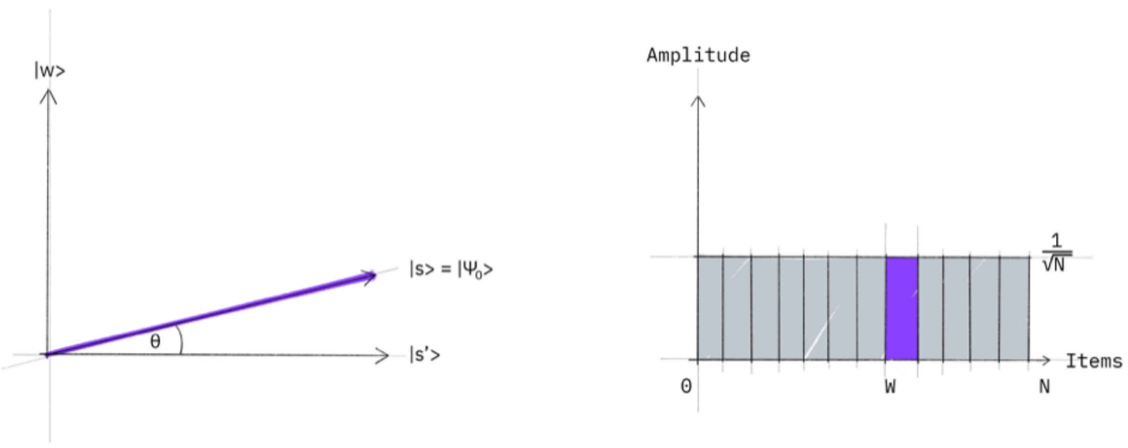
Feltételezve, hogy tudjuk, az adatbázisunk hány eleme kell nekünk (például hány zérushelye van a függvényünknek), meg tudjuk határozni a helyes/összes arányt $\frac{t}{N}$-t, ami még kelleni fog nekünk, ez legyen ε.

Az algoritmus két tükrözést használ. Ha a vektorokat nézzük, ki tudunk emelni egy vektort, legyen $|w\rangle$, amelyik a pontos megoldásunkat jelöli. Ki tudunk jelölni egy vektort, legyen $|s\rangle$ ami az egyenlő szuperpozíciót jelöli. Emellett szükségünk lesz egy harmadik állapotvektorra, $|s'\rangle$, amelyik egy olyan állapot ami a megoldásunkra merőleges, tehát lényegében a teljesen rossz megoldást kódolja el. A két tükrözés: indítva először az egyenlő szuperpozícióból, tükrözünk $|s'\rangle$ körül, majd az egyenlő szuperpozíció $|s\rangle$ körül. És ezt ismételjük. Egy idő után az állapotunk elég közel kerül $|w\rangle$-hez, így méréskor - ahol a távolsággal fordítottan arányos eséllyel mérjük ezt vagy azt - nagy eséllyel a megoldásunkat mérjük meg. Ez még egy nagyon száraz megfogalmazás, de megnézzük lépésenként, képekkel, hogy hogyan is működik az algoritmus.

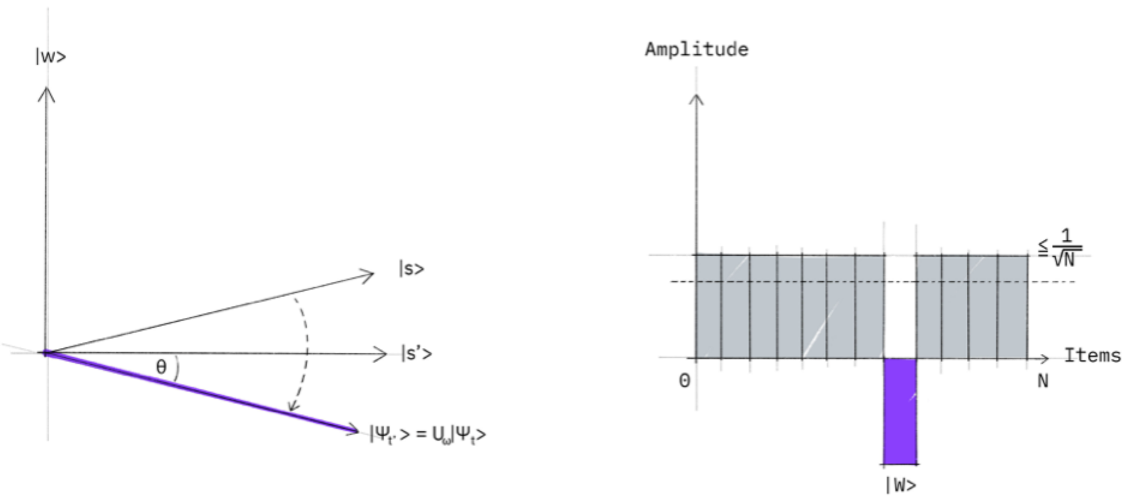
[1] n az adatbázis mérete, ami itt most 100.

Nézzük a lépéseket:

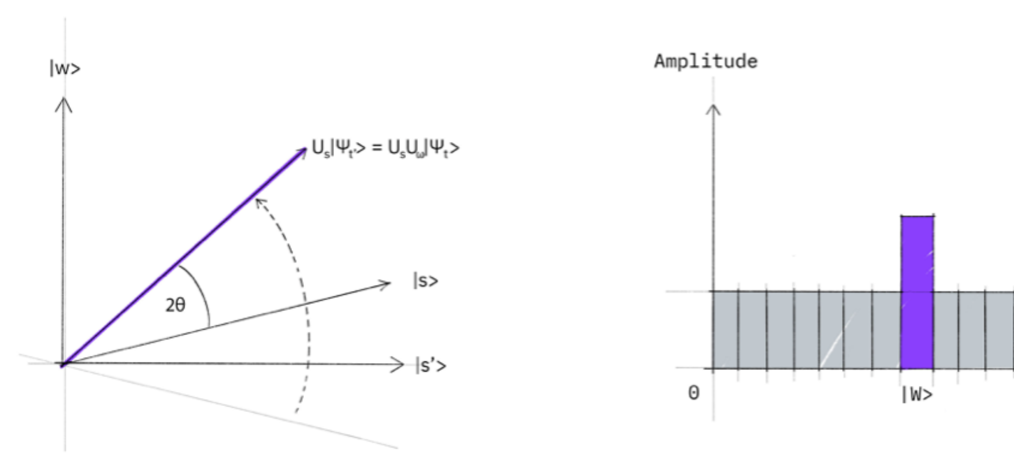
1. Állítsuk elő a kezdőállapotunkat, ami az egyenlő szuperpozíció $H^{\otimes n}|0^n\rangle$. Az adatainkból pedig paraméteresen meg tudjuk határozni, hogy $|s\rangle = sin(\theta)|w\rangle + cos(\theta)|s'\rangle$, és $\theta = arcsin\langle s|w\rangle = arcsin\frac{1}{\sqrt{N}}$.



2. Negáljuk a megoldásunk amplitudóját. Csak a megoldást tükrözzük, a "rossz" állapotunk körül. Kérdés, honnan tudnánk mi a megoldás, de ha visszaemlékszünk a Deutsch-Jozsa algoritmusra, ott megismerkedtünk a phase query fogalmával. Itt is erről van szó: a megoldás kap egy (-1)-es szorzót, a többi pedig marad.



3. Tükrözzünk az egyenlő szuperpozíció körül mindent. Az előző lépésben, a jobb oldali ábrából is láthatjuk, hogy a "rossz állapotok" amplitudói nőttek az egyenlőhöz képest, mivel a "jó állapot" amplitudója negálódott, és az összegük konstans kell legyen. Tehát ennél a tükrözésnél theta értéke nő, illetve a jó állapotunk amplitudója is nő, túlnő az egyenlőn. A többi állapoté pedig csökken.



A képek a Qiskit textbookból vannak, Grover algoritmusának leírásából. Ott is láthatók ezek, angol nyelven, részletesebben kifejtve.

A **2.** és **3.** lépéseket kell ismételni, a legközelebbi megoldáshoz $\frac{1}{\sqrt{\varepsilon}}$ alkalommal. Ez legrosszabb esetben, ha $\varepsilon = \frac{1}{N}$ akkor $\sqrt{N}$ lépés. Tehát a futásidő itt $O\left(\sqrt{N}\right)$ lesz.

11.1.1. Implementáció

Nézzünk egy egyszerű példát, ahol 2 bitünk van, tehát 4 elemű adatbázisunk. A 4 elemből legyen a "3"[2] a megoldás. Tehát lényegében maga a függvény, ilyen kicsiben egy klasszikus logikai ÉS műveletet valósít meg, de ez az információ nem áll rendelkezésünkre egy ilyen helyzetben. Úgy, mint Deutsch és Jozsa algoritmusában, itt is egy black box oracle a függvényünk.

Tehát akkor lépésekben: egyenlő szuperpozícióba tesszük a kvantum regiszterünket. Majd iteráljuk az oracle és diffuser operátorunkat egy előre kiszámított számú alkalom-

[2] azaz az "11" bitstring

mal, majd mérünk és reménykedünk. Négy bit esetén elég pontosan meg lehet határozni, hogy 1 iterációra lesz szükségünk.

1. lépés: szuperpozíció, és persze az áramkörünk inicializálása.

```
qc = QuantumCircuit(2)

# 1. lépés: egyenlő szuperpozíció
qc.h([0,1])
qc.draw('mpl', style='iqx')
```

2. lépés: oracle. Ehhez kell egy kis magyarázat. Az előző pontban láttuk, hogyan is szeretnénk az oracle-t előkészíteni: egy olyan operátor U_ω kell, ami minden állapotot békén hagy, kivéve a megoldásunkat - aminek ad egy negatív fázist:

$$\mathrm{U}_\omega \left|s\right\rangle = \mathrm{U}_\omega \frac{1}{2}\left(\left|00\right\rangle + \left|01\right\rangle + \left|10\right\rangle + \left|11\right\rangle\right) = \frac{1}{2}\left(\left|00\right\rangle + \left|01\right\rangle + \left|10\right\rangle - \left|11\right\rangle\right)$$

Nem nehéz észrevenni, hogy az oracle operátorunk mátrixa

$$\mathrm{U}_\omega = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} = \text{CZ lesz.}$$

Ehhez persze szükség van az előző lépésben az egyenlő szuperpozícióba inicializált áramkörre, tehát U_ω az egyenlő szuperpozíció $\left|s\right\rangle$ állapotra hat.

Elkezdhetjük építeni a Grover iterációnkat rejtő függvényünket egy CZ kapuval:

```
# 2. lépés: az Oracle
def oracle() -> QuantumCircuit:
    oracle = QuantumCircuit(2)
    oracle.cz(0,1)
    return oracle

# Készítjük a grover iterációnkat
def grover_iter(qc) -> QuantumCircuit:
    qc.compose(oracle(), [0,1], inplace=True)
    return qc

# Plot a mostani helyzettel
qc = grover_iter(qc)
qc.draw('mpl', style='iqx')
```

3. lépés: diffuser operátor. Igényel némi magyarázatot ez is. Eddig nem volt szó diffuser operátorról, csak mégegy tükrözésről. Mivel mi a $|\mathbf{00}\rangle$ állapotból kezdtünk, kell egy állapot, ami merőleges erre, amin át tükrözhetünk. A Deutsch-Jozsa algoritmusnál láttuk, hogy hogyan tudunk így tükrözni, egyenlő szuperpozícióban, Z kapukkal. Nekünk a "teljesen rossz" állapotunk, ahol a megoldásnak 0 az amplitudója, ezáltal merőleges a megoldás állapotára az állapotvektora, az az állapot lesz, ahol csak a kiindulási állapotunknak pozitív a fázisa, többinek negatív, azaz:

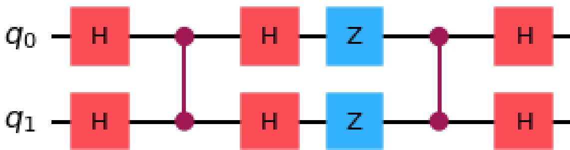
$$\mathrm{U}_{00}\,|s\rangle \;=\; \mathrm{U}_{00}\frac{\mathbf{1}}{\mathbf{2}}\left(|\mathbf{00}\rangle + |\mathbf{01}\rangle + |\mathbf{10}\rangle + |\mathbf{11}\rangle\right) \;=\; \frac{\mathbf{1}}{\mathbf{2}}\left(|\mathbf{00}\rangle - |\mathbf{01}\rangle - |\mathbf{10}\rangle - |\mathbf{11}\rangle\right)$$

Ez már nem annyira egyértelmű, mint az oracle-ünk, de kiszámolható könnyedén, hogy a kérdéses operátor itt $\mathrm{U}_{00} = (\mathrm{CZ})\,(\mathrm{Z}\otimes\mathrm{Z})$. No persze nekünk nem egészen ez kell, mert itt már eleve egyenlő szuperpozícióból indultunk. Tehát kell az operátor elé egy Hadamard transzformáció. És kell az operátor mögé is, "visszaszámítva" az előtte lévőt, tehát az diffuser operátorunk U_s

$$\mathrm{U}_s \;=\; \mathrm{H}^{\otimes 2}\mathrm{U}_{00}\mathrm{H}^{\otimes 2} \;=\; \mathrm{H}^{\otimes 2}(\mathrm{CZ})\mathrm{Z}^{\otimes 2}\mathrm{H}^{\otimes 2}$$

Lássuk tehát a kódját, bővítve az előzőt, és kiegészítve a grover_iter functiont is:

```
# 2. lépés: az Oracle
def oracle() -> QuantumCircuit:
    oracle = QuantumCircuit(2)
    oracle.cz(0,1)
    return oracle

# 3. lépés: Diffuser
def diffuser() -> QuantumCircuit:
    diffuser = QuantumCircuit(2)
    diffuser.h([0,1])
    diffuser.z([0,1])
    diffuser.cz(0,1)
    diffuser.h([0,1])
    return diffuser

# Készítjük a grover iterációnkat
def grover_iter(qc) -> QuantumCircuit:
    qc.compose(oracle(), [0,1], inplace=True)
    qc.compose(diffuser(), [0,1], inplace=True)
    return qc

# Plot a mostani helyzettel
qc = grover_iter(qc)
qc.draw('mpl', style='iqx')
```



Note, algebrailag az operátorok sorrendje egy állapoton jobbról balra történik, egy áramkör modellen pedig azt tesszük fel elsőnek, amit elsőnek hajtunk végre, tehát megfordul a sorrend. Itt most 1 iteráció elég lesz.

A tükrözés algebrailag úgy működik, hogy az egyenlő szuperpozícióból kiindulva minden állapot felírható

$$|\psi\rangle = sin(\theta_k)\,|\omega\rangle + cos(\theta_k)\,|s'\rangle \ , \quad \theta_1 = arcsin(\sqrt{t/N}) \text{ alakban.}$$

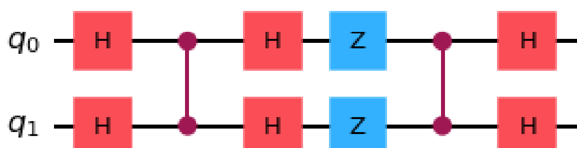
Emellett

$$\theta_{k+1} = (2k-1)\theta_k$$

Mi azt szeretnénk, hogy a $|w\rangle$ állapotot mérjük, tehát azt, hogy $sin(\theta_k) = 1$, tehát azt, hogy $\theta = \frac{\pi}{2}$ legyen. Indulásnál $\theta_1 = arcsin\left(\sqrt{\frac{1}{4}}\right) = arcsin\left(\frac{1}{2}\right) = \frac{\pi}{6}$. Ebből, és a képletbe behelyettesítve megkapjuk, hogy $\theta_2 = (4-1)\,\theta_1 = \frac{3\pi}{6} = \frac{\pi}{2}$.

A kódunk pedig, az iterációkkal együtt végül így fog kinézni. Megkaptuk, hogy itt az iterációszámunk 1 lesz, tehát:

```
# 2. lépés: az Oracle
def oracle() -> QuantumCircuit:
    oracle = QuantumCircuit(2)
    oracle.cz(0,1)
    return oracle

# 3. lépés: Diffuser
def diffuser() -> QuantumCircuit:
    diffuser = QuantumCircuit(2)
    diffuser.h([0,1])
    diffuser.z([0,1])
    diffuser.cz(0,1)
    diffuser.h([0,1])
    return diffuser

# Készítjük a grover iterációnkat
def grover_iter(qc) -> QuantumCircuit:
    qc.compose(oracle(), [0,1], inplace=True)
    qc.compose(diffuser(), [0,1], inplace=True)
    return qc

iterations = 1
for i in range(iterations):
    qc = grover_iter(qc)

# Plot a mostani helyzettel
qc.draw('mpl', style='iqx')
```



Három qubites megoldásnak, és kicsit részletesebb matematikai leírásnak akit érdekel, utána tud nézni a fent linkelt Qiskit textbook fejezetben.

11.2. Fázis becslés

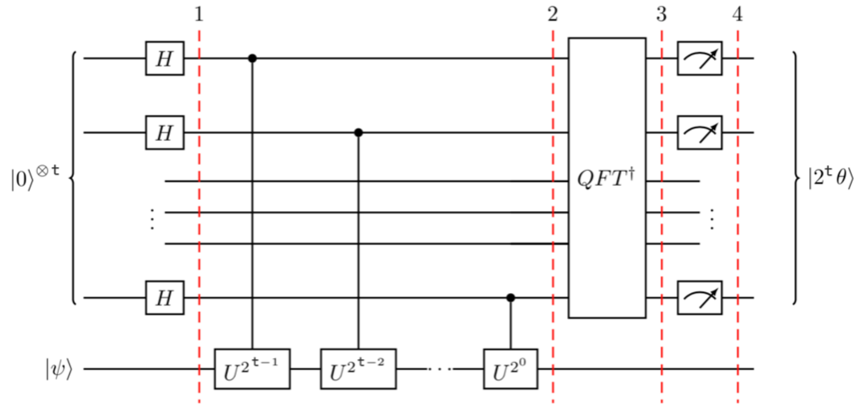
Volt szó arról, hogy nem csak a qubitek alap bináris reprezentációjával tudunk számítani, hanem a fázissal is. Ezt viszont meg kellene mérnünk valahogy. Ismét valami folytonossal van dolgunk, mint a szuperpozíció mérésénél. De míg a szuperpozíció összeomlik az egyik állapotra, mi szeretnénk itt n biten ábrázolni a fázist.

Valami ilyesmit már láttunk valahol! A Fourier transzformációnál pont valami ilyesmit csináltunk, de ott tudtuk milyen állapotból milyen fázist szeretnénk előállítani. Most nem tudjuk milyen fázisunk van, amit ábrázolni szeretnénk.

A példán egy pontos példát fogunk látni, ahol 4 qubiten az 11/16 fázist pontosan tudjuk ábrázolni. De pl. 3 qubiten megpróbálhatnánk a 4/7 fázist is ábrázolni, olyankor 4/8 és 5/8 közötti (nem egyenlő) szuperpozíciót fogunk kapni. Persze, mert 3 qubiten nyolcadfázisokat tudunk csak megjeleníteni.

Az algoritmus működése viszonylag egyszerű. Ha van egy operátorunk U, ami valami m qubites $|\psi\rangle$ áramkörre hat, és valamilyen algoritmust valósít meg, amely a végén ad egy θ fázist a regiszter valamilyen bázisállapotának[3]. És mi ezt szeretnénk megmérni és ábrázolni.

Ehhez pedig egy másik regiszterben - amiben előállítjuk az eredményt és majd mérni fogjuk - egyenlő szuperpozícióba tesszük a qubiteket, majd, pont mint a QFT áramkörénél, végrehajtjuk az $n.$ qubiten a C(U$^{2^n}$) kaput. Ekkor az $n.$ qubiten megjelenik egy $2^n\theta$ fázis. Ezután QFT^{-1} az eredményen, mérünk, reménykedünk, kinyitjuk a szemünket, és



[3]Note: ez relatív fázis, nem az egész regiszter kapja, hanem csak a szuperpozíció bizonyos bázisállapota

Tovább vizsgáljuk az áramkört. Tehát lépésekben:

1. A cél regiszterünket előkészítjük egyenlő szuperpozícióba, a munkaregiszterünket pedig abba a bázisállapotba (U egyik sajátállapotába), amelyik a fázist kapja.
2. Végrehajtjuk a C(U^{2^n}) kapukat
3. QFT^{-1} a cél regiszteren
4. Mérés a cél regiszteren

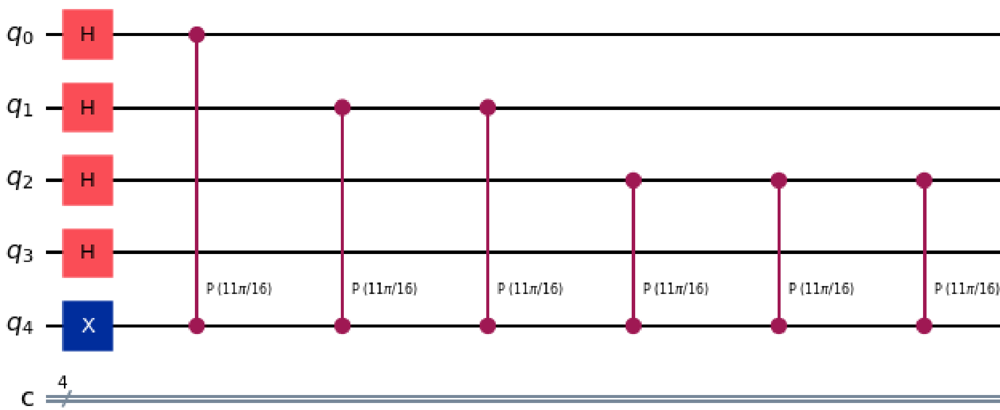
11.2.1. Implementáció

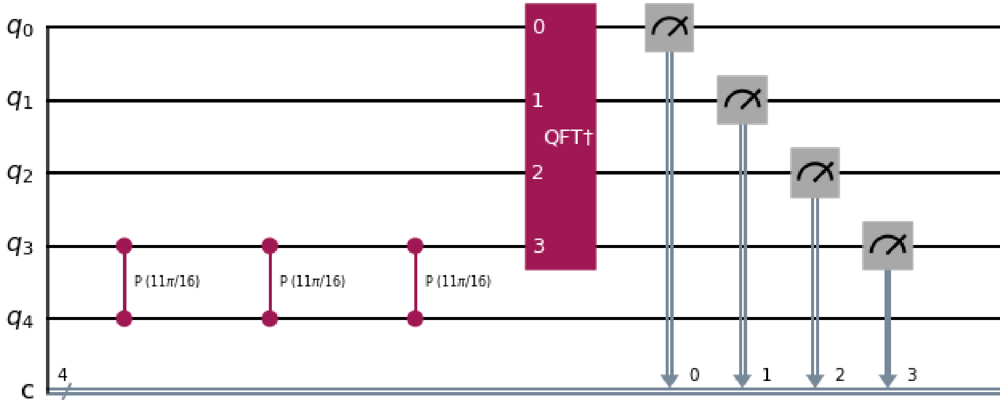
Most nézzük meg a példát. Tegyük fel, hogy van valami áramkörünk, ami 11/16 π fázist ad az áramkörünknek. Elsőre lehet nem logikus miért tenné ezt, de például lehetne a 4 bites áramkörünk megoldása (outputja) is 11 (azaz 1101), majd egy QFT-vel ebből pont ezt a fázist tudjuk előállítani. Egy bonyolultabb algoritmusnál nyilván nem tudjuk előre, ránézésre az inputból megmondani az outputot, ugyanígy itt se tudjuk a fázist - vagy tudjuk, de nem kell nekünk pontosan, csak 2^{-n} pontossággal.

Lássuk tehát a kódot:

```
n=4 # cél bitek
m=1 # munka bitek

qc = QuantumCircuit(n+m, n)

qc.h(range(n)) # cél
qc.x((n+m)-1)  # U megfelelő sajátállapota
               # itt ez most az |1>

# CU^(2^n) implementációja
for i in range(n):
    for j in range(2**i):
        # U = P(11/16 * pi) itt,
        #     1 qubites áramkör
        qc.cp(11*pi/16, i, (n+m)-1)

# Inverz QFT, importálva.
qftd = QFT(num_qubits=4, inverse=True, name='QFT†')
# és ráteve az áramkörünkre
qc.compose(qftd, [0,1,2,3], inplace=True)

# mérés
qc.measure([0,1,2,3], [0,1,2,3])
```

Az áramkör pedig így néz ki:





Az áramkör egy része nem fért rá a képre, a CU kapuk természetesen folytatódnak még, illetve a második sorban fejeződnek be. Az áramkör mérés után a 1101 állapotot adja, ebben az esetben 100% eséllyel elméletben, ha zajtalan környezetben futtatnánk.

11.2.2. Kitekintés, felhasználás

Egy szuperpozícióban az egyes állapotok amplitudóját nem tudjuk megmérni, ez összeomlik méréskor. Viszont át tudjuk konvertálni fázisokra, majd megbecsülni a fázist. Például ha van egy SAT problémánk[4] 3 literállal, 8 lehetséges értékadással, amiből 2 kielégíthető, akkor írni tudunk egy kvantum programot, ami 1 qubitre visszavezeti az egész problémát,

[4] SAT a formula kielégíthetőség. Döntési probléma, bináris megoldással: "kielégíthető-e a formula?"

és a qubit 2/8 eséllyel 1 lesz, egyébként 0. Mi nem tudhatjuk hogy pontosan 2 kielégítő értékadása létezik, és, bár sokszoros méréssel közelíthető, ebből a szuperpozícióból sem tudjuk egyértelműen meghatározni.

Viszont 3 literálból tudjuk hogy 8 lehetséges értékadása van, amit 3 biten tudunk ábrázolni, így például a phase estimation segítségével meg tudjuk mondani az előző 1 output bitünkből, hogy pontosan mennyi kielégítő értékadásunk van.

Ez a példa szemléltetésnek jó, de nem realisztikus, mert egy SAT problémánál nem jellemző, hogy szükségünk van a kielégítő értékadások számára, csak hogy ez a szám nemnulla-e, azaz a formula kielégíthető-e. Másrészt ez egy NP-teljes probléma, és az input méretében annyival bonyolultabb lesz a feladat, hogy az algoritmus elveszíti a hatékonyságát. Legalábbis nem hisszük, hogy hatékonyan megoldható lesz bármikor.

Shor faktorizáló algoritmusa

12.1. Motiváció

Az összetett számok prímtényezőkre bontása érdekes és nehéz feladat. Maga a probléma nagyon egyszerű, és ezért elég hamar feltűnt hogy algoritmikusan nem olyan könnyű megoldani. Később még központibb problémává vált, miután az RSA nyilvános kulcsú titkosítási protokoll elterjedt, ami pont eköré a probléma köré épül fel.

Megvizsgálni egy számról, hogy prím-e, viszont könnyű feladat, polilogaritmikus időben eldönthető. Az RSA-nak ez az alapja: könnyű nagy prímeket találni, könnyű őket összeszorozni, könnyű a szorzatukat elosztani egyikükkel. De a két prím ismeretének hiányában NP-beli problémát alkot, tehát klasszikusan nehezet. A legjobb klasszikus algoritmusok is egy N (decimális) számra $2^{(logN)^{\alpha}}$ időben futnak, ahol α egy konstans, ami 1/2-nél mindenképp kisebb.

12.1.1. Bonyolultságelmélet, törióra

Bár a faktorizálás problémája nem NP-teljes, és ezért egy hatékony algoritmus erre a problémára nem bizonyítaná, hogy P=NP egyenesen, de mást igen. Ehhez be kell vezetnünk két bonyolultságelméletről nem ismerős bonyolultsági osztályt, a BPP-t (Bounded-Error Probabilistic Polynomial time) és BQP (Bounded-Error Quantum Polynomial time) osztályát. Az angol nevük beszédes, az ide tartozó algoritmusok pontosan kiszámítható hibával futnak, polinomidőben. Például olyan, mintha a 3*5-ről szeretnénk megmondani, mennyi, és az esetek (pontosan) 60%-ában megkapjuk, hogy 15, a többiben pedig valami mást.

Ez a két osztály klasszikus-kvantum párt alkot, teljesen ugyanaz a definíciójuk. Tudjuk, hogy BPP $\subseteq$ BQP, és azt is, hogy FAKTORIZÁLÁS $\in$ BQP. De nem tudjuk, hogy BPP-ben benne van-e. Ha igen, akkor lenne rá egy hatékony probabilisztikus klasszikus algoritmus, ami ilyen formán feltörné az RSA-t. Ilyet nem sikerült még találni, és úgy gondoljuk, nem is lehet. Ugyanakkor, ha valaki *bebizonyítja*, az ellenpéldát mutatna az erős Church-Turing tézisre, ami kimondja, hogy minden valós számítási modell polinomiálisan ek-

vivalens. Erre is van törekvés, de ez is egy nagyon erős állítás, csakúgy mintha valaki találna egy BPP algoritmust a problémára.

Magyarul ez az algoritmus erős sejtés ad arra, hogy a kvantumszámítás nem ekvivalens számítási modell a klasszikussal, és határozottan többre képes egyes problémákon. A 90'-es évek közepén ez óriási szenzációt keltett, és felkeltette sokak figyelmét, a kvantumszámítás pedig intenzív fejlődésnek indult, ami azóta is tart. Ugyanakkor azóta sem talált senki szuperpolinomiális vagy annál nagyobb gyorsítást adó algoritmusra példát, ami nem Shor algoritmusára épül.

12.2. Algoritmus leírása

Shor algoritmusa jó faktorizálásra, de egy annál általánosabb problémát old meg, ez pedig a rejtett részcsoport probléma. Erről a problémáról lesz szó a fejezet végén.

A problémáknak van egy közös struktúrális tulajdonsága: a megoldásaik periodikusak. Hogyan lehet egy faktorizálást periódus-keresésre visszavezetni? Mondjuk emlékezzünk vissza, általános iskola alsó tagozatban hogyan tanulunk osztani. Bocsánat, bennfoglalni. "15-ben meg van az 5?" - 5, 10, 15. Igen, meg! Hányszor? 5 (1), 10 (2), 15(3). Háromszor! Tehát a 15-nek van egy 5 hosszú periódusa, ami háromszor ismétlődik benne.

Az alap ötlet, dimat3-ról, és a véges testekről ismerős lehet. Ha van egy véges test, pl. Z_{13}, akkor ha veszünk egy számot belőle, mondjuk az 5-öt, akkor az 5 hatványai (mod 13) ki fogják adni az összes elemet, azaz az összes számot 0 és 12 között.

Na ez kicsit árnyalódik. Test helyett multiplikatív csoportunk van ($\mathbb{Z}_N^*$), de ez senkit ne ijesszen meg. Parametizáljuk a feladatot, nézzük általánosabban. Az előző példából a 13, a szám amit faktorizálni szeretnénk, legyen N. A példában az 5, amit hatványozunk, legyen x. Így nézzük a következő sorozatot:

$$x^0 \pmod N, \quad x^1 \pmod N, \quad x^2 \pmod N, \ldots$$

aminek az első eleme triviálisan 1. Ez a sorozat egy idő után ciklizálni fog, azaz tudunk találni egy $0 < r \leq N$ számot, amivel $x^r = 1 (\text{mod } N)$. Ez az r lesz a sorozat periódusa (és a csoport rendje is).

Az algoritmus szempontjából vizsgálva, nekünk azok az esetek kellenek, ahol N páratlan és nem prím, különben a feladatnak könnyű gyors megoldása van. Mindkét

feltételt könnyen meg lehet vizsgálni klasszikusan. Ekkor bizonyítható, hogy valami 1/2-nél nagyobb eséllyel a periódus r páros és $x^{r/2}+1$ vagy $x^{r/2}+1$ nem többszöröse N-nek. Ekkor

$$\begin{aligned} x^r &\equiv 1 \pmod N \iff \\ \left(x^{r/2}\right)^2 &\equiv 1 \pmod N \iff \\ \left(x^{r/2}+1\right)\left(x^{r/2}-1\right) &\equiv 0 \pmod N \iff \\ \left(x^{r/2}+1\right)\left(x^{r/2}-1\right) &\equiv \mathrm{k}N \text{ valamilyen k-ra.} \end{aligned}$$

A szorzat egyik tagja sem nulla, ezért k>0. Ezért $x^{r/2}+1$-nek vagy $x^{r/2}-1$-nek vagy N-nel közös osztója. És mivel egyik szám se többszöröse N-nek, ezért ez az osztó kisebb is lesz N-nél. Innen egyszerűen euklideszi algoritmussal kiszámolható a legnagyobb közös osztó. Persze ez lehet nem fog megoldást adni, de valami 1/2-nél nagyobb eséllyel igen. Ha nem ad megoldást, ismételhetjük az algoritmust addig, amíg ad.

Az algoritmus kvantum része r megtalálására irányul. Utána klasszikus módon meg tudjuk találni a kérdéses osztót. A perióduskeresésre a legjobb ismert klasszikus (BPP) algoritmusnak $\Omega(N^{1/3}\sqrt{\log N})$ időre van szüksége. Shor algoritmusának egy kicsit gyorsabb, mint négyzetes futásideje van.

Az algoritmus matematikai leírása bonyolultabb a kurzus szintjénél, ezért akit érdekel, innen jó forrást talál hozzá a 37. oldaltól. Illetve hasznos lehet hozzá Simon algoritmusának az ismerete a 21. oldaltól.

12.3. Implementáció, példa

A Qiskit textbook példáján fogunk végighaladni. A 15-öt fogjuk felbontani. Van pár bottleneck az algoritmusban, ami kellemetlen hosszú tud lenni kézzel összerakva. Ez a bottleneck mindenhol szembejön: a moduláris hatványozás, és ennek az implementációja.

Az algoritmus alapjába véve ugyanúgy működik, mint a Phase Estimation az előző fejezetből, szteroidokon: az U operátor belőle a moduláris hatványozást végrehajtó áramkör. Ahogy láthattuk az előző fejezetben, az n. qubiten a Controlled-(U^{2^n}) operációt kellene végrehajtani. Ezzel két probléma van. Az egyik, hogyha valóban az exponenciálisan sok kaput akarjuk felpakolni, az nagyon hatékonytalan. A másik, hogy ha áramköri azonosságok mentén le akarjuk bontani, hogy ne legyen exponenciálisan sok kapu (ami lehetséges,

és szükséges is), akkor minden U^{2^n}-ra különböző áramkört kell építeni, ami kellően nagy n esetén nem túl hálás feladat.

Lássuk az implementációt. Először kelleni fog nekünk egy pár lib, most ezeket is szükséges megemlíteni:

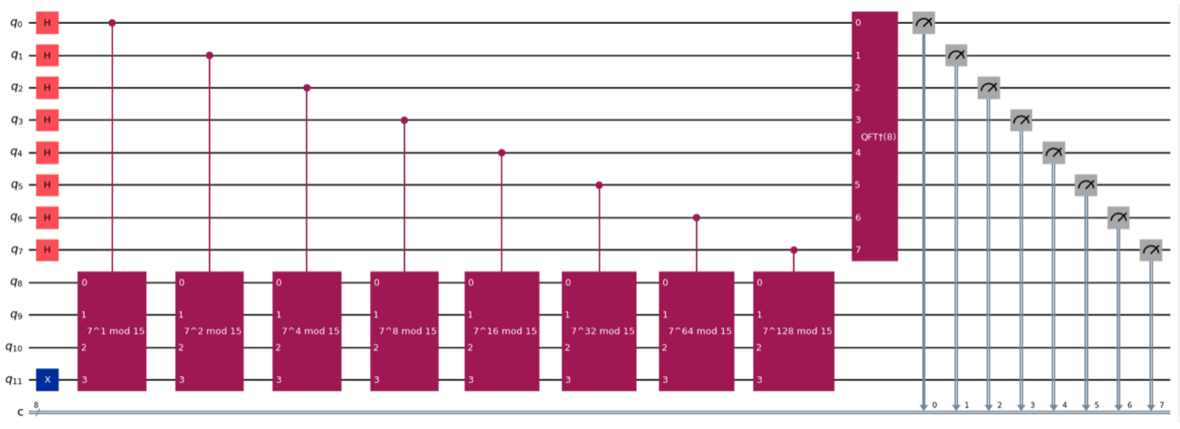
```
import numpy as np

# Importing standard Qiskit libraries
from qiskit import QuantumCircuit, transpile, assemble, Aer, IBMQ
from qiskit.circuit.library import QFT
from qiskit.tools.jupyter import *
from qiskit.visualization import *
from ibm_quantum_widgets import *
from qiskit.providers.aer import QasmSimulator
from fractions import Fraction
from math import gcd

# Loading your IBM Quantum account(s)
# provider = IBMQ.load_account()
```

Tehát először meg kell építenünk U-t, úgy, hogy $U|y\rangle = |\alpha y \bmod 15\rangle$. Ehhez α értékét 7-re állítjuk most. Ez lesz a részcsoportunk generálóeleme, amely segítségével megkeressük a periódust. A 15-öt bontjuk fel, ez 4 biten elfér. A fázist 8 biten fogjuk becsülni.

```
N = 15 # 15 lesz amit felbontunk
n = 8 # 8 bites fázisbecslésses
a = 7 # 7 lesz a generálóelemünk

def c_amod15(a, p):
    U = QuantumCircuit(4) # ceil(log_2 15)

    # 7^p mod 15
    for iteration in range(p):
        U.swap(2,3)
        U.swap(1,2)
        U.swap(0,1)
        for q in range(4):
            U.x(q)
    U = U.to_gate()
    U.name = f"{a}^{p} mod {N}"
    c_U = U.control()
    return c_U
```

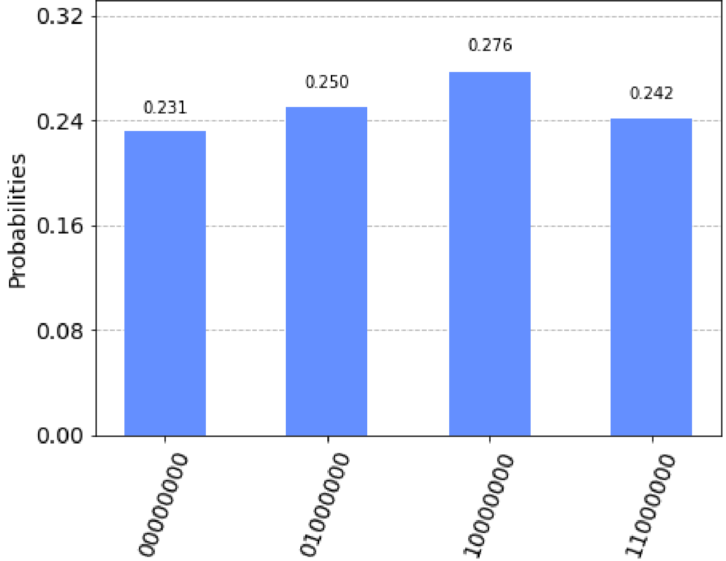
Most pedig rakjuk össze az algoritmust. A lépéseket a kommentekben olvashatjuk.

```
qc = QuantumCircuit(n + 4, n) # 8 számoló, 4 munka qubit

qc.h(range(n)) # H trafó a számoló qubiteken
qc.x(n+3) # phase query előkészítése

for q in range(n):
    # hozzáfűzzük az eddigiekhez a C^2^q mod hatványt
    # control qubit(ek): q
    # target qubit(ek): munka regiszter, azaz n+ 0,1,2,3
    qc.append(c_amod15(a, 2**q), [q] + [i+n for i in range(4)])

# QFT^-1 a számoló regiszterre
qft_dg = QFT(num_qubits=n, inverse=True, name=f'QFT†({n})')
qc.append(qft_dg, range(n))

# Számoló regiszter mérése
qc.measure(range(n), range(n))
qc.draw(style='iqx', fold=-1)
```



Ezzel nem vagyunk kész még, csak kaptunk egy eredményt valamilyen s/r értékre. De először nézzük meg, ha sokszor futtatnánk ezt az áramkört, mit kapnánk:

```
aer_sim = Aer.get_backend('aer_simulator')

# A saját kapuinkat le kell bontania
t_qc = transpile(qc, aer_sim)
qobj = assemble(t_qc)

results = aer_sim.run(qobj).result()
counts = results.get_counts()
plot_histogram(counts)
```



Azaz 4 lehetséges kimenetel van, lehet sejteni, hogy r 4 lesz. De valójában amikor futtatjuk az algoritmust, csak az egyiket kapjuk meg. Például megkaphatjuk a 01000000 bitstringet, ami a 64/256 = 1/4 fázisnak felel meg, ami tök jó, mert egyből kiderül, hogy r=4. De megkaphatjuk pl. azt is, hogy 1/2. De ilyenkor ebből is tudunk számolni. Nem mindig, de több mint 50% eséllyel igen. Tehát tegyük fel, hogy egyszer futtattunk, kaptunk egy eredményt, és az eredmény amit kaptunk:

```
measured_phases = []
for output in counts:
    decimal = int(output, 2)  # 10-es számrendszerre konvert.
    phase = decimal/(2**n)  # a fázis ennek a 2^n-ed része
    measured_phases.append(phase)

# random elem, legyen 1
phase = measured_phases[1]
```

Nézzük a fázisunknak mi a számlálója, milyen r-t kaptunk:

```
# a számláló remélhetőleg r lesz pont
frac = Fraction(phase).limit_denominator(15)
s, r = frac.numerator, frac.denominator
r
```

```
2
```

És a fentiek alapján nézzük, milyen megoldásokat kaptunk ezzel a fázissal:

```
# keressük valamelyik megoldást
guesses = [gcd(a**(r//2)-1, N), gcd(a**(r//2)+1, N)]
guesses
```

```
[3, 1]
```

És 3 egy jó megoldás pont (1 is, de az triviális). A Fraction-ra azért volt szükség, mert ezek a számok nem feltétlenül jönnek ki ilyen szép egészre, és olyankor lánctörtekkel kell megkeresni a legközelebbi "szép" számot. Ezt megteszi helyettünk a python.

12.4. Alkalmazások

Ennek az algoritmusnak való életbeli felhasználási lehetőségei is vannak. Fontos kiemelni, hogy ennek vannak, mert nagyon kevés kvantum algoritmusnak van praktikusan realizálható mértékű előnye. Jellemzően a kvantum algoritmusok polinomiális gyorsulást tudnak felmutatni, ugyanakkor van egy nagyjából ezerszeres hátrányuk műveleti gyorsaságban. Ezt persze az ordó elrejti, tehát gyakorlatban például egy négyzetes előny úgy néz ki, hogy a klasszikus algoritmus fut n^4 időben, a kvantum algoritmus pedig $1000n^3$ időben. Például Grover algoritmusa $O(n)$ helyett $O(\sqrt{n})$ időben fut. Valóságban n helyett $1000\sqrt{n}$. Kis számolással meg is láthatjuk, hogy ilyenkor, hogy a kvantum algoritmus valóban gyorsabb legyen, n-nek nagyon nagynak kell lennie[1]. Egyrészt teljesen egyértelmű, hogy ez nem praktikus. Másrészt persze azt is érdemes megemlíteni,

[1] $n > 1000000$ kell teljesüljön

hogy egy nagyon gyorsan fejlődő iparágról van szó, és főleg a kvantumhardveres dolgok folyamatosan, iszonyatos tempóban javulnak. Lehet mire ennek a mondatnak vége lesz, már csak 100 lesz az az 1000-szeres overhead.

Gyors fejlődés ide vagy oda, *jelenleg* nincs felhasználási területe nagyon sok kvantum algoritmusnak, mégha matematikailag jobbak is. Egyedül Shor algoritmusa ad akkora (jelen esetben szuperpolinomiális) előnyt, hogy praktikus felhasználási lehetőségei legyenek a jelenleg ismert klasszikus algoritmusok felett.

12.4.1. Kriptográfiai jelentősége

Az RSA titkosításról és pontos megvalósításáról számítógép hálózatokon és információbiztonságon is tanulhattatok. Ahogy arról már volt szó, a pontos eljárás a prímvizsgálat és a szorzás könnyűségén, és a faktorizálás nehézségén alapul. Most már a csapból is az folyik, hogy ez az algoritmus feltöri az RSA-t és úristen, apokalipszis. Dehát ez csak egy titkosítási protokoll! Mégis mi a jelentősége akkor ennek a kvantum algoritmusnak ezen túl?

Minden nyilvános kulcsú titkosító protokollnak az a célja, hogy egy könnyen kiszámítható módon generáljon kulcsokat, amelyeket utána visszafejteni nehéz. Ilyen tökéletesen "one-way-function" nem létezik, ha P$\neq$NP.

A problémakör nagyon érdekes. Tegyük fel magunknak a kérdést: hogyan alkotnánk olyan titkosítási protokollt, aminek a kulcsait kiszámítani könnyű (maximum P-teljes probléma), de visszafejteni nehéz (legalább NP-teljes)?

12.4.2. NP-köztes problémák

És el is jutottunk a bonyolultságelmélet egyik ingoványos részére. Tehát vannak P-teljes problémáink, és vannak NP-teljesek. És van a faktorizálás, amelyik egyik sem. NP-beli probléma, de nem NP-teljes, ugyanis nem vezethető vissza rá minden NP-teljes probléma.

A faktorizálás az NP-köztes problémák osztályába tartozik. És nagyon sokáig minden nyilvános kulcsú titkosítási protokoll alapján NP-köztes problémák képezték. Ugyanakkor ezekre a problémákra időnként felbukkannak klasszikusan polinomidejű determinisztikus

algoritmusok. Ezért annyira nem életbiztosítás NP-köztes problémára feltenni minden pénzünket például. Ezen kívül persze az sem biztos, hogy P≠NP, de ez egy jóval erősebb állítás[2], sejtés, mint hogy 1-1 NPI (NP-Intermediate) problémára nem létezik klasszikus algoritmus.

Maga a probléma amivel szembesülünk, egészen könnyen felfogható, bárki számára jó gondolkodási lehetőséget ad, és mégis nagyon bonyolult. Hogy alkossunk olyan titkosítást, ami nem NP-köztes problémán alapul? Az teljesen biztos innen, hogy NP-teljes problémát kell alapul vennünk, mert muszáj 1-1 példának könnyen kiszámíthatónak lennie, az egész problémának pedig nehéznek lennie.

Létezik ilyen protokoll, zajos mátrixok szorzásán alapul, és leginkább az egészértékű programozás problémájához lehetne hasonlítani: egy (nem lineáris) térben rácspontjaink vannak, amiket elzajosítunk valamilyen módszerrel. Így megtalálni egy ponthoz legközelebbi rácspontot a hipertérben, a zajosító transzformáció ismerete nélkül olyan, mintha branch and bound módszerrel szeretnék ILP problémát megoldani. Ez a módszer egy egész protokollcsalád alapját képezi, és nagy valószínűséggel ilyen lesz az első szabványosított poszt-kvantum titkosítási protokoll is.

12.4.3. Rejtett részcsoport probléma

Shor algoritmusa által megoldott, legáltalánosabb problémáról van szó. A jegyzet nem lesz explicit csoportelméleti fogalmakkal, ezért akinek nem tiszta, ezen a linken találja a diszkrét matematika 2-es absztrakt algebrai anyagot. Egy rövid intuitív összefoglalót azért megnézünk, hogy tisztuljon a kép, de bizonyos műveleti tulajdonságok, és fogalmak nélkül nem lesz sok értelme a következő résznek.

Röviden: a csoport egy olyan matematikai struktúra, ami lényegében egy szimbólumhalmaz, amelyeken értelmezett egy kétváltozós művelet, és ez a művelet kielégít pár tulajdonságot a csoport elemein. Emellett a művelet nem is "vezet ki" a csoportból, tehát például az egyjegyű számokon megszokott módon nem lehet az összeadást mint műveletet megjelölni, mert 5+8 például nem egyjegyű.

Pár tulajdonságnak teljesülnie kell, ami egyfajta jól struktúráltságot ad a csoportnak.

[2]Ez matematikailag nem pontos állítás, de érthető. Azt jelenti, hogy ha P=NP, akkor minden NP-köztes probléma is P-ben van (Ladner, 75')

Tehát a tulajdonságok, amelyeknek teljesülniük kell:

1. asszociativitás (ez a csoportosíthatóság),
2. van benne egységelem, azaz van benne egy elem, amelyre a művelet úgy hat, bármilyen számmal, hogy a másik számot kapjuk (szorzásnál ez pl. az 1, mert 1-szer bármi az bármi, 1×5=5),
3. minden elemnek van inverze. Azaz minden elemnek van egy "párja", amivel ha összeműveljük, az egységelemet fogjuk kapni. Például az összeadás művelet esetén az egységelem a 0. És bármilyen számnak az inverze az ellentettje (pl. 5 additív inverze a -5).

Hogy visszatérjünk a faktorizálás problémájához: vehetjük például a $\mathbb{Z}_7^*$ multiplikatív csoportot. Azaz 7-hez relatív prím egész számok, mindennek a 7-tel vett maradéka. A művelet pedig a szokásos szorzás. Na most, a művelet ismétlése ugyanazzal az elemmel érdekes dolgokra képes. Tehát például ha kiválasztjuk az 5-öt, és elkezdjük összeművelni önmagával[3], akkor

$$\mathbf{5^1 = 5, \quad 5^2 = 4, \quad 5^3 = 6, \quad 5^4 = 2, \quad 5^5 = 3, \quad 5^6 = 1 \ldots}$$

Minden ugye 7-tel vett maradékkal, tehát $\mathbf{5^2 = 25}$ mod $\mathbf{7 = 4}$. Észrevehetjük, hogy 0-t sosem kaptunk, illetve ezen kívül minden egyéb elemet megkaptunk. Ekkor azt mondhatjuk, hogy 5 *rendje* 6. Ez pont 1-gyel kevesebb a csoport elemeinek számánál.

Most nézzünk egy másik példát. Legyen a csoportunk $\mathbb{Z}_{15}^*$, az elemünk legyen a 4. A 4 második hatványa 16 (mod 15) = 1, azaz a 4 periódusa 2, kételemű részcsoportot generál, és ennek az elemei a 4 és az 1. Shor algoritmusa ilyenkor a $2^{\frac{4}{2}} - 1$ vagy $2^{\frac{4}{2}} + 1$ és N lnko-jára mondja, hogy osztója lesz a 15-nek, és valóban az is, éppen mindkettő.

Ha nagyon sok eleme van a csoportunknak, tegyük fel $\boldsymbol{N}$, akkor már egyáltalán nem ennyire egyértelmű találni egy többelemű részcsoportot - egy rejtett részcsoportot, tehát megtalálni $\boldsymbol{N}$ egy osztóját.

Persze, ezek a számelméleti csoportok elég speciálisak, csoportot alkothat nagyon sok minden, például bizonyos sokszögek egyes geometriai transzformációkkal, stb.

És minden ilyesmi struktúrájú csoportnak kereshetjük rejtett részcsoportját Shor algoritmusának segítségével.

[3] vedd észre, ez ismételt szorzás, pont hatványozás!