

Course Notes

Ravi Budhrani

January 2025

Contents

1	Engineering and developing software	5
1.1	Software development lifecycle	6
1.1.1	SDLC models.	7
1.2	Principles of good software	7
1.3	Version control.	8
1.3.1	Git.	8
2	Python	13
2.1	Installing Python	13
2.2	Hello world	14
2.3	Variable and data structures	14
2.3.1	Numbers	15
2.3.2	Strings.	15
2.3.3	Lists	16
2.3.4	Dictionaries	17
2.4	Conditions and loops.	17
2.4.1	if condition	17
2.4.2	for loop	19
2.4.3	while loop.	19
2.5	Functions.	20
2.6	Packages	21
2.6.1	pip package manager	21
2.6.2	Virtual environments	21
3	Quantum computing	25
3.1	Motivation	25
3.2	Quantum acceleration	26
3.3	Ecosystem	26
4	Quantum information	29
4.1	The classical bit	29
4.2	The quantum bit	30
4.2.1	Superposition	30
4.2.2	Measurement	31
4.2.3	Entanglement	32
5	Manipulating qubits	35
5.1	The logic gate	35
5.2	The quantum gate	37
5.2.1	Braket notation	41
5.2.2	Global phase	42

6	Quantum circuits	45
6.1	Entanglement generation	45
6.2	Grover's Algorithm	46
6.2.1	The Boolean oracle	47
6.2.2	The Phase Oracle	49

1

Engineering and developing software

The titles software engineer and software developer are used interchangeably. However, there are differences between the 2, just as there are similarities. It is also not uncommon to see one person perform tasks related to both development and engineering. Table 1.1 gives a break-

Software engineering	Software development
Tasks span the entire software development lifecycle, all the way from design to deployment	Tasks are mostly about implementation of the software
Requires interaction with different stakeholders to help design the software	Requires interaction with other software developers or engineers

Table 1.1: Software engineering versus development

down of the work involved with software engineering and development. These differences result in different types of problems that need to be solved by each discipline. Software development problems are usually more focused on the implementation of the solution. The problems can be more technical as they require a strong grasp of the language. Software engineering problems can span the entire development lifecycle. Any problem that is not about the technical implementation is usually in the field of software engineering. Having said that, even if the problem is technical it does fall in the domain of software engineering. Software development can be considered a part of software engineering. A software engineer

can work on software implementation but usually has other tasks on their plate.

1.1. Software development lifecycle

The software development lifecycle (SDLC) is a process used to create software. The steps in this cycle are:

1. **Planning and analysis:** In this step the goals, scope and requirements of the software project are determined. This is done by talking to all the stakeholders involved in the project, since every stakeholder will bring different competencies to the project. Each stakeholder will have a different requirement, that is specific to their discipline. The software engineers must address all these requirements to create the final scope and goals of the project. Market research is another important task for this step. The feasibility and risks of the project need to be evaluated to have a good picture on the planning of the project. Budget and time considerations are very important in evaluating if a project is feasible.
2. **Design:** In this step the software engineers design the architecture of the project, analogous to architects coming up with the blueprint of a building before it is built. This step is highly dependent on the project itself. Decision on database design and high level interfaces are made here. If this project is part of a larger piece of software, then the integration design is also done in this step. Sometimes this step also entails creating a prototype, to get early insight and feedback for the project.
3. **Development:** With the decisions made earlier, the developers can start to implement the software. If the choice of language(s) is not made in the design step, it is done here. The development step has the most information about choosing the language, unless this project is about developing a software component that is a part of a larger software. In that case, the choice of language is made before this step. In the development phase the developers also test the code programmatically as much as possible, before passing it to the testers. After this step, the software should be fully functioning.
4. **Testing:** In this step testers rigorously test the software. The tests include functional tests, to check that the software does what it was designed to do and it meets the requirements that were set out in the beginning of the cycle. The tests can also include cyber security tests, if that was a requirement of the software. If the software is part of a larger piece of software, then integration tests are also performed, to ensure that the whole system is bug free.

5. **Deployment:** Once there are no bugs in the software, it is deployed/released to the public/client.
6. **Maintenance:** Even though the software is now being used by the public, there are chances that some bugs slipped through. In this step software engineers and developers continue to fix these bugs and even add improvements based on user feedback.

1.1.1. SDLC models

There are several approaches to the SDLC. 2 common models are:

- **Agile:** This is an iterative model. The entire project is split into small increments. Each increments usually lasts a couple of weeks or slightly more. These increments called sprints. This model is flexible to change. As the team grows increments can be performed in parallel, making this a scalable model as well. If a bug slipped through in one sprint, it can be fixed in the next sprint and since each spring is a small part of the project, the bug is usually fixable within one sprint as well. This makes this model highly responsive to bugs.
- **Waterfall:** In this model a step in the SDLC is performed only when the previous one is complete. This works well for projects that have fixed requirements. For this very reason this model is also very inflexible. Once a step is complete there is no going back.

1.2. Principles of good software

Writing software is very opinionated, and can be classified as an art more than a science. Everyone has their own preferences and conventions. While writing software can be subjective based on the developer, there are some principles that should be followed:

1. **Functional:** Software should function as dictated by the requirements.
2. **Testable:** It should be possible to test the software, to find any issues with it.
3. **Scaleable:** If the software needs to handle a higher load because more people are using it, then it should be able to without any decrease in performance.
4. **Performant:** Software should use the hardware resources optimally.
5. **Maintainable:** It should be easy to modify, fix or improve the software.
6. **Secure:** Software should prevent unauthorised access to its systems and data.

7. **Reliable:** Software should perform the same consistently.
8. **User friendly:** Software should be easy to use for the client.

There are a few fun acronyms that have popped up over the years to summarise these principles:

1. **DRY (Don't repeat yourself):** Avoid writing the same piece of code multiple times. Instead move it into a function to be re-used. This makes the code more readable.
2. **KISS (Keep it simple, stupid):** Avoid overcomplicating software and write it in its simplest form. This improves readability and maintainability.
3. **YAGNI (You aren't gonna need it):** Only implement software that you need. Don't implement software that you may need in the future. This makes the software unnecessarily complex.

1.3. Version control

In a software projects, it is always a good idea to track the changes that have been made. Consider a team of software engineers and developers working on a navigation application like Google Maps. The app currently shows you the best routes to take if you are driving, walking or taking public transport. The team has decided to introduce a new feature that shows you the best route for cycling from one spot to another. The feature is developed and is deployed to the users. Once the users update their app, they can use this new feature. It turns out that there was a bug in this feature. Whenever the application gives the cycling route it still uses the estimated time from the driving route. The team is notified of this bug and they can revert back to the previous version of the application because they are using version control. Then they can release the app again without the feature and figure out how to fix this bug before releasing it again. In a version control system a software project is known as a *repository*. It is a collection of files.

1.3.1. Git

The most popular version control system is git, an open source version control system. When you create a repository with git, it creates a hidden folder with all the necessary files that git needs to provide history and tracking abilities. These files are automatically updated using git commands and manual editing is not required and strongly not recommended. While it is possible to work on your repository offline, git can let you *push* your repository to a remote server and create a remote repository. Pushing a repository means uploading it to the remote repository. This makes collaboration easier. Figure 1.1 shows how multiple developers can collaborate using git.

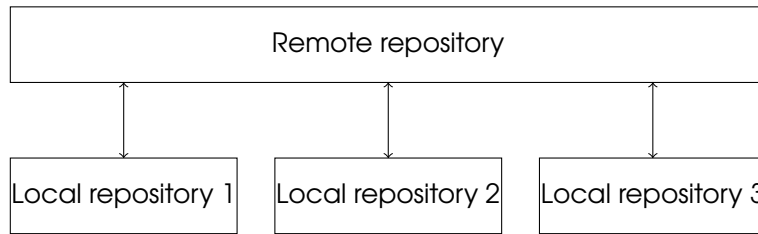


Figure 1.1: Git collaboration

Each developer has a local copy of the remote repository. They can make changes locally and when they are ready, they can push those changes to the remote repository. Any other developer who would like these changes, needs to *pull* from the remote repository. Pulling is the opposite of pushing. It updates the local repository with the remote repository.

The benefits of git are summarised below:

1. **Change tracking:** Records every change that was made to the software and thus providing a timeline of the software. This is analogous to writing a journal. In a journal we store the daily/weekly/monthly updates in our lives. A version control system does the same for software.
2. **Change recovery:** As discussed in the example earlier, by storing the changes made to a piece of software it is possible to revert any changes that caused problems. Tools that provide version control systems, like GitHub, have very nice interfaces to compare 2 versions of software. With these tools, finding the changes that caused problems is much easier.
3. **Collaboration:** Facilitates multiple people working on a project simultaneously. Everyone has a copy of the repository on their local machine. They can make changes locally and push them to the remote repository. If multiple developers are working on the same file and push these changes, git will alert the developers that they must resolve these conflicts first.

Collaboration on git is done using *branches*. There is usually one main branch, that is the most stable version of the repository. Anyone wanting to make changes, needs to make a branch. This way the changes they make will be separate from the main branch. Their changes will not affect the main branch. Once all the changes have been made the branch can be *merged* back into the main branch. Figure 1.1 gives an example of how branches and collaboration works in git. The black line and dots represent the main *branch* of the repository. The dots represent a *commit* or a change to the branch. The coloured lines and dots are

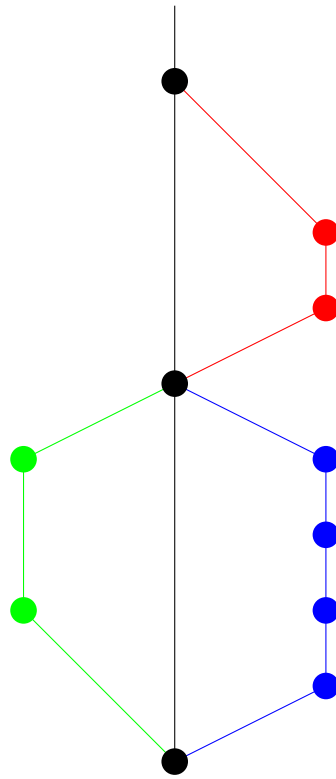


Figure 1.2: Git branching and collaborating

branches. The development of the repository is from bottom to top. The first black dot on the bottom is the first commit that was made to the repository. As you move up, you can track how the repository evolves. The blue branch, for example made 4 changes and then merged back to master.

2

Python

The programming language used for this course is Python. Before we can begin with any programming, we need to setup our laptop/PCs to run Python programs. This chapter is meant as a refresher to Python. We will look at a few concepts and let the reader explore the rest.

2.1. Installing Python

Python downloads page: <https://www.python.org/downloads/>

To install Python visit the official Python downloads page, and follow the instructions for your OS. If you are using Windows, I highly recommend using Windows Subsystem for Linux (WSL), as I am familiar with Unix and Linux systems. For this course we will use Python 3.11, so make sure that you download this version. Once Python is installed on the system, you can confirm this by opening an interactive shell, by running the `python3` command.

```
$ python
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

Listing 2.1: Output of the `python3` command.

This command will print out the version of Python that is installed. The `>>>` is the input for the interactive shell. Any python command typed here will be executed. To exit the interactive shell type `exit()` and press enter.

If you see something other than Python 3.11.X, then you have multiple python versions installed on your system. Use the command `python3.11`. This will ensure that you are using the right version.

```
$ python3.11
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

Listing 2.2: Output of the `python3.11` command.

2.2. Hello world

In typical programming fashion, we begin by writing our first hello world program. A novice programmer always starts by writing a program that outputs the text **Hello World!** when starting their programming journey. A python program is written in a file with the extension `.py`. Each file is called a *module*. To write your first python script, make a file and call it `hello.py`. Then add the line in Listing 2.3 to the script.

```
1 print('Hello World!')
```

Listing 2.3: Hello world in Python.

Run the script by using the command `python hello.py`, and you will see the output **Hello World!** show up in your terminal.

Let make this hello world script a little more personal.

```
1 import sys
2
3 print('Hi ', sys.argv[1])
```

Listing 2.4: Hello world in Python.

If you run the code above you will need to add something to your command from the previous example, `python hello.py Ravi`. Replace Ravi with your name. Lets break down what happens in each line:

- **Line 1:** Here we import the `sys` module. We need this in line 3.
- **Line 3:** This is the `print` command. It takes 2 arguments in this example. The first one is the string "Hi" and the second is an element of a list. We will look at lists later. `sys.argv` is the list of all command-line arguments, with `sys.argv[0]` being the script. In my case `sys.argv[1]` is "Ravi".

2.3. Variable and data structures

Python docs: <https://docs.python.org/>

A good companion to have while going through the next few sections is the official Python documentation. This chapter not cover all the data types offered by Python.

2.3.1. Numbers

In Python there are 2 types of numbers, integers and floats. Integers have no decimal parts and floats do. With type annotation these can be distinguished.

```
1 a: int = 10
2 b: float = 1.5
```

Listing 2.5: `int` and `float` in Python.

`a` is a integer and `b` is a floating point number. Declaring and initialising an `int` and `float` can be done without type annotation.

```
1 a = 10
2 b = 1.5
```

Listing 2.6: `int` and `float` in Python without type annotation.

In this course, we will stick to type annotation to make our code more readable and maintainable.

2.3.2. Strings

Strings in Python are declared with either single quotes or double quotes.

```
1 a: str = 'quantum'
```

Listing 2.7: Declaring and initialised a Python string.

In listing 2.7, the variable `a` holds the value `quantum`. It is also given the type annotation for `str` with `a: str`, which tells IDEs and linters that `a` is of type string. It also improves the readability of the code.

Python has several built in string methods. The `upper()` method converts the string to its upper case form. `a.upper()` converts `quantum` to `QUANTUM`.

```
1 a: str = 'quantum'
2 print(a.upper()) # will print "QUANTUM"
```

Listing 2.8: `upper()` string method.

This is a good time to introduce *mutability*. String in Python are immutable. `a` will contain `quantum` and after `a.upper()` it will still contain `quantum`. The output of `a.upper()` is `QUANTUM`. For a full list of the string methods, look at the Python documentation.

2.3.3. Lists

The string `quantum` is a list of 7 characters. Each character has a position/index in the list.

Character	q	u	a	n	t	u	m
Index	0	1	2	3	4	5	6

Table 2.1: List of characters

Python lists are 0 indexed, which means that the first element has index 0. Each character can be indexed with square brackets.

```
1 a: str = 'quantum'
2 print(a[2]) # this will print "a"
3 print(len(a)) # this will print 7
```

Listing 2.9: Strings as lists in Python.

The method `len()`, prints out the numbers of elements in a list. Lists can have any type of variable in them.

```
1 planets: list[str] = ["mercury", "venus", "earth"]
2 masses: list[float] = [0.0553, 0.815, 1]
```

Listing 2.10: Lists in Python.

`planets` is a list of strings with 3 planets. `masses` is a list of floats with 3 masses. Just as strings, ints and floats have methods, so do lists. The full

String	mercury	venus	earth
Index	0	1	2

Table 2.2: `planets` Python list

Float	0.0553	0.815	1
Index	0	1	2

Table 2.3: `masses` Python list

list of special methods for lists can be found in the Python docs. As an example the `append()` method is shown in listing 2.11.

```
1 masses: list[float] = [0.0553, 0.815, 1]
2 masses.append(0.1075)
3 print(masses) # prints [0.0553, 0.815, 1, 0.1075]
```

Listing 2.11: `append()` method for list.

The `append()` method adds an element to the end of the list. `masses` has a length of 4 after the append operation.

2.3.4. Dictionaries

Dictionaries are key-value pairs.

```
1 planet_masses: dict[str, float] = {  
2     "mercury": 0.0553,  
3     "venus": 0.815,  
4     "earth": 1  
5 }  
6  
7 print(planet_masses["earth"]) # prints 1
```

Listing 2.12: Dictionaries in Python.

`planet_masses` is a dictionary, where the key is a string and the value is a float. Indexing a dictionary is done by providing the key that needs to be read. `planet_masses["earth"]` will have a value of 1. Dictionaries in Python have their own built-in methods as well. The `pop()` method, for example, removes the specified item from the dictionary.

```
1 planet_masses: dict[str, float] = {  
2     "mercury": 0.0553,  
3     "venus": 0.815,  
4     "earth": 1  
5 }  
6  
7 planet_masses.pop("earth")  
8  
9 print(planet_masses)  
10 # prints {'mercury': 0.0553, 'venus': 0.815}
```

Listing 2.13: `pop()` method for dictionary.

2.4. Conditions and loops

2.4.1. if condition

Python code can be executed conditionally.

```
1 light: str = "green"  
2  
3 if light == "green":  
4     # executes next line if light is equal to "green"
```

```
5 print("You can drive.")
```

Listing 2.14: if example.

The if condition can be paired with elif (else if) and else cases.

```
1 light: str = "green"
2
3 if light == "green":
4     # executes next line if light is equal to "green"
5     print("You can drive.")
6 elif light == "yellow":
7     # executes next line if light is equal to "yellow"
8     print("Slow down and prepare to stop.")
9 elif light == "red":
10    # executes next line if light is equal to "yellow"
11    print("You cannot drive.")
12 else:
13    # executes if none of the condition above are true
14    print("Something is wrong.")
15 print("I have gone through all the conditions.")
```

Listing 2.15: if, elif and else example.

Let's break down what happens in listing 2.15:

- **Line 3:** Here the code will check if the variable, `light`, is equal to `green`. If it is then it executes line 5. In this example, this condition is true, so line 5 is executed. Once this happens the program moves to line 15. It skips all the other conditions, since the first one was met.
- **Line 6:** The program arrives at this line if the condition in line 3 was false. Here the code will check if the variable, `light`, is equal to `yellow`. If it is then it executes line 8. If the variable `light` was set to `yellow` in line 1, then after line 8 the program would move to line 15.
- **Line 9:** The program arrives at this line if the conditions in line 3 and line 6 were false. Here the code will check if the variable, `light`, is equal to `red`. If it is then it executes line 11. If the variable `light` was set to `red` in line 1, then after line 11 the program would move to line 15.
- **Line 12:** The program arrives at this line none of the previous conditions were true. Consider this as the default value of this light checking code. If the variable `light` was set to `blue` in line 1, this `else` condition activates and line 14 is executed. Then the program continues on to line 15

2.4.2. for loop

Sometimes you want to run the same code multiple times. This can be achieved with loops in Python. One such loop is the `for` loop.

```
1 for num in range(5):  
2     print(num)
```

Listing 2.16: `for` loop example.

The code in listing 2.16 will print the numbers from 0 to 4, each in a new line. `range` returns a sequence of numbers from 0 to 4. The `for` loop iterates over those numbers and prints it out. In each iteration of the loop, the variable `num` is updated with the next value in the range. The length of range is 5, so `print(num)` is executed 5 times, 1 for each number in the range.

Loops are very helpful when dealing with lists.

```
1 planets: list[str] = ["mercury", "venus", "earth"]  
2 for p in planets:  
3     print(p)
```

Listing 2.17: `for` loop over a list.

Listing 2.17 prints out the name of each planet in `planets` on a new line.

Recall that a string is also a list, so if we iterated over a string, we would iterate over the characters in the string.

```
1 a: str = "quantum"  
2 for c in a:  
3     print(c)
```

Listing 2.18: `for` loop over a list.

The program in listing 2.18 would print out each letter of the word `quantum` on a new line. This loop has 7 iterations since `quantum` is 7 characters long.

2.4.3. while loop

In some cases, we want a piece of code to work, until a condition is met.

```
1 a: int = 0  
2 while a < 50:  
3     a = a + 10  
4     print("I am done!")
```

Listing 2.19: `while` loop example.

In listing 2.19, a `while` condition checks if `a` is less than 50 every iteration. Once it is equal to or greater than 50, it exists the while loop and goes

Iteration	a @ line 2	a @ line 3
1	0	10
2	10	20
3	20	30
4	30	40
5	40	50

Table 2.4: Breaking down listing 2.19

to line 4. Table 2.4 shows what happens in each iteration. After the fifth iteration, the value of `a` is 50, which is no longer less than 50. The program can exit the `while` loop and continue with line 4.

2.5. Functions

Functions allow the reuse of code in Python. Consider the snippet in listing 2.20.

```
1 print("Hi Ravi, welcome to this course!")
2 print("Hi Andreea, welcome to this course!")
3 print("Hi Sandeep, welcome to this course!")
```

Listing 2.20: Repetitive code.

It will print the same message three times, except for the name. This can be refactored, so it is cleaner and there is less repetition.

```
1 def greet_person(name: str) -> None:
2     print(f"Hi {name}, welcome to this course!")
3
4 greet_user("Ravi")
5 greet_user("Andreea")
6 greet_user("Sandeep")
```

Listing 2.21: Example of a function.

`greet_person` is a function. It has one parameter `name` which is of type `string`. `greet_person` has a return type of `None` which means that it does not return anything. It is called by passing an argument to it. For example, the first call to the function is `greet_user("Ravi")`, which passed the argument `Ravi` to the function. The function then uses this argument to print the full string. Using this function has done a few things:

1. Reduced code duplication
2. Made the code easier to read

3. Made the code easier to test

The last part is something that we have not discussed. By abstracting code into a function, that function can be tested without testing anything around it. This is beneficial since tests can become modular. Functions have a `scope`.

```
1 a: int = 10
2 def double_num(num: int) -> int:
3     a: int = 2 * num
4     return a
5
6 double_num(a)
7 print(a) # this will print 10
```

Listing 2.22: Scope of a function.

`double_num` takes a number and doubles it. When it is called, `double_num(a)`, it returns the double of `a` which is 20 in this case. However, the value of `a` remains unchanged. The variable `a` inside `double_num` is limited to the function. Once the function ends and returns, `a` in the function is destroyed.

2.6. Packages

A collection of Python modules is a package. One package that we have used is the `sys` package. We used this package to get the command line arguments passed to a script. Without this package we would have to write the code to get the arguments from scratch. Packages are helpful because the wheel does not need to be reinvented every time. If someone has released a package that satisfies what you want to do, then use that package.

2.6.1. pip package manager

`pip` is the package manager for Python. When we want to use existing packages, like `numpy`, `qiskit` etc. we use `pip` to install these packages. Packages are installed using `pip install numpy`. If you run this command from your terminal you will install `numpy` system wide. Packages are released with a version number, because packages are always being updated with features and bug fixes. Take a look at the Changelog for the `numpy` package at <https://github.com/numpy/numpy/releases>. You will find multiple versions of the `numpy` and each version has changes compared to the previous one.

2.6.2. Virtual environments

With `pip install numpy` the package `numpy` is installed system wide. This means that whenever you use `numpy` in a python project, it uses the

system wide version. With virtual environments we can choose which version of a package we want to use with each project. Consider the following scenario:

1. You have a Python project called ProjectA, which is using the `numpy` that was installed system wide. The system installed `numpy` is v1.0.0.
2. You make a second Python project, ProjectB. In this project you want to use the new `numpy` features that are not available in v1.0.0, but in v2.0.0.
3. You upgrade your system wide `numpy` package to v2.0.0.
4. ProjectA and ProjectB now use `numpy` v2.0.0

The above scenario works great if the only difference between v1.0.0 and v2.0.0 are additions to the package that are backwards compatible. This is best explained with an example of a situation that is not backwards compatible. Lets say that `numpy` v1.0.0 had a function called `addition()` that added 2 arrays, but in v2.0.0 the name of this function was changed to `add()`. In ProjectB, which you are yet to start, this is no a problem, as you can simply use the new function `add()`, however for ProjectA, which you have already finished, `numpy.add()` does not exist. This is bad because ProjectA is now broken. There are 2 ways to fix this:

1. Rewrite ProjectA to use `numpy` v2.0.0
2. Use virtual environments

Virtual environments create an isolated environment for your project. In this course we will use the module `venv`. It is good practice to create this virtual environment inside the root folder of your Python project.

```
$ python -m venv .venv
```

Listing 2.23: Creating a `venv`.

When you run the command in listing 2.23 from the root directory of your project, you will find a folder `.venv`, which contains everything needed for the virtual environment. Once this is made you can activate the virtual environment using `source .venv/bin/activate` from the root directory of your project.

```
$ source .venv/bin/activate
(.venv) $
```

Listing 2.24: Activating a `venv`.

`(.venv)` tells you that you are currently in the virtual environment named `.venv`. Anything you now install with `pip` will not be installed system wide. Instead it will be installed in this virtual environment and placed inside

the folder `.venv`. Now you have have a different virtual environment for each of your projects. The dependencies of all projects will be isolated. To exit the virtual environment use the command `deactivate`.

```
(.venv) $ deactivate
$
```

Listing 2.25: Deactivating a venv.

3

Quantum computing

3.1. Motivation

Classical computing has been evolving to help solve harder problems. Following Moore's Law the miniaturisation of transistors has helped improve performance by increasing the density of transistors on a chip. But this law has its limits. As these transistors got smaller, they experienced more heat dissipation, which limited the improvements in the clock frequency of CPUs. This, along with a few other factors, led the industry to move to multi-core processors and hardware accelerators. The hardware accelerators we are all familiar with is the GPU (graphics processing unit). As the name suggests it is an accelerator to process graphics. It is used extensively for image processing and is much faster at doing this than a CPU. Approaching the limits of classical computing means that new ways of computing are needed. Quantum computing is one such paradigm.

There are certain of problems that cannot be solved, or rather will take too long to solve on a classical computer, making it practically infeasible. One such problem is the prime-factorisation problem. In 1994 Peter Shor came up with a prime-factorisation algorithm that could be completed in polynomial time on a quantum computer [1]. The widely used RSA cryptosystem and several other cryptosystems depend on the hardness of the prime-factorisation problem. With Shor's algorithm this problem is no longer hard. Another application of quantum computing is the ability to simulation quantum systems itself. Google was able to simulate a hydrogen molecule [2] using a quantum computer. Being able to simulate molecules accurately and in reasonable time will speed-up the development of drugs.

3.2. Quantum acceleration

A quantum computer does not replace a classical computer. Instead it will be used as an accelerator, similar to GPUs and FPGAs. The training of AI models is primarily done on GPUs. In the last couple years NVIDIA has skyrocketed due to its ability to provide GPUs for AI training. While these models are trained on GPUs, CPUs are still needed to talk to these GPUs. They act as hosts for the GPU.

An accelerator is a special piece of hardware that accelerates a specific task. Accelerators as co-processors are shown in figure 3.1. A generic accelerator is viewed as a separate I/O device connected to the host CPU. QPU is the *quantum processing unit*.

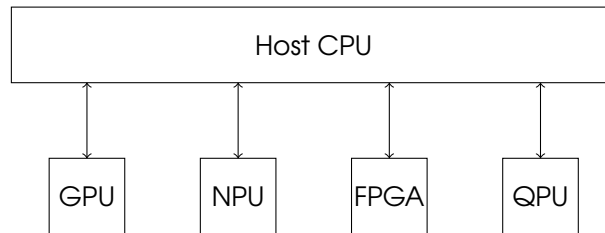


Figure 3.1: Accelerator as a co-processor

Quantum computing will not replace classical computing completely. It is a type of accelerator that speeds up certain types of tasks. The CPU will allocate a certain task to it, which the QPU will execute and send the results back to the host CPU.

3.3. Ecosystem

There are several companies working on making quantum computing a reality. Listed below are just a few examples.

1. **Google:** Their latest quantum chip, Willow, is a 105 qubit chip, finished a standard benchmark computation in under 5 minutes [3]. A classical computer would take 10 septillion years to do that. Willow is milestone 2 of the 6 milestones set by Google Quantum AI.
2. **IBM:** While we are familiar with IBM because of Qiskit, which is what we will learn in this course, they also develop quantum processing units. Their latest is the Heron chip which has 156 qubits.
3. **QBLOX:** They provide a hardware solution to control and readout qubits. It is a cluster that contains several instruments like sequence processors, arbitrary waveform generators, digitizers etc. Their modular design makes it very easy to scale up the system, and specialised modules can easily be added to the cluster.

4. **Single Quantum:** They provide photon detectors that can measure individual photons. As a results of this high sensitivity to photons these devices can not only be used in the quantum domain, but also in Lidar and atmospheric measurements.
5. **Q*Bird:** They build solutions for secure data communication. The latest family of devices in the Falcon series can create a network between users that allows users to share keys with each other that remain secret to the rest of the network. Furthermore their solutions are easily scaleable as it does not require a complete redesign of the network.

Notice, that the ecosystem does not only include companies that build the QPUs or the software frameworks for quantum, but also the support electronics that are needed to control those QPUs. This is analogous to how the classical computer is not just the CPU, but also the GPU, the motherboard, the power supply unit and all the way don to the cables.

4

Quantum information

4.1. The classical bit

Classical information exists all around us and not just in computers.

- A light bulb has 2 states. It can either be **ON** or **OFF**.
- A traffic light has 3 states. It can either be **RED**, **YELLOW** or **GREEN**.
- Flipping a coin yields 2 states. It can either be **HEADS** or **TAILS**.

In classical computing, information is stored as **bits**. A bit can be in one of two possible states: either **0** or **1**. In other words, classical computing uses a 2 state system to store information. With multiple bits, it can store information like letters, numbers, special characters, and more. The American Standard Code for Information Interchange (ASCII) uses 8 bits, also known as a byte, to represent 256 characters and symbols. 8 bits can represent 2^8 unique combinations. Examples of ASCII characters with their binary representation are shown in 4.1.

Binary	ASCII character
00100110	&
01000001	A
01011010	Z
01100001	a

Table 4.1: Snippet showing ASCII characters

4.2. The quantum bit

The quantum analogue of a bit is a *quantum bit* or *qubit*. A qubit is a two-state quantum mechanical system. Unlike a classical bit, a qubit is not a scalar, but a vector. Before diving into why this is the case, let us introduce the bra-ket (also known as Dirac) notation to represent a qubit. The 2 classical states **0** and **1** are written as $|0\rangle$ and $|1\rangle$ respectively. These 2 *kets* are vector representations of the state and are equivalent to the following

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}. \quad (4.1)$$

Multiple qubits are represented as a tensor product of individual qubits. For example given 2 qubits $|0\rangle$ and $|1\rangle$, the 2 qubit state, $|01\rangle$, can be written as a tensor product of the individual qubits,

$$|0\rangle \otimes |1\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \begin{bmatrix} 0 \\ 1 \end{bmatrix} \\ 0 \begin{bmatrix} 0 \\ 1 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}. \quad (4.2)$$

4.2.1. Superposition

The real power of quantum information lies in having a combination of quantum states represented by a qubit. Think of a flipping a coin. When it is in the air, it is spinning and you do not know if it is heads or tails. Once it lands, you know the outcome. One could say that when the coin is in the air it is in a superposition of **HEADS** or **TAILS**. There is no way to tell until it lands and we look at it. Let us make a 2 state quantum system from this. The 2 states are $|H\rangle$ for **HEADS** and $|T\rangle$ for **TAILS**,

$$|H\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad |T\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}. \quad (4.3)$$

$|H\rangle$ and $|T\rangle$ make up the *basis* for this system. A basis is a set of states that span the state space of a quantum system. $|0\rangle$ and $|1\rangle$ make up a basis, called the *computational basis* or the *standard basis*. Similarly, there is the *Hadamard basis*, which is made up of the states $|+\rangle$ and $|-\rangle$. This basis can be written in terms of the computational basis

$$|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \quad |-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle). \quad (4.4)$$

In some cases it is easier to represent qubits in this basis. Going back to the coin example, in the air, the coin is in a superposition of **HEADS** and **TAILS**. This superposition can be written as

$$|\psi\rangle = \alpha |H\rangle + \beta |T\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}, \quad (4.5)$$

where $|\psi\rangle$ is the quantum state of the coin in the air, and α and β are probability amplitudes. They represent the likelihood of $|\psi\rangle$ being in the state $|H\rangle$ or $|T\rangle$ respectively. These values are complex numbers, which means that they have a real part and an imaginary part

$$\alpha, \beta \in \mathbb{C}. \quad (4.6)$$

The general form of α and β can be written as

$$\alpha = a + b \cdot i \quad \beta = c + d \cdot i \quad (4.7)$$

where i is the imaginary number. a and c are the real parts of α and β respectively and b and d are the imaginary parts of α and β respectively. Another way to write a complex number is using Euler's formula,

$$e^{i\theta} = \cos(\theta) + i \cdot \sin(\theta). \quad (4.8)$$

Using this form α and β can be written as

$$\alpha = r_1(\cos(\theta_1) + i \cdot \sin(\theta_1)) = r_1 e^{i\theta_1} \quad (4.9)$$

$$\beta = r_2(\cos(\theta_2) + i \cdot \sin(\theta_2)) = r_2 e^{i\theta_2}. \quad (4.10)$$

r_1 and r_2 are scalar constants to scale the numbers. We can write the state of a qubit in superposition using this notation

$$|\psi\rangle = r_1 e^{i\theta_1} |H\rangle + r_2 e^{i\theta_2} |T\rangle. \quad (4.11)$$

4.2.2. Measurement

Now that we know how to describe the coin in the air, let us look at what happens when it lands. When the coin lands, we are measuring this quantum state. The coin will land in $|H\rangle$ or $|T\rangle$. We can say that it has *collapsed* to one of the states $|H\rangle$ or $|T\rangle$ with probabilities $|\alpha|^2$ and $|\beta|^2$, respectively. Since the sum of the probabilities of an event add up to one, we can derive a mathematical relationship for $|\alpha|^2$ and $|\beta|^2$,

$$|\alpha|^2 + |\beta|^2 = 1. \quad (4.12)$$

Let us see how this measurement works in the world of linear algebra, and how we came up with the terms $|\alpha|^2$ and $|\beta|^2$. Let us introduce another term called the *measurement basis*, which is nothing more than the possible states that the coin can end up in. The measurement basis, M , for the coin flip is

$$M = \{|H\rangle, |T\rangle\}. \quad (4.13)$$

If you look at the vector representation of these 2 states, you will see that they are orthogonal/perpendicular to each other. They are also of unit length, i.e. the norm of each vector is 1. This makes the 2 states *orthonormal*. Earlier we introduced the *bra-ket* notation and used *kets*

to denote a qubit. To perform a measurement we project the state $|\psi\rangle$ onto the state in the measurement basis that we want to measure. This is where we use the *bra* representation of a qubit. *kets* are column vectors and *bras* are row vectors.

$$\langle H| = [1 \quad 0] \quad \langle T| = [0 \quad 1]. \quad (4.14)$$

If we want to measure the probability that the coin will be **HEADS** when it lands, let's call that $P(H)$, we project the state $|\psi\rangle$ onto $|H\rangle$. Mathematically, we can say that the probability that the coin is **HEADS** when it lands, $P(H)$, is the squared inner product of $|\psi\rangle$ onto $|H\rangle$.

$$P(H) = |\langle H|\psi\rangle|^2 = |[1 \quad 0] \cdot \begin{bmatrix} \alpha \\ \beta \end{bmatrix}|^2 = |\alpha|^2. \quad (4.15)$$

Similarly, the probability that the coin is **TAILS** when it lands is,

$$P(T) = |\langle T|\psi\rangle|^2 = |[0 \quad 1] \cdot \begin{bmatrix} \alpha \\ \beta \end{bmatrix}|^2 = |\beta|^2. \quad (4.16)$$

For a coin that is perfectly balanced and flipped without any bias, we know that the probability of getting **HEADS** or **TAILS** is equal, i.e. $P(H) = P(T) = 50\%$. For this unbiased coin, the quantum state would be,

$$|\psi\rangle = \frac{1}{\sqrt{2}} |H\rangle + \frac{1}{\sqrt{2}} |T\rangle. \quad (4.17)$$

4.2.3. Entanglement

While single qubit states are easy to understand, for something more practical and useful we need multiple qubits. Let's introduce multi qubit quantum systems, by introducing 2 parties, Alice and Bob. Let's say that each party has the quantum state of our unbiased coin, $|\psi\rangle$,

$$|\psi\rangle_A = \alpha |H\rangle_A + \beta |T\rangle_A \quad (4.18)$$

$$|\psi\rangle_B = \alpha |H\rangle_B + \beta |T\rangle_B. \quad (4.19)$$

The shared quantum state can be written as

$$|\psi\rangle_{AB} = \left(\frac{1}{\sqrt{2}} |H\rangle_A + \frac{1}{\sqrt{2}} |T\rangle_A\right) \left(\frac{1}{\sqrt{2}} |H\rangle_B + \frac{1}{\sqrt{2}} |T\rangle_B\right) \quad (4.20)$$

$$|\psi\rangle_{AB} = \frac{1}{2} |H\rangle_A |H\rangle_B + \frac{1}{2} |H\rangle_A |T\rangle_B + \frac{1}{2} |T\rangle_A |H\rangle_B + \frac{1}{2} |T\rangle_A |T\rangle_B \quad (4.21)$$

$$|\psi\rangle_{AB} = \frac{1}{2} |HH\rangle_{AB} + \frac{1}{2} |HT\rangle_{AB} + \frac{1}{2} |TH\rangle_{AB} + \frac{1}{2} |TT\rangle_{AB}. \quad (4.22)$$

This state, $|\psi\rangle_{AB}$, can be separated into its single qubit states, $|\psi\rangle_A$ and $|\psi\rangle_B$, since it was constructed by taking the product of the 2 states.

When a 2 qubit state cannot be separated into 2 one qubit states, we say that the state is *entangled*. The state $|\phi\rangle_{AB}$ is entangled,

$$|\phi\rangle_{AB} = \frac{1}{\sqrt{2}} |HH\rangle_{AB} + \frac{1}{\sqrt{2}} |TT\rangle_{AB}. \quad (4.23)$$

It is impossible to know the individual qubit states for both Alice and Bob. Their states are entangled with each other. For a 2 qubit system the measurement basis, M , would be

$$M = \{|HH\rangle, |HT\rangle, |TH\rangle, |TT\rangle\}. \quad (4.24)$$

Recall that the states in a measurement basis, must be *orthonormal*. A good exercise would be to prove that for the above measurement basis.

$|\phi\rangle$ has 2 possible outcomes, $|HH\rangle$ and $|TT\rangle$. When a measurement takes place, these are the 2 outcomes that the quantum system can collapse to. Alice and Bob will share the same state after measurement, **HEADS** if the system collapses to $|HH\rangle$ and **TAILS** if the system collapses to $|TT\rangle$. Alice will know that if she measures $|H\rangle$, then Bob also measured $|H\rangle$. The same applies for Bob.

This is not the case for $|\psi\rangle$, since the states of Alice and Bob are not entangled and separable. If Alice measured $|H\rangle$, then the shared quantum state would be

$$|\psi\rangle_{AB, A=H} = |H\rangle_A \cdot \frac{1}{\sqrt{2}} (|H\rangle_B + |T\rangle_B). \quad (4.25)$$

It might not be obvious how the probability amplitudes changed. There are 2 ways to understand this. The first one is to understand that the probabilities must add up to 1 for a quantum state, so when we measure part of it, we must normalise the state. The second is to split up the probability amplitudes of $|\psi\rangle$, so we can see which state contributes to the probability amplitudes

$$\begin{aligned} |\psi\rangle_{AB} &= \frac{1}{\sqrt{2}} |H\rangle_A \cdot \frac{1}{\sqrt{2}} |H\rangle_B + \frac{1}{\sqrt{2}} |H\rangle_A \cdot \frac{1}{\sqrt{2}} |T\rangle_B \\ &\quad + \frac{1}{\sqrt{2}} |T\rangle_A \cdot \frac{1}{\sqrt{2}} |H\rangle_B + \frac{1}{\sqrt{2}} |T\rangle_A \cdot \frac{1}{\sqrt{2}} |T\rangle_B. \end{aligned} \quad (4.26)$$

The $\frac{1}{2}$ probability amplitudes is made up of $\frac{1}{\sqrt{2}}$ from Alice, and the same from Bob. When Alice measures $|H\rangle$, we can remove the terms with $|T\rangle_A$ and the probability amplitude contributed by $|H\rangle_A$ is one, since it has collapsed into a classical state. This gives,

$$|\psi\rangle_{AB, A=H} = |H\rangle_A \cdot \frac{1}{\sqrt{2}} |H\rangle_B + |H\rangle_A \cdot \frac{1}{\sqrt{2}} |T\rangle_B \quad (4.27)$$

5

Manipulating qubits

5.1. The logic gate

When performing computations like additions, subtractions etc., computers use logic gates to manipulate bits. An example of such a gate is an AND gate. An AND gate takes n inputs and outputs 1 if all the inputs are 1, otherwise it outputs 0. An example of a 2 input AND gate is shown in figure 5.1. For every logic gate, we can construct a table to show its

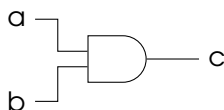


Figure 5.1: 2 input AND gate

output given its input(s). Such a table is called a *truth table*. The truth table for the 2 input AND gate in figure 5.1 is given in table 5.1.

a	b	c
0	0	0
0	1	0
1	0	0
1	1	1

Table 5.1: Truth table for 2 input AND gate

Logic gates are the building blocks in digital electronics. They perform logical functions. Combining logic gates results in complex circuits that can perform mathematical operations. Table 5.2 lists all the logic gates. It is often easier to write the logic of logic gates as boolean expressions,

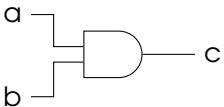
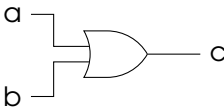
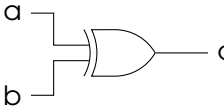
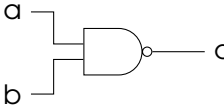
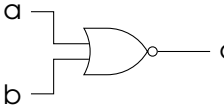
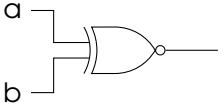
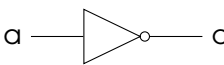
Name	Truth table	Circuit Symbol	Boolean expression															
AND	<table><tr><th>a</th><th>b</th><th>c</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	a	b	c	0	0	0	0	1	0	1	0	0	1	1	1		$c = a \cdot b$
a	b	c																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
OR	<table><tr><th>a</th><th>b</th><th>c</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	a	b	c	0	0	0	0	1	1	1	0	1	1	1	1		$c = a + b$
a	b	c																
0	0	0																
0	1	1																
1	0	1																
1	1	1																
XOR	<table><tr><th>a</th><th>b</th><th>c</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	a	b	c	0	0	0	0	1	1	1	0	1	1	1	0		$c = a \oplus b$
a	b	c																
0	0	0																
0	1	1																
1	0	1																
1	1	0																
NAND	<table><tr><th>a</th><th>b</th><th>c</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	a	b	c	0	0	1	0	1	1	1	0	1	1	1	0		$c = \overline{a \cdot b}$
a	b	c																
0	0	1																
0	1	1																
1	0	1																
1	1	0																
NOR	<table><tr><th>a</th><th>b</th><th>c</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	a	b	c	0	0	1	0	1	0	1	0	0	1	1	0		$c = \overline{a + b}$
a	b	c																
0	0	1																
0	1	0																
1	0	0																
1	1	0																
XNOR	<table><tr><th>a</th><th>b</th><th>c</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	a	b	c	0	0	1	0	1	0	1	0	0	1	1	1		$c = \overline{a \oplus b}$
a	b	c																
0	0	1																
0	1	0																
1	0	0																
1	1	1																
NOT	<table><tr><th>a</th><th>c</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	a	c	0	1	1	0		$c = \bar{a}$									
a	c																	
0	1																	
1	0																	

Table 5.2: Logic gates

especially as these are used to create complex circuits.

Apart from the NOT gate which is fixed to one input, logic gates can easily be extended to n inputs, since they are based on boolean logic. For a 4 input AND gate, the truth table is given in table 5.3. The 4 input

Input 1	Input 2	Input 3	Input 4	Output
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Table 5.3: Truth table for 4 input AND gate

AND gate only outputs a 1 if all inputs are 1. The boolean expression for a 4 input AND gate is

$$o = i_1 \cdot i_2 \cdot i_3 \cdot i_4, \quad (5.1)$$

where o is the output and i_i is the list of inputs.

5.2. The quantum gate

The quantum gate is the quantum analogue to the logical gate. It manipulates qubits just as logical gates manipulate bits. Quantum gates can be represented as matrices. This fits with our earlier representation of qubits as vectors. Quantum gates are reversible. This is easier to explain with an example. The Pauli X gate is the quantum analogue of the logical NOT gate. It inverts $|0\rangle$ to $|1\rangle$ and vice versa,

$$X |0\rangle = |1\rangle \quad X |1\rangle = |0\rangle. \quad (5.2)$$

Its matrix representation is

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}. \quad (5.3)$$

The conjugate transpose, i.e. the gate that reverse the operation, of the X gate is X^\dagger ,

$$X^\dagger = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}. \quad (5.4)$$

For the X gate the conjugate transpose is the same as the gate itself. It has the same effect on the qubit, so if the X gate flips $|0\rangle$ to $|1\rangle$, X^\dagger flips $|1\rangle$ back to $|0\rangle$. A mathematical relationship between a gate and its conjugate transpose is that they multiply to give the identity matrix,

$$XX^\dagger = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = I. \quad (5.5)$$

This relationship reiterates the reversible nature of quantum gates,

$$XX^\dagger = I|0\rangle = |0\rangle. \quad (5.6)$$

A gate that, U , that satisfies this relationship, $UU^\dagger = I$ is called a unitary matrix. Quantum gates are *Unitary*

The Pauli X gate is one of the 3 gates in the Pauli family. The entire set of Pauli gates is shown in table 5.4.

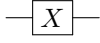
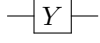
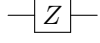
Name	Matrix	Circuit Symbol
Pauli-X	$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$	
Pauli-Y	$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$	
Pauli-Z	$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$	

Table 5.4: Pauli gates

Pauli gates are a specific version of the generalised rotation gates. These generalised rotation gates rotate the qubit along a specific axis. In this course we have only looked at the vector representation of a qubit. Another representation is the Bloch sphere representation, which is a sphere that shows the qubit in 3 dimensional space. It has 3 axis, X , Y and Z . These rotation gates, rotate the qubit along these axis given an angle. The rotation gates are show in 5.5.

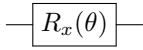
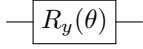
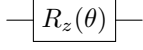
Name	Matrix	Circuit Symbol
Rotation-X	$R_x(\theta) = \begin{bmatrix} \cos \frac{\theta}{2} & -i \sin \frac{\theta}{2} \\ -i \sin \frac{\theta}{2} & \cos \frac{\theta}{2} \end{bmatrix}$	
Rotation-Y	$R_y(\theta) = \begin{bmatrix} \cos \frac{\theta}{2} & -\sin \frac{\theta}{2} \\ \sin \frac{\theta}{2} & \cos \frac{\theta}{2} \end{bmatrix}$	
Rotation-Z	$R_z(\theta) = \begin{bmatrix} e^{-i \frac{\theta}{2}} & 0 \\ 0 & e^{i \frac{\theta}{2}} \end{bmatrix}$	

Table 5.5: Rotation gates

There are a few other single qubit gates that are important to know. These are shown in 5.6. The Hadamard gate is a special gate, as it is the




Name	Matrix	Circuit Symbol
Hadamard	$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$	
Phase	$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$	
T	$T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i \frac{\pi}{4}} \end{bmatrix}$	

Table 5.6: Other single qubit gates

gate that creates superposition,

$$H|0\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle. \quad (5.7)$$

It does the same for the $|1\rangle$ state except with a negative on the $|1\rangle$ state,

$$H|1\rangle = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle. \quad (5.8)$$

It also changes the basis from the *standard basis* to the *Hadamard basis*,

$$H|0\rangle = |+\rangle \quad H|1\rangle = |-\rangle. \quad (5.9)$$

The Hadamard gate is also used to switch back to the *standard basis*,

$$H|+\rangle = |0\rangle \quad H|-\rangle = |1\rangle. \quad (5.10)$$

Quantum gates also have multi qubit gates. An example of one such gate is the CNOT, or the controlled NOT. This gate takes one qubit as the control and one as the target. If the control is $|1\rangle$ it flips the target,

$$CNOT|00\rangle = |00\rangle \quad CNOT|10\rangle = |11\rangle. \quad (5.11)$$

The most commonly used ones are shown in table 5.7

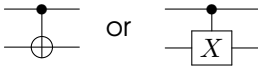
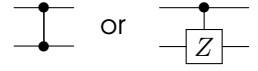

Name	Matrix	Circuit Symbol
Controlled Not	$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$	
Controlled Z	$CZ = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$	
Swap	$SWAP = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	

Table 5.7: Commonly used 2 qubit gates

These gates can be extending to more than 2 qubits. The 3 qubit version of the CNOT is the CCNOT or Toffoli gate. It has 2 control qubits

compared to the 1 control qubit that the CNOT has. When both control qubits are $|1\rangle$ the target qubit is flipped,

$$CCNOT|000\rangle = |000\rangle \quad CCNOT|110\rangle = |111\rangle. \quad (5.12)$$

The gate does nothing to the state of the control qubits. They are unchanged. 2 examples of 3 qubits gates are shown in table 5.8.



Name	Unitary	Circuit Symbol
Toffoli	$CCNOT = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$	
Fredkin	$CSWAP = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$	

Table 5.8: Commonly used 3 qubit gates

5.2.1. Bracket notation

Quantum gates can also be written in the *braket* notation. This makes it easier to evaluate the effect a gate has on a quantum state. Transforming the whole system to vectors and matrices is no longer required.

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad \langle 0| = [1 \quad 0] \quad (5.13)$$

$$|1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad \langle 1| = [0 \quad 1] \quad (5.14)$$

The outer product of $|0\rangle$ and $\langle 0|$ is given by

$$|0\rangle \langle 0| = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}. \quad (5.15)$$

By combining outer products of the basis states we can create the matrix representation of quantum gates. The X gate can be written as

$$X = |1\rangle \langle 0| + |0\rangle \langle 1| \quad (5.16)$$

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad (5.17)$$

$$X = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}. \quad (5.18)$$

Vectors have an inner product as well, as we saw in 4.2.2. The basis states $[|0\rangle, |1\rangle]$ are orthogonal, which means that

$$\langle 0|0\rangle = 1 \quad \langle 0|1\rangle = 0 \quad \langle 1|0\rangle = 0 \quad \langle 1|1\rangle = 1. \quad (5.19)$$

Applying the new notation of the X gate to $|0\rangle$ gives

$$X|0\rangle = (|1\rangle \langle 0| + |0\rangle \langle 1|)|0\rangle \quad (5.20)$$

$$X|0\rangle = |1\rangle \langle 0|0\rangle + |0\rangle \langle 1|0\rangle \quad (5.21)$$

$$X|0\rangle = |1\rangle. \quad (5.22)$$

Multi qubit gates can also be written in the bracket notation.

$$CNOT = |00\rangle \langle 00| + |01\rangle \langle 01| + |11\rangle \langle 10| + |10\rangle \langle 11|. \quad (5.23)$$

Reading the gates in this notation gives an intuitive understanding behind the function it performs. Every term has a *bra* and *ket*. The gate takes the *bra* term to the *ket* term.

5.2.2. Global phase

In the *computational basis* the equation 4.11 can be rewritten as

$$|\psi\rangle = e^{i\theta_1}(r_1|0\rangle + r_2e^{i(\theta_2-\theta_1)}|1\rangle). \quad (5.24)$$

If we applying the Hadamard gate to this state we get

$$H|\psi\rangle = e^{i\theta_1}\left(r_1\left(\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)\right) + r_2e^{i(\theta_2-\theta_1)}\left(\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)\right)\right) \quad (5.25)$$

$$H|\psi\rangle = e^{i\theta_1}\frac{1}{\sqrt{2}}((r_1 + r_2e^{i(\theta_2-\theta_1)})|0\rangle + (r_1 - r_2e^{i(\theta_2-\theta_1)})|1\rangle). \quad (5.26)$$

The probability amplitudes of $|0\rangle$, lets call it α , and $|1\rangle$, lets call it β , are

$$\alpha = e^{i\theta_1} \frac{1}{\sqrt{2}} (r_1 + r_2 e^{i(\theta_2 - \theta_1)}) \quad (5.27)$$

$$\beta = e^{i\theta_1} \frac{1}{\sqrt{2}} (r_1 - r_2 e^{i(\theta_2 - \theta_1)}). \quad (5.28)$$

Before calculating the probabilities $|\alpha|^2$ and $|\beta|^2$, lets write out the complex conjugate of the imaginary terms

$$(e^{i\theta_1})^* = e^{-i\theta_1} \rightarrow e^{i\theta_1} \cdot e^{-i\theta_1} = 1 \quad (5.29)$$

$$(e^{i(\theta_2 - \theta_1)})^* = e^{i(\theta_1 - \theta_2)} \rightarrow e^{i(\theta_2 - \theta_1)} \cdot e^{i(\theta_1 - \theta_2)} = 1. \quad (5.30)$$

The probability of measuring $|0\rangle$, P_0 , is

$$P_0 = |\alpha|^2 \quad (5.31)$$

$$P_0 = |e^{i\theta_1} \frac{1}{\sqrt{2}} (r_1 + r_2 e^{i(\theta_2 - \theta_1)})|^2 \quad (5.32)$$

$$P_0 = \frac{1}{2} \left(\sqrt{e^{i\theta_1} \cdot e^{-i\theta_1} (r_1 + r_2 e^{i(\theta_2 - \theta_1)}) (r_1 + r_2 e^{i(\theta_1 - \theta_2)})} \right)^2 \quad (5.33)$$

$$P_0 = \frac{1}{2} (r_1 + r_2 e^{i(\theta_2 - \theta_1)}) (r_1 + r_2 e^{i(\theta_1 - \theta_2)}) \quad (5.34)$$

$$P_0 = \frac{1}{2} (r_1^2 + r_1 r_2 e^{i(\theta_1 - \theta_2)} + r_1 r_2 e^{i(\theta_2 - \theta_1)} + r_2 e^{i(\theta_2 - \theta_1)} r_2 e^{i(\theta_1 - \theta_2)}) \quad (5.35)$$

$$P_0 = \frac{1}{2} (r_1^2 + r_1 r_2 e^{i(\theta_1 - \theta_2)} + r_1 r_2 e^{i(\theta_2 - \theta_1)} + r_2^2). \quad (5.36)$$

The $e^{i\theta_1}$ term is gone from the probability. This is the *global phase* of the quantum system. This shows that the global phase has no effect on the quantum system can be ignored. The relative phase $e^{i(\theta_1 - \theta_2)}$ and $e^{i(\theta_2 - \theta_1)}$ remain and do have an effect on the quantum system. The same can be done for the probability of measure $|1\rangle$, The probability of measuring $|1\rangle$, P_1 , is

$$P_1 = |\beta|^2 \quad (5.37)$$

$$P_1 = |e^{i\theta_1} \frac{1}{\sqrt{2}} (r_1 - r_2 e^{i(\theta_2 - \theta_1)})|^2 \quad (5.38)$$

$$P_1 = \frac{1}{2} \left(\sqrt{e^{i\theta_1} \cdot e^{-i\theta_1} (r_1 - r_2 e^{i(\theta_2 - \theta_1)}) (r_1 - r_2 e^{i(\theta_1 - \theta_2)})} \right)^2 \quad (5.39)$$

$$P_1 = \frac{1}{2} (r_1 - r_2 e^{i(\theta_2 - \theta_1)}) (r_1 - r_2 e^{i(\theta_1 - \theta_2)}) \quad (5.40)$$

$$P_1 = \frac{1}{2} (r_1^2 - r_1 r_2 e^{i(\theta_1 - \theta_2)} - r_1 r_2 e^{i(\theta_2 - \theta_1)} + r_2 e^{i(\theta_2 - \theta_1)} r_2 e^{i(\theta_1 - \theta_2)}) \quad (5.41)$$

$$P_1 = \frac{1}{2}(r_1^2 - r_1 r_2 e^{i(\theta_1 - \theta_2)} - r_1 r_2 e^{i(\theta_2 - \theta_1)} + r_2^2). \quad (5.42)$$

Since the global phase has no effect on the state, the state $|\psi\rangle$ is equal to $e^{i\theta} |\psi\rangle$. The following are some examples of quantum states with a global phase,

$$|\psi\rangle = e^{i\pi} |\psi\rangle = -1 |\psi\rangle \quad (5.43)$$

$$|\psi\rangle = e^{i\frac{\pi}{2}} |\psi\rangle = i |\psi\rangle \quad (5.44)$$

$$|\psi\rangle = e^{i\frac{\pi}{4}} |\psi\rangle = \left(\frac{1}{\sqrt{2}} + i\frac{1}{\sqrt{2}}\right) |\psi\rangle. \quad (5.45)$$

6

Quantum circuits

6.1. Entanglement generation

To recap, an entangled 2 qubit state is,

$$|\psi\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}. \quad (6.1)$$

The circuit to generate this state is shown in figure 6.1.

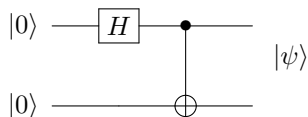


Figure 6.1: Quantum circuit to generate 2 qubit entanglement

Lets call the input state to this circuit $|\phi\rangle$, which is the state $|00\rangle$. The first gate is the Hadamard gate on the top qubit, followed by a CNOT gate, with the top qubit as the control and the bottom qubit as the target. The CNOT gate is a controlled Pauli X gate, which means that if the top qubit is $|1\rangle$ then the bottom qubit is flipped. The first set of gates that are applied are the Hadamard gate on the top qubit and the Identity gate on the bottom qubit. When no gate is applied, it is equivalent to applying the Identity gate on the qubit state vector. The resultant gate is given by the Kronecker product of the Hadamard gate, H , and the Identity gate, I ,

$$H \otimes I = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}. \quad (6.2)$$

Given two 2 by 2 matrices g_1 and g_2 ,

$$g_1 = \begin{bmatrix} a_1 & b_1 \\ c_1 & d_1 \end{bmatrix} \quad g_2 = \begin{bmatrix} a_2 & b_2 \\ c_2 & d_2 \end{bmatrix} \quad (6.3)$$

the Kronecker product, u , of these matrices is

$$u = \begin{bmatrix} a_1 \begin{bmatrix} a_2 & b_2 \\ c_2 & d_2 \end{bmatrix} & b_1 \begin{bmatrix} a_2 & b_2 \\ c_2 & d_2 \end{bmatrix} \\ c_1 \begin{bmatrix} a_2 & b_2 \\ c_2 & d_2 \end{bmatrix} & d_1 \begin{bmatrix} a_2 & b_2 \\ c_2 & d_2 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} a_1 a_2 & a_1 b_2 & b_1 a_2 & b_1 b_2 \\ a_1 c_2 & a_1 d_2 & b_1 c_2 & b_1 d_2 \\ c_1 a_2 & c_1 b_2 & d_1 a_2 & d_1 b_2 \\ c_1 c_2 & c_1 d_2 & d_1 c_2 & d_1 d_2 \end{bmatrix} \quad (6.4)$$

Therefore the Kronecker product of H and I is

$$H \otimes I = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}. \quad (6.5)$$

The state of the qubits after the Hadamard gate is

$$|\phi\rangle_H = H \otimes I |\phi\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}. \quad (6.6)$$

This can be written using the bra-ket notation,

$$|\phi\rangle_H = \frac{1}{\sqrt{2}}(|00\rangle + |10\rangle). \quad (6.7)$$

The CNOT gate is then applied to this state,

$$|\psi\rangle = CNOT \cdot |\phi\rangle_H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}. \quad (6.8)$$

This is the entangled state state in equation 6.1.

6.2. Grover's Algorithm

Grover's algorithm is an example of a quantum algorithm that offers a quadratic speed-up over its classical counterpart. The algorithm searches for a item in an unsorted database. For example, take a setup with 4 cups flipped over with one hiding a coin. In the worst case you will need to flip at least 3 cups to find the coin. In the classical algorithm this will take $O(N)$ time, which means that the execution time grows with the input size N . Grover's algorithm offers a speed-up that takes $O(\sqrt{N})$.

There are 2 ways to look at Grover's algorithm. The difference lies in the oracle, which is the black box that tags the solution in the database. There are 2 ways to implement the oracle that tags the solution. One uses a Boolean oracle to tag the solution and the other uses a phase oracle. The boolean oracle uses an ancillary qubit and implementing this oracle classically would flip this ancillary qubit if the input to the circuit was the solution. This is exactly what happens with the quantum version as well. The ancillary qubit is flipped for the solution state. In quantum computing, it is possible to initialise the state to the system with a superposition of all possible states and therefore the ancillary qubit is no longer required. The point of using the Boolean oracle is to compare it to the classical analogue of Grover's search algorithm, which uses a state marking scheme for performing a classical search. The state marking scheme marks the state with the solution with a phase. The Boolean and Phase oracles are equivalent, and the Boolean oracle helps show the superiority of the quantum version of classical algorithms.

6.2.1. The Boolean oracle

An example of Grover's algorithm on 3 qubits (2 data qubits, q_0 and q_1 , defining the search space and 1 ancillary qubit, q_a) using the Boolean oracle is shown in figure 6.2. The oracle is solution specific. This particular

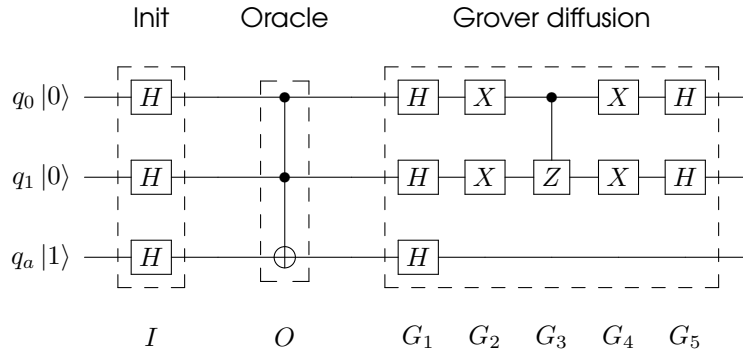


Figure 6.2: Grover's algorithm using a Boolean oracle for a 2 qubit search space with solution $|00\rangle$

oracle tags the state $|00\rangle$ as the solution. After the *Init* stage the quantum state of the circuit is given by the state $|\psi\rangle_I$ which contains 3 qubits,

$$|\psi\rangle_I = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \cdot \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \cdot \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle). \quad (6.9)$$

$$|\psi\rangle_I = \frac{1}{2\sqrt{2}}(|000\rangle - |001\rangle + |010\rangle - |011\rangle + |100\rangle - |101\rangle + |110\rangle - |111\rangle). \quad (6.10)$$

This is a superposition of all possible state for 3 qubits. The Hadamard gate creates a superposition for one qubit. By applying it to all qubits, a

superposition of 3 qubits is created. After the Toffoli gate in the *Oracle* the state of the circuit is,

$$|\psi\rangle_O = \frac{1}{2\sqrt{2}}(|000\rangle - |001\rangle + |010\rangle - |011\rangle + |100\rangle - |101\rangle + |111\rangle - |110\rangle). \quad (6.11)$$

With some reordering, this state can be rewritten as,

$$|\psi\rangle_O = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle - |11\rangle) \cdot \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle). \quad (6.12)$$

This can be confirmed, by expanding out this state and checking the result. The ancillary qubit has been factored out, to observed how the data qubits change. After the *Oracle* the solution $|00\rangle$ has been tagged with a phase, indicated by the negative sign. Since the solution is now tagged with a negative amplitude, an inversion about the mean can be performed. After the first set of Hadamard gates, i.e. G_1 , in the *Grover diffusion* stage the quantum state is transformed from $|\psi\rangle_O$ to

$$|\psi\rangle_{G1} = \frac{1}{2}(|++\rangle + |+-\rangle + |-+\rangle - |--\rangle) |1\rangle. \quad (6.13)$$

Note that the data qubits have been written in the *Hadamard basis* to make it more readable. The next set of gates, i.e. G_2 are the X gates on all qubits except the ancillary qubit. The X gates change the quantum state to

$$|\psi\rangle_{G2} = \frac{1}{2}(|++\rangle - |+-\rangle - |-+\rangle - |--\rangle) |1\rangle. \quad (6.14)$$

After expanding out the $|+\rangle$ and $|-\rangle$ states in the standard basis, the equation above is equal to the following in the standard basis

$$|\psi\rangle_{G2} = \frac{1}{2}(-|00\rangle + |01\rangle + |10\rangle + |11\rangle) |1\rangle. \quad (6.15)$$

After the controlled Z gate, i.e. G_3 , the state is

$$|\psi\rangle_{G3} = \frac{1}{2}(-|00\rangle + |01\rangle + |10\rangle - |11\rangle) |1\rangle. \quad (6.16)$$

The X gates in G_4 transform the state to

$$|\psi\rangle_{G4} = \frac{1}{2}(-|11\rangle + |10\rangle + |01\rangle - |00\rangle) |1\rangle. \quad (6.17)$$

In the Hadamard basis this can be written as

$$|\psi\rangle_{G4} = \frac{1}{2}(-|1\rangle(|1\rangle - |0\rangle) + |0\rangle(|1\rangle - |0\rangle) |1\rangle) = -|-\rangle|-\rangle |1\rangle. \quad (6.18)$$

After the final set of Hadamard gates the state is

$$|\psi\rangle_{G5} = -|111\rangle. \quad (6.19)$$

The solution is the first 2 qubits as the third one is the ancillary qubit,

$$|\psi\rangle_{G5} = -|11\rangle_{q_0 q_1} \otimes |1\rangle_{q_a}. \quad (6.20)$$

This is the solution that the oracle tagged. Therefore the solution has been found, along with a global phase (the negative sign before the state $|11\rangle$) that can be ignored.

6.2.2. The Phase Oracle

The Boolean oracle requires more resources and an extra qubit. The Phase oracle does not. The same 2 qubit example is shown figure 6.3.

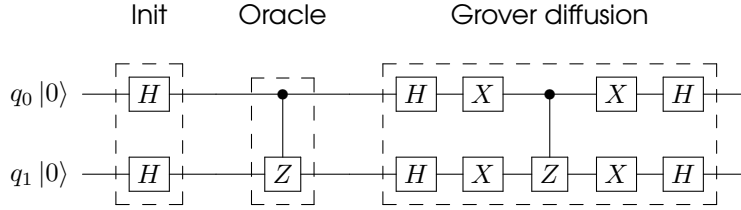


Figure 6.3: Grover's algorithm using a Phase oracle for a 2 qubit search space with solution $|00\rangle$

The state after the *Init* stage is

$$|\psi\rangle_I = |++\rangle. \quad (6.21)$$

The controlled Z gate in the *Oracle* changes the quantum state to

$$|\psi\rangle_O = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle - |11\rangle). \quad (6.22)$$

The solution is now tagged with a phase. The rest of the circuit is completely identical to the one in figure 6.2, as the ancillary qubit is separable.

Grover's algorithm is probabilistic, as is all of quantum computing. Therefore getting the correct solution is not always possible with perfect certainty. The success probability can be improved, by performing part of the circuit multiple times. The oracle and the Grover diffusion operation are performed $\lfloor \frac{\pi}{4}\sqrt{N} \rfloor$ times, where $N = 2^n$ is the number of states. This is where the speed-up comes from.

Bibliography

- [1] P. W. Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer," *SIAM Review*, vol. 41, no. 2, pp. 303–332, Jan. 1999. doi: 10.1137/S0036144598347011.
- [2] P. O'Malley, R. Babbush, I. Kivlichan, *et al.*, "Scalable quantum simulation of molecular energies," *Physical Review X*, vol. 6, no. 3, Jul. 2016. doi: <https://doi.org/10.1103/physrevx.6.031007>.
- [3] H. Neven, *Meet willow, our state-of-the-art quantum chip*, Dec. 2024. [Online]. Available: <https://blog.google/technology/research/google-willow-quantum-chip/>.