**A picture containing logo

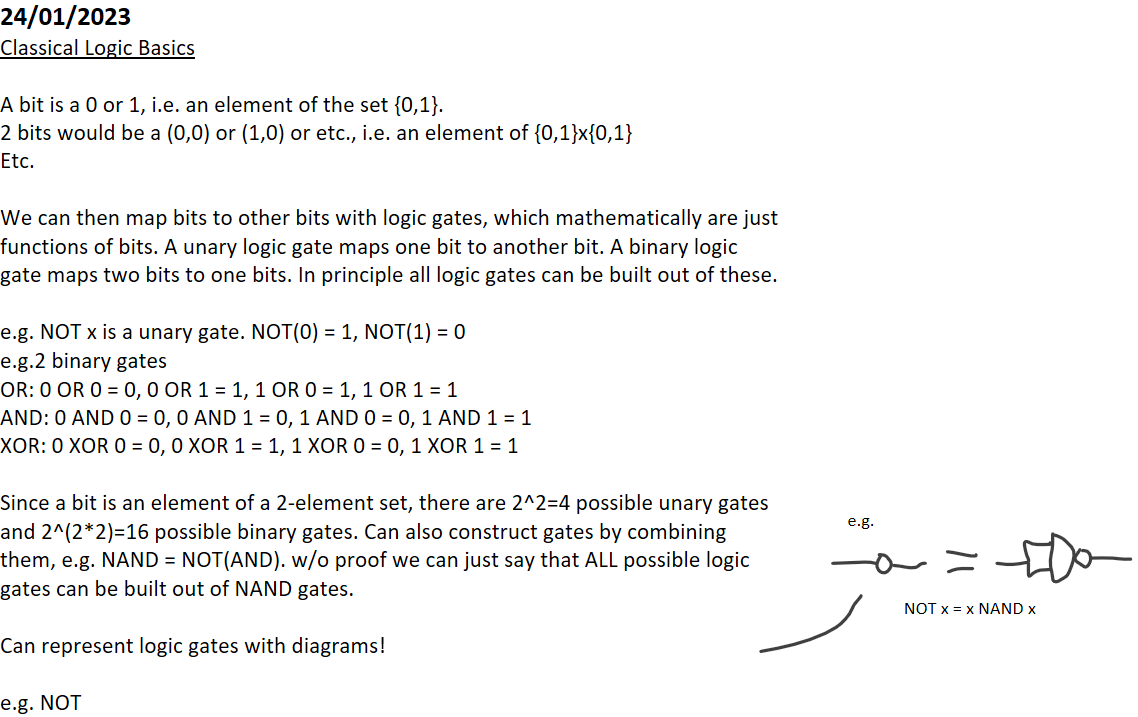
Description automatically generated**

Hello world! (17/01/2023)

**A picture containing graphical user interface

Description automatically generated**

**Week 12**



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | e.g. NAND = NOT(AND) | | | | |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  | |  |  |  |  |  |  |  |  |
|  |  |  |  |  | |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  | | | |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  | Figure 1: Examples of classical NOT, AND and NAND gates.  A NAND gate can be created by combining a NOT and NAND gate | | | | | | | |
|  |  |  |  |

These diagrams can be used to describe logic circuits! On a mathematical level a circuit is a set of rules for applying logic functions in a certain order.

Quantum Logic  
In quantum computing, the basis states are |0> and |1> and then a qubit can be some linear superposition of these w/ unit magnitude. In other words, a qubit isn't an element of a discrete set but instead an element of a 2 dimensional Hilbert Space. I'll label this space H for convenience.

If we have two qubits, then they live in the tensor product space H⊗H. So the basis states are:

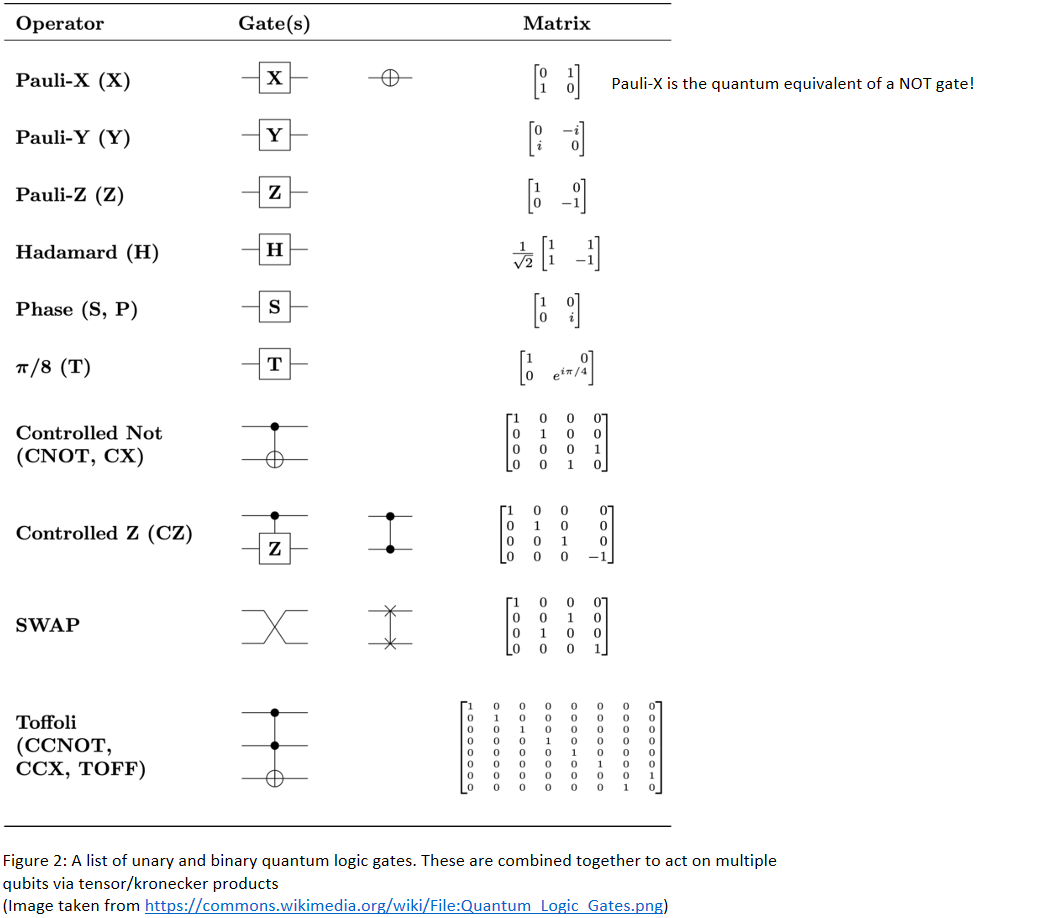
|0>⊗|0> = |00>, |0>⊗|1> = |01>, |1>⊗|0> = |10>, |1>⊗|1> = |11>

Since this is a tensor product space, the possible qubit states may or may not be decomposable into tensor products |A>⊗|B>. If we can write this, then the state is separable and the two qubits are not correlated (i.e. they are statistically independent). If we can't separate the states into a tensor product, then the state is entangled and the two qubits are correlated.

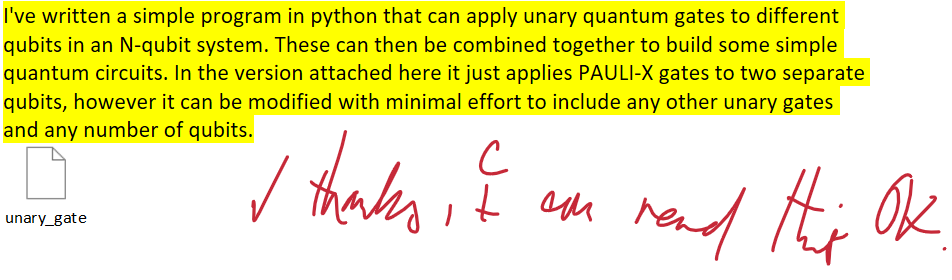
|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  | Represent as column vectors: A|00>+B|01>+C|10>+D|11>= |  |
|  |  |  |

A quantum gate maps qubits to qubits. Since these are now vectors in a Hilbert Space that are linear combinations of |0> and |1> then these must be linear maps --> matrices. Unlike in classical computing, we don't map multiple qubits to a single qubit. Instead we can do more complicated logic on n qubits. We also require these matrices to be unitary, which means all quantum logic is completely reversible. This leads to details such as the no-cloning theorem whereby states can't be duplicated without destroying the original.

**25/01/2023**



Quantum circuit consists of applying matrix calculations to N qubits and then measuring at the end. If there are N qubits then there are 2^N basis states and the matrix has size (2^N)^2



**26/01/2023**

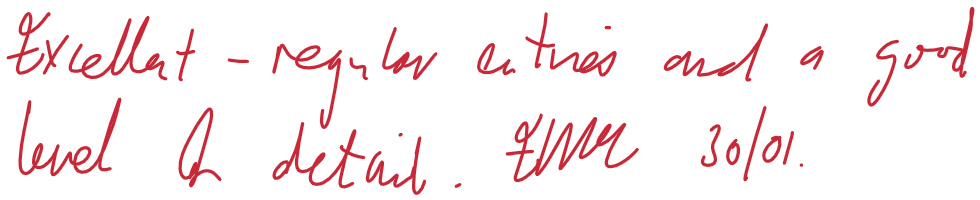
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |
|  | Code explanation: 1) The initial state vector is initialised ("q0"). The program ensures it is normalised and then counts the number of qubits it corresponds to. 2) The unary gate matrices (PAULI-X, PAULI-Y, PAULI-Z) are defined 3) Using a function called "*extend\_unary*", these unary gates can be applied to specific qubits. In this example, the matrices for NOT on just qubit 0 and NOT on just qubit 1 are constructed with this function. 4) The matrices for the two NOT gates are matrix multiplied together to form the overall quantum circuit.  5) The initial state vector q0 is multiplied by the quantum circuit matrix |  |  |  |  |  |  |
|  |  |  | | |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  | |  |
|  |  |  |  |  |  |  |
|  |  |  |  | | | |
|  |  |  |  |

It would be more computationally efficient in this specific example to just take the tensor product of two NOT gates and not use *extend\_unary*, however this code is designed to handle any quantum circuit of unary gates. A better improvement is to allow the *extend\_unary* function to accept lists of gates and the qubits they're applied to, instead of only accepting a single unary gate and a single target qubit. This should be an easy addition.

Can represent quantum circuits using diagrams w/ the symbols in figure 2. Gates applied simultaneously are combined via tensor product, gates applied sequentially are combined via matrix product

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  | corresponds to the matrix *CCNOT\*(X⊗H⊗I*)  Where X is the Pauli-X gate, H is the Hadamard Gate and  I is the identity matrix. |
|  |  |  |

Figure 3: An example of a quantum circuit diagram. Each horizontal line corresponds to a single qubit (so this diagram overall corresponds to a 3 qubit setup). The first qubit is sent through a Pauli-X (I.e. quantum NOT) gate whilst the second is sent through a Hadamard Gate. Then all three qubits are sent through a CCNOT gate with the first and second qubits as the control bits. This means an initial qubit state vector is first multiplied by the matrix *(X⊗H⊗I*) and then by the matrix *CCNOT*.



**Week 13**

**30/01/2023**

In our meeting today we split up our group into two projects. Me and Ana will be working on implementing Grover's Quantum Search Algorithm onto an exoplanet database to look for certain types of planets. Sam and Sid will be working on a project in quantum cryptography.

One of the datasets we may use for exoplanets is the "The Extrasolar Planets Encyclopaedia" (available @ <http://exoplanet.eu/> [last accessed 30-01-2023]).

**31/01/2023**

All qubit gates are unitary matrices and thus are reversible -> gates like AND and OR from classical computing don't make sense in quantum computing because they're not reversible. This is an important factor for binary qubit gates. I began working to extend two qubit gates to N qubit systems in a similar manner to how I extended unary gates.

def extend\_binary(q=None,gate=None,bits=None):

"""

Extend binary gate to an N qubit state

q = indices of adjacent qubits the gate is applied to. Indexing of qubits starts from ZERO! (tuple of 2 integers)

gate = binary gate to be extended to a multi-qubit state. (2D complex numpy array, size 4\*4)

bits = number of qubits (int)

"""

if q is None:

raise SyntaxError("Input qubit indices not specified")

if gate is None:

raise SyntaxError("No gate specified")

if bits is None:

raise SyntaxError("Number of qubits not specified")

if q[1] != q[0]+1:

raise SyntaxError("Gate must be applied to adjacent qubits")

temp\_gate = np.array(1)

on\_second\_bit = False

for i in range(bits):

if on\_second\_bit:

on\_second\_bit = False

continue

if (i,i+1) == q: # Gates are combined together via tensor/kronecker product

temp\_gate = np.kron(temp\_gate,gate) # Insert the gate into the resulting matrix that acts on desired qubit

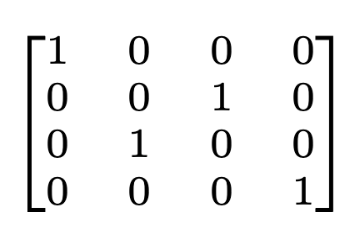
on\_second\_bit = True

else:

temp\_gate = np.kron(temp\_gate,np.identity(2,dtype=complex)) # Insert an identity matrix to act on a different qubit

return temp\_gate

This function can extend a 2-qubit gate to an N qubit state vector and apply it to *adjacent* qubits.  
For example, given the SWAP gate



it can apply the SWAP gate to the 1st & 2nd qubits, the 2nd & 3rd, the 3rd & 4th, etc.  
However, it can't extend the gate to non-adjacent qubits such as the 1st & 3rd.

Grover's Search Algorithm  
  
Grover's Search Algorithm is used to find an element of a database that satisfies a specific condition required by the search. This condition is codified by an operator called a "quantum oracle"

Let x be an arbitrary element of a database and we are searching for a specific element x\_i in the database that satisfies a search requirement. This search requirement can be codified by a function called a quantum oracle which returns a Boolean value

e.g the database is a list of colours and we are searching for the colour red. Then the simplest oracle function is f(x) = 1 for x = red, f(x) = 0 otherwise  
  
The quantum oracle is then the operator given by O|x> = (-1)^(f(x))|x>. Intuitively, each diagonal of the operator corresponds to a possible question that can be "asked" of the oracle, and input state vectors correspond to superpositions of all possible questions asked of the oracle. If the question is the correct one it multiplies the input by -1, otherwise the input remains unchanged.

Applying the oracle to a state vector therefore changes the phase of the state. This is not useful information itself as it does not make it clear which question was correct, since phase isn't measurable. Instead, the output state vector is then passed through a "Grover Diffusion Operator" which converts this phase difference into a magnitude difference.

If we have n classical bits, then these can index 2^n elements of any database. Hence Grover's Search Algorithm requires n =⌈log\_2(N)⌉ qubits.

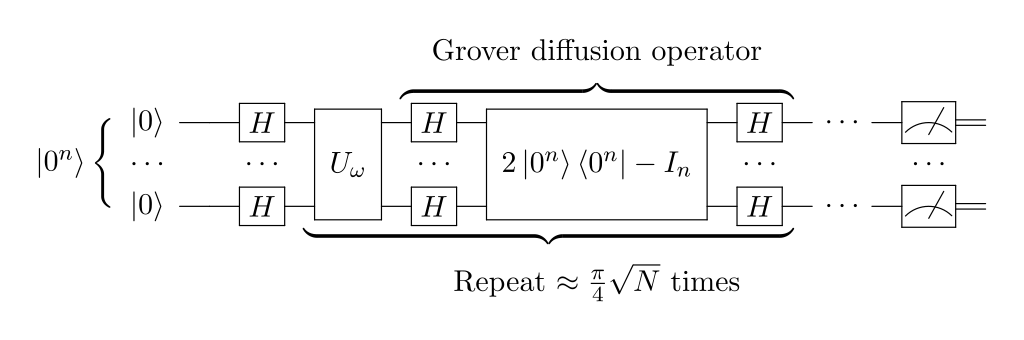


Figure 1: Quantum circuit diagram for Grover's Algorithm. A set of n qubits all set to 0 are sent through Hadamard Gates. They are then passed through the Quantum Oracle and the Grover Diffusion Operators ~√N times. This repeated application will increasingly align the state with the desired target state vector more and more with each iteration. The final output state is then measured.

**01/02/2023**I've looked into IBM's quantum experience (<https://quantum-computing.ibm.com/>). On a free account we can use a maximum of either 5 or 7 qubits for up to 3 hours per program. For both projects this means we could potentially compare our simulation to performance on an actual quantum computer.

7 qubits means we can use a quantum computer to search through a database with a maximum of 128 elements. The extrasolar planets encyclopaedia has over 5000 planets, so this requires taking a small sample of the data.  
  
Applying Grover's Algorithm first requires converting the database into a more usable form. The data is stored in the form of a .csv file, with each entry being of the form

*name, planet\_status, mass, mass\_error\_min, mass\_error\_max, mass\_sini, mass\_sini\_error\_min, mass\_sini\_error\_max, radius, radius\_error\_min, radius\_error\_max, orbital\_period, orbital\_period\_error\_min, orbital\_period\_error\_max, semi\_major\_axis, semi\_major\_axis\_error\_min, semi\_major\_axis\_error\_max, eccentricity, eccentricity\_error\_min, eccentricity\_error\_max, inclination, inclination\_error\_min, inclination\_error\_max, angular\_distance, discovered, updated, omega, omega\_error\_min, omega\_error\_max, tperi, tperi\_error\_min, tperi\_error\_max, tconj, tconj\_error\_min, tconj\_error\_max, tzero\_tr, tzero\_tr\_error\_min, tzero\_tr\_error\_max, tzero\_tr\_sec, tzero\_tr\_sec\_error\_min, tzero\_tr\_sec\_error\_max, lambda\_angle, lambda\_angle\_error\_min, lambda\_angle\_error\_max, impact\_parameter, impact\_parameter\_error\_min, impact\_parameter\_error\_max, tzero\_vr, tzero\_vr\_error\_min, tzero\_vr\_error\_max, k*, *k\_error\_min, k\_error\_max, temp\_calculated, temp\_calculated\_error\_min, temp\_calculated\_error\_max, temp\_measured, hot\_point\_lon, geometric\_albedo, geometric\_albedo\_error\_min, geometric\_albedo\_error\_max, log\_g, publication, detection\_type, mass\_detection\_type, radius\_detection\_type, alternate\_names, molecules, star\_name, ra, dec, mag\_v, mag\_i, mag\_j, mag\_h, mag\_k, star\_distance, star\_distance\_error\_min, star\_distance\_error\_max, star\_metallicity, star\_metallicity\_error\_min, star\_metallicity\_error\_max, star\_mass, star\_mass\_error\_min, star\_mass\_error\_max, star\_radius, star\_radius\_error\_min, star\_radius\_error\_max, star\_sp\_type, star\_age, star\_age\_error\_min, star\_age\_error\_max, star\_teff, star\_teff\_error\_min, star\_teff\_error\_max, star\_detected\_disc, star\_magnetic\_field, star\_alternate\_names*

Not every line has complete entries and most of this data is not actually relevant to the task, so we should trim the data down to include only specific elements of each line (e.g. just the name, mass and radius of each planet). This can be achieved with the following python code:

def trim\_csv\_data(filename=None,wanted\_indices=None):

"""

filename = Name of the .csv file (str)

wanted\_indices = The indices of each entry to include in the trimmed dataset (list)

"""

with open(filename,newline="") as f:

data = csv.reader(f,delimiter=",")

trimmed\_data = []        # Create a new list to store the trimmed lines

for i,line in enumerate(data):

if i == 0:        # Skip the first line containting the labels of each column

continue

trimmed\_line = []

bad\_line = False

for j in wanted\_indices:        # Extract the desired entries from this line

if line[j] == "":        # If there is a desired entry missing from this line, do not include this line

bad\_line=True

break

trimmed\_line.append(line[j])        # Otherwise, add this entry to the trimmed line

if bad\_line:

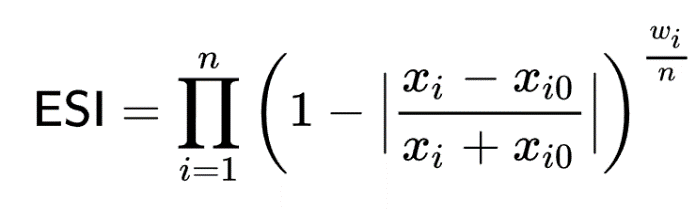
continue

trimmed\_data.append(trimmed\_line)        # Append the trimmed line to the new list

return trimmed\_data

**02/02/2023**  
  
We had another group meeting today to discuss our project plans and prepare our presentation. We now have a fully fleshed out plan for implementing Grover Search. We will be using the Grover Adaptive Search Algorithm as a subroutine to find the most habitable planet in a given dataset. Our implementation will work as follows:  
  
Step 1) Trim down the given database to include only the names, masses, radii and calculated temperatures of each planet

Step 2) Calculate the *Earth Similarity Index* of each planet as outlined in the paper *Schulze-Makuch et al, 2011, "A Two-Tiered Approach to Assessing the Habitability of Exoplanets", Astrobiology 11(10):1041-52, doi:10.1089/ast.2010.0592.* This is a value between 0 and 1 given by the formula:



where x\_i is a numerical property of a planet, x\_i0 is the value of that property for the Earth, w\_i is the the statistical weight of that property and n is the total number of properties.

The relevant properties are:

|  |  |
| --- | --- |
| Property | Weight |
| Radius | 0.57 |
| Density | 1.07 |
| Surface Escape Velocity | 0.70 |
| Surface Temperature | 5.58 |

Step 3) Select a random planet at X\_1 in the register from the database. Its ESI will be referred to as Y\_1

Step 4) Construct an oracle function for a planet at the register x with ESI y as:  
h(x,y) = 1 if y > Y\_1, 0 otherwise  
Use this to define the quantum oracle

Step 5) Perform the Grover search with this oracle for a certain number of iterations. Measure the output, call this X\_2 and the corresponding ESI Y\_2

Step 6) If Y\_2 > Y\_1, define a new quantum oracle with this "baseline ESI" and repeat steps 4 to 6. Otherwise, repeat steps 4 to 6 with the previous X and Y.

Step 7) Repeat steps 4 to 6 a certain number of times and then eventually read the final output.

We can then check how accurate this quantum algorithm is at detecting the most habitable planets in a given dataset for different iteration times and different sample sizes, compare it to a classical search algorithm etc.



**Week 14**

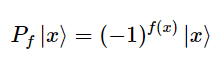
**06/02/2023**  
  
We had our group meeting today. We finalised our presentation and project plan documents for tomorrow and have decided what we will do. This week I will be working on implementing Grover's Algorithm with numpy so that it can be used in Grover Adaptive Search (GAS).

Ana has written a program to extract the relevant data needed to calculate the ESI values from the Extrasolar Planets Encyclopaedia - there was an issue with the units used in the calculations which she will now fix.

To implement Grover search, the most sensible choice would be to define a python class which I'll call *Grover*. Each instance of this class would be instantiated with a specific oracle function and a number of qubits to apply it to. It would then compute its oracle function and diffuser, storing this as class variables. It would then have a class method to compute the outcome of a Grover search for a certain number of iterations.

Quantum Oracle

The definition of the quantum oracle for a given classical oracle function is:



Where f(x) is the oracle function, |x> is a basis qubit state vector (|000>, |001>, |010> etc.) and P is the quantum oracle. In other words, the quantum oracle is a diagonal matrix with 1s for entries where the oracle function returns 0 and -1s where the oracle function returns 1. This can be implemented using the following code:

def compute\_oracle(oracle\_function,bits):

"""

Oracle\_function: Classical oracle function to use (function with single int argument)

Bits: Number of qubits used in the algorithm

"""

quantum\_oracle = np.identity(2\*\*bits)

for i in range(2\*\*bits):

try:        # Use try/except in case the number of bits exceeds original register bitlength

if oracle\_function(i):

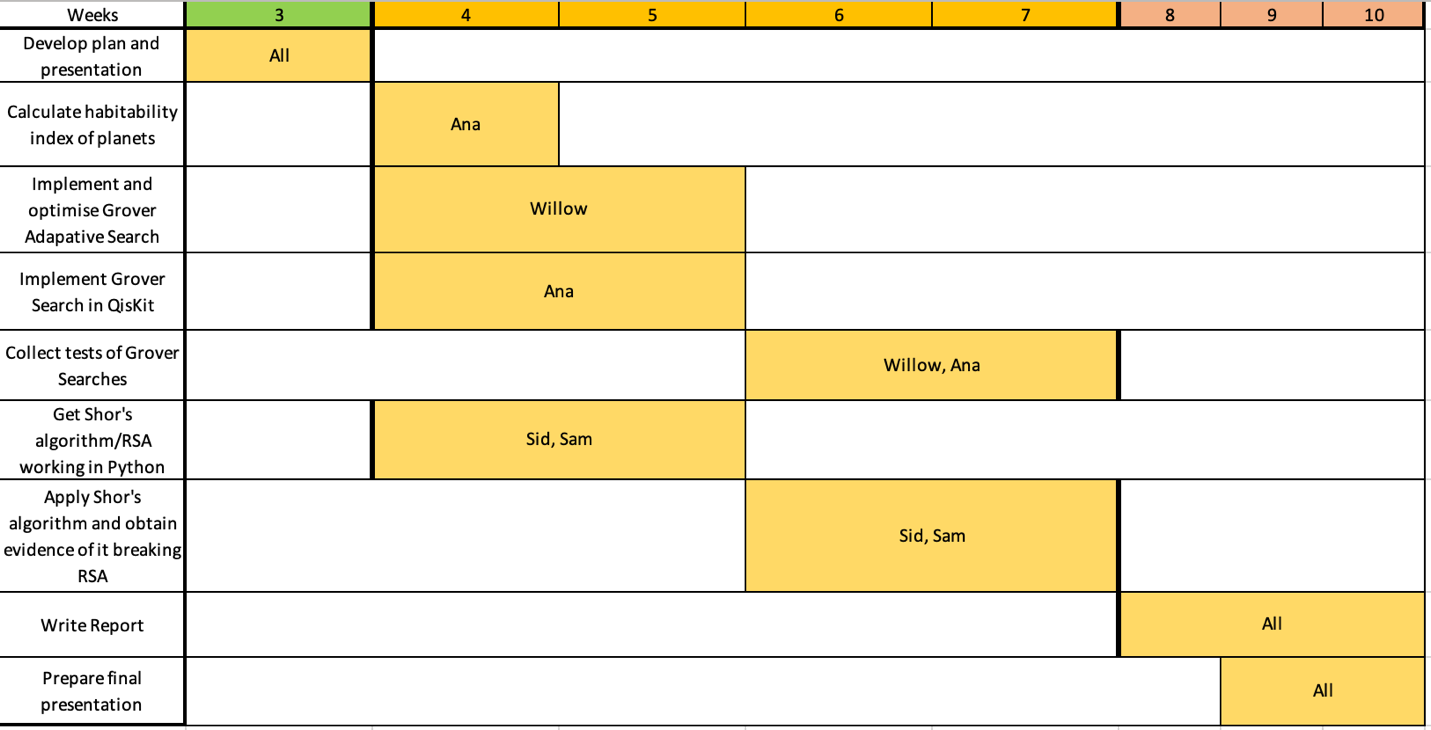
quantum\_oracle[i,i] = -1

except:

continue

return quantum\_oracle

**07/02/2023**  
  
Today we delivered our initial project presentation and project plans. We also have a timeline for what work we need to do outlined in this Gantt Chart:



(Figure 1: The Gantt Chart for our project)

I have written a python class to perform Grover searches given an oracle function, bitnumber and iteration length:

class Grover:

def \_\_init\_\_(self, oracle\_function, bits, verbose=None):

if verbose is None:

self.verbose = False

else:

self.verbose = verbose

self.bitnumber = bits

if self.verbose: print("Computing Hadamard Network...")

hadamard\_gate = 1/(np.sqrt(2))\*np.array([[1,1],[1,-1]],dtype=np.float32)

self.hadamards = extend\_unary(gate=hadamard\_gate,bits=self.bitnumber,verbose=self.verbose)

if self.verbose: print("Done!")

self.diffuser = self.compute\_diffuser()

self.quantum\_oracle = self.compute\_oracle(oracle\_function)

def compute\_diffuser(self):

if self.verbose: print("Computing Diffuser...")

diffuser = np.identity(2\*\*self.bitnumber,dtype=np.float32)

diffuser[0,0] = -1

diffuser = np.matmul(self.hadamards,diffuser)

diffuser = np.matmul(diffuser,self.hadamards)

if self.verbose: print("Done!")

return diffuser

def compute\_oracle(self,oracle\_function):

if self.verbose: print("Computing Quantum Oracle...")

quantum\_oracle = np.identity(2\*\*self.bitnumber,dtype=np.float32)

for i in range(2\*\*self.bitnumber):

try:        # Use try/except in case the number of bits exceeds original register bitlength

if oracle\_function(i):

quantum\_oracle[i,i] = -1

except:

continue

if self.verbose: print("Done!")

return quantum\_oracle

def search(self,iterations):

"""

Performs a Grover Search for a given number of iterations. Returns a numpy array.

Iterations: The number of iterations to compute (int)

"""

target\_state = np.zeros(2\*\*self.bitnumber,dtype=np.float32)

target\_state[0] = 1

target\_state = np.matmul(self.hadamards,target\_state)

circuit = np.identity(2\*\*self.bitnumber,dtype=np.float32)

for i in range(iterations):

circuit = np.matmul(circuit,self.quantum\_oracle)

circuit = np.matmul(circuit,self.diffuser)

if self.verbose: print("Completed {}/{} Grover Iterations...".format(i+1,iterations), end="\r",flush=True)

if self.verbose: print("\nDone!")

target = np.matmul(circuit,target\_state)

return target

I will do some controlled in-depth unit tests to prove this works, for now here is an illustrative graph of a Grover Search applied to a list of 1024 elements. Each element is a random integer between 0 and 800 and it is constructed so that there are *at* least 10 elements equal to 400 (in practice there are usually 9-12). A Grover search to find elements equal to 400 was then applied to the database with 1600 iterations for 1024 shots. The output of each shot of the algorithm was then measured and plotted using matplotlib: Chart, scatter chart

Description automatically generated

(Figure 2: A frequency plot of the measured outcome of each Grover search of 1600 iterations for a database of 1024 elements. By looking at the elements which are observed statistically more frequently it is clear that the program found 11 elements in the database equal to 400.)

After doing some actual formal unit tests with lower iteration counts (1600 is *incredibly* overkill), I could just use this for the project and be done with it. However, it is quite slow to run since it is an entirely linear algorithm (technically numpy does the array calculations in parallel, but this is still quite limited by the capacity of the CPU). Quantum computers are massively parallel machines, so it would be more suitable to run the calculations as much in parallel as possible.  
  
To do this, I will rewrite the code to use a python module that allows for the use of the CUDA Toolkit. This means the matrix calculations can be run on a GPU instead of the CPU. GPUs are designed to do large matrix calculations very quickly, so this should massively boost performance.

(My laptop has a built-in Nvidia RTX 3060 running at the full power rating of the card of 115W, with the ability to boost up to 130W).  
  
  
**08/02/2023**

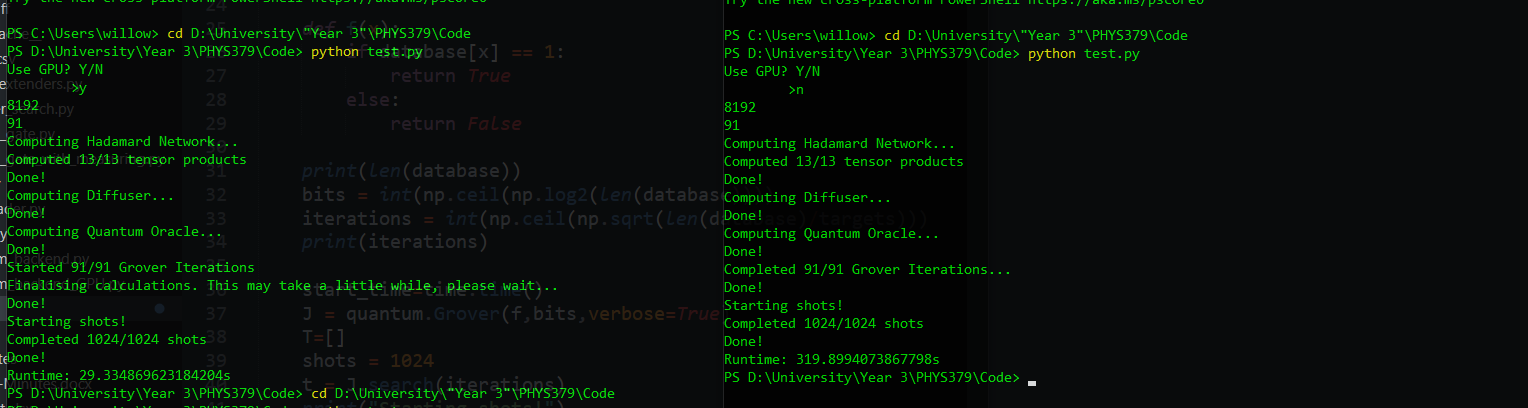
I have now implemented the use of CUDA in my program to perform calculations on the GPU by using the CuPy library (<https://cupy.dev/>). This library is designed to work almost identically to NumPy except it offloads calculations to the GPU. This means the code for using CuPy is *almost* identical to the numpy code, except all the numpy arrays used in the calculations are replaced with cupy arrays. There are a few quirks that need to be accounted for though:

1. The default floating point type used for numpy arrays is float32, whereas the default for cupy arrays is float64. Leaving cupy arrays in float64 completely wipes out the performance advantage gained from hardware acceleration on my GPU since it is optimised for float32 (this is also the case for almost all GPUs). Although this does lead to the calculations being more numerically accurate, for the sake of actual practical application I have explicitly constructed all cupy arrays to be in float32.
2. Performing a loop on a numpy array is already somewhat slow, but it is *even slower* on a cupy array since that is not what GPUs are designed to do. This means measuring a state vector is faster on the CPU than the GPU, so the Grover search algorithm is still written to return a numpy array and not a cupy array.
3. The CPU and GPU have separate memory pools. On my laptop there is 16GB DDR4 of RAM available to the CPU and 6GB GDDR6 of VRAM available to the GPU. This means transferring data between the CPU and GPU introduces extra latency, which means the GPU version may actually take longer to run than the CPU version for low qubit counts. Also this means the number of qubits the program can handle will be limited by the VRAM capacity. But this was already a factor for the CPU-bound version as well.

The full python code for the hardware-accelerated version of the "quantum computing backend" is attached here:

<<quantum\_backend\_GPU.py>>

The speedup this gives seems to be consistently around 10 times faster on the GPU for 12 or more qubits. For example, running a grover search for a single element on a database spanned by 13 qubits took ~29.3s in one test, whereas on the CPU it took ~319.9s.



(Figure 3: Command line output of the two programs. On the final line of each program is the overall runtime required to compute 1024 shots. On the left is the GPU version which took ~29.3s to run, on the right is the CPU version which took ~319.9s)

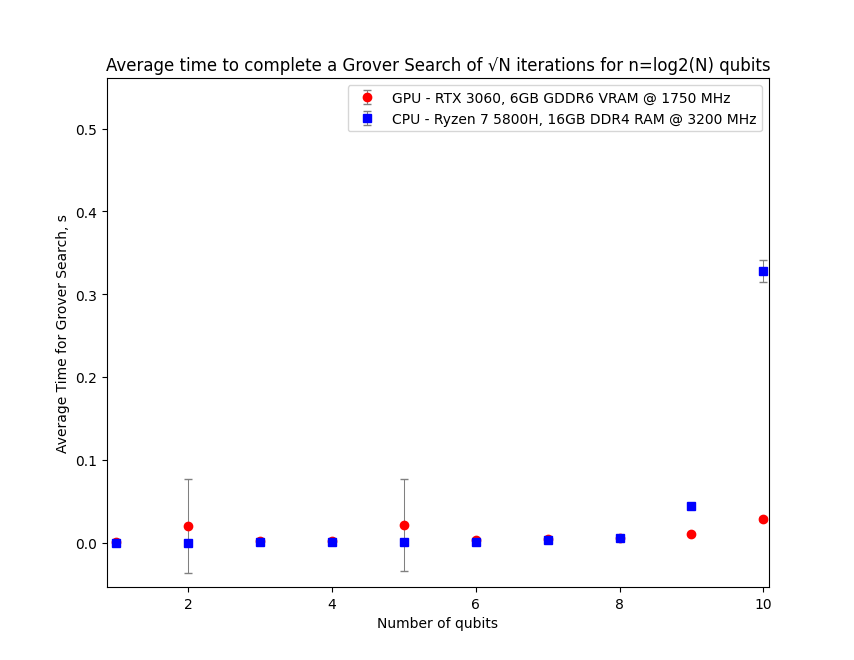
The GPU version can run calculations for up to 14 qubits on my computer - any higher and the memory required to store all the matrices exceeds the VRAM capacity (My rough estimate is that 15 qubits would require ~18GB of VRAM overall, which is only available on high-end GPUs). However, I believe I can fix this issue by changing how the matrices are handled. At present, they are stored as numpy/cupy arrays as dense matrices. By changing them into *sparse* matrices the VRAM used should go down greatly.  
  
**09/02/2023**

I have attempted to convert the matrices into sparse matrices using scipy's sparse matrix library and cupy's variant of this. I had success with converting the diffuser and quantum oracle into sparse matrices, however they do not conform to the same sparse matrix types. The quantum oracle makes sense to be stored as a diagonal sparse matrix, whereas for the diffuser it makes more sense to be stored as a csc sparse matrix. I did not have any success converting the network of Hadamard gates into a sparse matrix.

As a result of all this, the program actually ends up using *more* VRAM than before since the Hadamard network takes up so much space compared to everything else. Since this isn't working I will not implement sparse matrices and instead implement the adaptive algorithm with what I already have. This isn't a problem since we only need 13 qubits to index the *entire* Extrasolar Planets Encyclopaedia and only 10 to index the ESI dataset.

**10/02/2023**

I have written a program to directly compare the speed of the CPU and GPU versions of my code. The program creates 12 lists of lengths 2^1, 2^2, ..., 2^12 which are hence indexed by 1-14 qubits (this can be varied with a slight modification to handle up to 14 qubits). Each element of the list is an integer equal to 0, except for the first element which is equal to 1. The program then executes Grover searches on each list 10 times to find the element equal to 1, using both CPU and GPU methods. This number can be varied with a slight modification to the code. It then calculates the average time to complete each Grover search and the standard deviations. It then plots the results. The code is attached here:

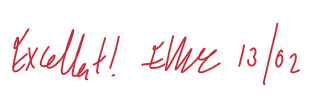
<<test2.py>>

A picture containing chart

Description automatically generated

(Figure 4: The full graph for 12 qubits. On the right is a zoomed in view of the graph for 10 qubits.)

From figure 4 we can see that for qubit counts below 9 the GPU version is almost identical in performance to the CPU version. However, at 9 and above qubits the CPU version becomes exponentially slower than the GPU version and thus the benefit of hardware-acceleration quickly becomes enormous.



**Week 15**

**13/02/2023**

Today I started working on incorporating Grover's algorithm into the full Grover Adaptive Search (GAS) algorithm. It will be easier to first write this for a general database with an arbitrary adaptive oracle function and then apply it to the ESI database problem.

The basis of the adaptive search algorithm to implement will be the one outlined in Boyer et al*, Tight Bounds on Quantum Searching* (1996) (<https://arxiv.org/abs/quant-ph/9605034>)  
Baritompa et al gives the following summary for the algorithm in GROVER’S QUANTUM ALGORITHM APPLIED TO GLOBAL OPTIMIZATION:

*1. Initialise m = 1.*

*2. Choose a value for the parameter λ.*

*3. Repeat:*

*(a) Choose an integer j uniformly at random such that 0 ≤ j<m.*

*(b) Apply Grover’s algorithm with j rotations, giving outcome i.*

*(c) If i is a target point, terminate.*

*(d) Set m = λm.*

*(Boyer's GAS Algorithm)*

This algorithm can then be generalised further into that described by Durr and Hoyer in *A Quantum Algorithm for Finding the Minimum* (1996) (<https://arxiv.org/abs/quant-ph/9607014>). This is described in Baritompa et al in a corrected format (the original version contained errors) as follows:

1. *Generate X1 uniformly in S, and set Y1 = f(X1).*
2. *Set m = 1.*
3. *Choose a value for the parameter λ (as in the previous algorithm).*
4. *For n = 1, 2,... until a termination condition is met, do:*

*(a) Choose a random rotation count rn uniformly distributed on {0,...,ceiling(m − 1)}.*

*(b) Perform a Grover search of rn rotations on f with threshold Yn, and denote the outputs by x and y.*

*(c) If y<Yn, set Xn+1 = x, Yn+1 = y, and m = 1; otherwise, set Xn+1 = Xn, Yn+1 = Yn, and m = min(λm,sqrt(N))*

*(Durr-Hoyer Algorithm)*

(By threshold Yn, this refers to having an oracle function which returns 1 if y<Yn for every element y in the database f. x and Xn refer to registers of elements in the database)

Baritompa et al generally makes the recommendation of choosing λ=1.34. For now I'll choose this but it will probably be useful to vary it and compare how effective the algorithm is.

The current termination condition I am using is to terminate if the program does not return a smaller value after 5 attempts. Using this condition, I have written a program to implement the Durr-Hoyer algorithm that creates a database with 1024 integers between 1 and 800 and then sets the first element equal to 0 (I'm picking this element every time for consistency). It then searches through the database to find the element equal to 0.

import numpy as np

import random

import quantum\_backend\_GPU as quantum

def get\_database():

database = []

length = 2\*\*10

for i in range(length):

database.append(random.randint(1,800))

database[0]=0

return database

def adaptive\_oracle(x,x\_0,database):

Y = database[x\_0]

if database[x] < Y:

return True

else:

return False

def main():

database = get\_database()

bits = int(np.ceil(np.log2(len(database))))

x\_0 = random.randint(0,len(database)-1)

scaling = 1.34

m = 1

fails = 0

while fails < 5:

iterations = random.randint(1,np.ceil(m))

J = quantum.Grover(lambda x: adaptive\_oracle(x,x\_0,database),bits)

q = J.search(iterations)

x\_1 = quantum.measure(q)

print("x0:",x\_0,database[x\_0],"x1:",x\_1,database[x\_1])

print(adaptive\_oracle(x\_1,x\_0,database))

if adaptive\_oracle(x\_1,x\_0,database):

x\_0 = x\_1

fails = 0

else:

fails += 1

m = min(scaling\*m,np.sqrt(2\*\*bits))

print(x\_0)

print(database[x\_0])

if \_\_name\_\_=="\_\_main\_\_":

main()

**14/02/2023**

I have extended my code so that it can run multiple consecutive shots of the Durr-Hoyer algorithm and then plot the results. Using the criteria listed previously for the test database (first element equal to zero so that it is the global minimum targeted by the algorithm, then the rest random integers between 1 and 800 for 1024 total elements) and the chosen restrictions on the algorithm (terminate after landing on Xn+1=Xn 5 times in a row, set λ=1.34) I have tested the algorithm for 100 shots. I repeated this "100 shot test" ten times and then plotted the average results.

Graphical user interface

Description automatically generated

(Figure 1: A frequency plot of the output of the Durr-Hoyer algorithm for 100 shots, averaged over 10 trials. The minimum element in the database was at the start of the database and was the target to find. The frequency at which this element is the output divided by the total number of shots gives the success rate. This means the *fail* rate for the Durr-Hoyer algorithm in this case was (29.0±3.55)%, which is unacceptably high. Thus the chosen condition for terminating each shot of the algorithm is insufficient in ensuring the accuracy of the search)

It is evident from the high failure rate seen in figure 1 that a more sophisticated termination condition is required. I have also realised that my implementation of the algorithm is slightly incorrect as well. In step 4c) it states that m should be set to 1 if a element with y<Yn is found. However, when I made this modification to the code the fail rate grew dramatically to around 90%, so this element of the algorithm will *not* be included. This is due to the fact the algorithm assumes the target set in the database is large and so was included to reduce computational complexity, however in our case the target set is very small.

I'll call the required number of consecutive failures to find a smaller value of Yn the "termination threshold" ***µ*** (so the previous work done so far had µ=5).

Meetings

We then had our meeting with Ed which was followed up our own internal meeting. Sam will send his qiskit code to Ana so it can be run on her machine. Sid has got RSA working. Once Sam has QFT working he can incorporate it directly into Shor's algorithm and then hopefully that should work. Ana has implemented Grover's algorithm in Qiskit and will now test it, incorporate it into a class and also start some work on our report.

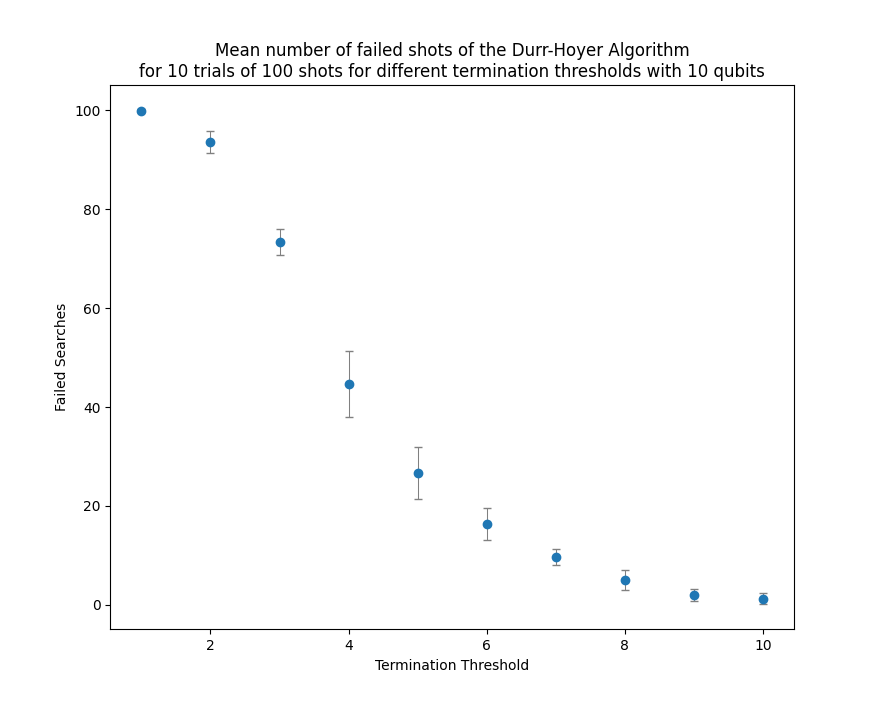
For the Durr-Hoyer implementation, we will be taking a different approach to terminating the search. Instead of attempting to find the planet with the global maximum ESI in a given dataset, we will instead search for a planet with an ESI greater than a given minimum. This means the search should be terminated when a planet with a desired ESI is found. This is a more sensible approach since it is the one Durr-Hoyer is designed for and it is trivial to implement. That said, it will still be useful to implement the global maximum searcher with the termination threshold method as well.

*On another aside I have actually thought about the fact that what I have written so far searches for the minimum value entry. Since we are looking for the maximum value entry in the ESI database I have clearly made a mistake here. Luckily this is an easy fix (just need to flip the order of the inequality condition in the oracle function).*

**15/02/2023**

Returning to the µ discussion, I wrote a program to test different values of µ. The ESI database contains 745 elements and thus requires 10 qubits to search through, so I tested µ for a 10 qubit search. This was done like before by running the Durr-Hoyer algorithm on the previously discussed randomised 1024 element dataset to find the minimal value for 100 shots (averaged over 10 trials) and then calculating the resulting failure rate for µ between 1 and 10. This means a total of 10,000 individual runs of the Durr-Hoyer algorithm were run (and hence a random number of Grover Searches, at least 45,000 at minimum overall).

(45,000 minimum as µ variation means total minimum Grover searches is 1000+2000+3000+...+10,000=45,000)



|  |  |  |
| --- | --- | --- |
| µ (10 qubits) | Mean Fail Rate (%) | Mean Fail Rate Uncertainty (%) |
| 1 | 99.90 | 0.30 |
| 2 | 93.60 | 2.29 |
| 3 | 73.40 | 2.58 |
| 4 | 44.60 | 6.70 |
| 5 | 26.60 | 5.28 |
| 6 | 16.40 | 3.23 |
| 7 | 9.60 | 1.62 |
| 8 | 5.00 | 2.00 |
| 9 | 1.90 | 1.22 |
| 10 | 1.20 | 1.17 |

 (Figure 2: On the left is the full table of fail rates for different values of µ. On the right is a scatter plot of the data. Assuming a maximum tolerance of 5% failure, it is clear that the ideal threshold for a 10 qubit search is µ=9. µ=8 would also work pretty well, however its uncertainty means it can go above 5% failure. It does have the benefit of running faster than µ=9 though. Overall it is clear that there is very little advantage in going beyond µ=9 in this case)

**16/02/2023**

The variation of the Grover search algorithm which can be run on an actual quantum computer via the IBM Quantum Experience can only use up to 7 qubits. This means we also needed to find an appropriate value of µ for a 7 qubit search. I repeated the test for 7 qubits with the exact same method, except in this case the database had 128 elements instead of 1024.

Chart, scatter chart

Description automatically generated

|  |  |  |
| --- | --- | --- |
| µ (7 qubits) | Mean Fail Rate (%) | Mean Fail Rate Uncertainty (%) |
| 1 | 96.40 | 1.36 |
| 2 | 80.10 | 2.74 |
| 3 | 53.30 | 6.39 |
| 4 | 28.10 | 3.62 |
| 5 | 15.50 | 4.59 |
| 6 | 10.00 | 3.03 |
| 7 | 4.80 | 2.09 |
| 8 | 1.70 | 1.10 |
| 9 | 1.10 | 1.04 |
| 10 | 0.30 | 0.64 |

(Figure 3: Left - Fail rates for different values of µ for 7 qubits. Right - A graph of the data. Assuming a maximum tolerance of 5% failure, a threshold of µ = 7 or 8 is ideal (like before though, 7 is just on the edge of acceptable since its uncertainty means it can go above 5%). There is very little benefit in going beyond µ=8.)

**18/02/2023**

I have now modified my code so that it imports the ESI database, adjusts the length of the list so that it can be used with Grover search and then searches through the database to find the planet with the maximal ESI. This was done with the following code:

def import\_ESI\_data():

with open("esi.csv",newline="") as file:

data\_reader = csv.reader(file,delimiter=",")

data = [row for row in data\_reader]

del data[0]                # Remove Names/ESI line at the start of the file

data\_length = len(data)

bits = int(np.ceil(np.log2(data\_length)))

entries = 2\*\*bits

for i in range(entries-data\_length):        # Ensure the database length is a power of 2

data.append(None)

return data

The full python code is attached here, along with the ESI database:

<<grover\_adaptive.py>>

<<esi.csv>>

The required number of qubits to search through this database is 10 qubits. Using a threshold of µ=9, the Durr-Hoyer algorithm was used to search for the entry with the maximum ESI in the database for 100 shots. This planet is TRAPPIST 1d with an ESI of *0.873098354375771.* It is located at index 636 in the database (assuming the column labelling entry at the beginning is removed and that indexing starts from 0. If they are not removed and indexing starts from 1 it is at index 638).

A picture containing chart

Description automatically generated

(Figure 4: The output of the search program w/ µ=9 when applied to the ESI database with the task of finding the planet with the highest ESI. The mode output was the register x=636, which is the correct position and is the location of TRAPPIST 1d in the database. In this instance there were 4 out of 100 shots which gave an incorrect output)

In essence, this is the main aim of this half of the project now "done." However, we now need to actually test this extensively to see how well it works and also extend it to work with the qiskit version so that it can run on a quantum computer.



**Week 16**

**20/02/2023**

Ana has written Grover search in qiskit. I adapted her code so that it uses a class that is designed to have similar syntax to what I have already written in numpy/cupy.

class Grover:

def \_\_init\_\_(self,oracle\_function,bits,verbose=None):

if verbose is None:

self.verbose = False

else:

self.verbose = verbose

self.bitnumber = bits

self.circuit = QuantumCircuit(self.bitnumber)

self.reflector = self.build\_reflector()

self.quantum\_oracle = self.build\_oracle(oracle\_function)

def build\_reflector(self):

if self.verbose: print("Computing Diffuser...")

reflector\_matrix = -1\*np.identity(2\*\*self.bitnumber)

reflector\_matrix[0,0] = 1

reflector = qi.Operator(reflector\_matrix)

if self.verbose: print("Done!")

return reflector

def build\_oracle(self,oracle\_function):

if self.verbose: print("Computing Quantum Oracle...")

oracle\_matrix = np.identity(2\*\*self.bitnumber)

for i in range(2\*\*self.bitnumber):

try:        # Use try/except in case the number of bits exceeds original register bitlength

if oracle\_function(i):

oracle\_matrix[i,i] = -1

except:

continue

quantum\_oracle = qi.Operator(oracle\_matrix)

if self.verbose: print("Done!")

return quantum\_oracle

@staticmethod

def hadamards(qcircuit,bits):

"""

qcircuit = qiskit circuit to apply hadamard gates to

bits = number of bits in the circuit

"""

for i in range(bits):

qcircuit.h(i)

return qcircuit

@staticmethod

def diffusion(reflector,circuit,bits):

circuit = Grover.hadamards(circuit,bits)

circuit.unitary(reflector,range(bits),label='J')

circuit = Grover.hadamards(circuit,bits)

return circuit

def search(self,iterations):

"""

Performs a Grover Search for a given number of iterations. Returns a numpy array.

Iterations: The number of iterations to compute (int)

"""

self.circuit = self.hadamards(self.circuit, self.bitnumber)

for i in range(iterations):

self.circuit.unitary(self.quantum\_oracle, range(self.bitnumber), label='ORACLE')

self.circuit = self.diffusion(self.reflector,self.circuit,self.bitnumber)

if self.verbose: print("Started {}/{} Grover Iterations".format(i+1,iterations),end="\r",flush=True)

self.circuit.measure\_all()

if self.verbose: print("\nDone!")

I then made use of the qiskit runtime library so that the resulting circuit can be sent off to IBM's quantum experience service to run on an actual quantum computer. The full python code is attached here (although it is slightly bugged at the moment):

<<quantum\_backend\_qiskit.py>>

Roughly speaking, what it does is create a list of length 2^5 with every element equal to 0 except for one random element equal to 1. It then runs a 5 qubit Grover search with the appropriate oracle to find this element. In this version, it runs the program on the *IBMQ-Manila* quantum computer for 1024 shots.

In practice, this didn't work very well. The queue time for the program was *50 mins, 33.8 s* and it then took *27 s* to run, which *is 2 to 4 orders of magnitude slower than the simulation I already built.* Furthermore, it failed to actually find the target element! Either due to noise and errors in the circuit or (more likely) issues with constructing the quantum oracle, there was no combination of bits which was measured at a statistically higher rate than any others (i.e. it failed to project the qubit state onto the target as it was supposed to).

Chart, histogram, waterfall chart

Description automatically generated

(Figure 1: The quasi-probability of each bit combination being measured after the application of Grover's algorithm for 5 qubits over 1024 shots. As can be seen, there is no bit combination with a statistically significant probability of being measured and thus the program has failed to find the target)

Full data here:

<<20230220-IBMQ-Manila-Data.csv>>

Furthermore, although I adapted Ana's code with the intention of then using it in Grover Adaptive Search, it appears this may not be possible. IBM's service only allows you to upload a wholly quantum circuit to their service, not a program that uses multiple different quantum circuits which it then adaptively modifies as seen in GAS.

**21/02/2023**

Held our Group Meeting Today  
After our meeting today, we have decided we will not attempt to implement our code on the IBM quantum computers as it is unlikely we can get this working before the deadline. Instead, this week I will work on incorporating simulating errors into the program.

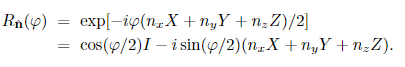
**22/02/2023**

Today I started working on implementing error simulation into the program. There are two obvious methods for doing this:

1. When computing the matrices used in the circuit, occasionally mess up gates at random by changing elements.
2. At random intervals, insert a random number of randomised unitary matrices to act on random qubits.

The second method is significantly better than the first as it scales much more easily, it is much more general and it explicitly preserves the unitarity requirement of any quantum mechanical process.

A general 2D unitary matrix that acts on a single qubit is given by



where I is the idenity matrix, X,Y,Z are the pauli matrices, phi is some angle, and nx,ny,nz are the components of some normalised real vector (from the PHYS483 notes and also PHYS366). By randomly choosing the values of phi, nx, ny and nz we can construct any random unary gate and then extend it to multiple qubits via tensor products. This random unary gate will represent some "error" in the system.

I have implemented the construction of these randomised error "gates" using the following code:

def get\_error\_matrix(bits,errorp):

# Define generators of U(2) and the identity matrix

X = cp.array([[0,1],[1,0]])

Y = cp.array([[0,-1j],[1j,0]])

Z = cp.array([[1,0],[0,-1]])

I = cp.array([[1,0],[0,1]])

# Create a list of targets to apply random error "gates" to

targets = [[random.randint(0,bits-1)]]

for i in range(bits):

if i in targets:

continue

elif random.random() <= errorp:

targets.append([i])

matrices = []

for target in targets:

n\_vec = cp.random.rand(3)        # Create randomised axis vector for the gate

for i,component in enumerate(n\_vec):

n\_vec[i] = component\*((-1)\*\*random.randint(0,1))        # Flip sign of components at random

n\_vec = n\_vec/cp.linalg.norm(n\_vec)                # Ensure the axis vector is normalised

angle = (np.pi/8)\*random.random()        # Pick a random angle between 0 and pi/8

matrix = cp.cos(angle/2)\*I-1j\*np.sin(angle/2)\*(n\_vec[0]\*X+n\_vec[1]\*Y+n\_vec[2]\*Z)        # Construct the gate

extended\_matrix = extend\_unary(targets=target,gate=matrix,bits=bits)        # Extend the gate to the multi-qubit setup

matrices.append(extended\_matrix)

# Multiply all the error "gates" together to construct the overall error "gate"

error\_matrix = cp.identity(2\*\*bits,dtype=cp.float32)

for matrix in matrices:

error\_matrix = cp.matmul(error\_matrix,matrix)

return error\_matrix

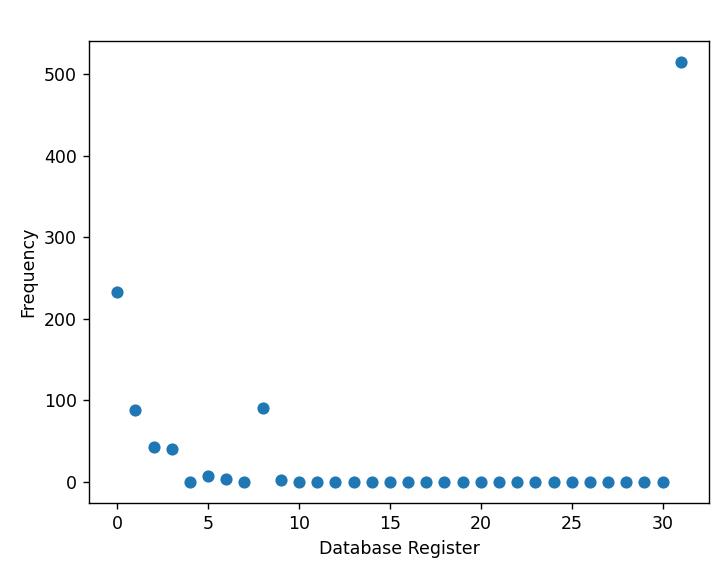
(This is the GPU version. For the CPU version all references to the cupy module labelled as cp must be replaced with references to numpy)

**23/02/2023**

Using the above code I have implemented the error gates into the grover search circuit. Error "gates" are randomly generated in three different areas of the circuit:

* Before the quantum oracle
* Between the quantum oracle and the diffusion operator
* After the diffusion operator

(This corresponds to every possible location they can be inserted)  
I have also introduced a second argument into the Grover.search() function called errorp. Errorp should be a value between 0 and 1 and represents the probability that any given qubit will experience an error (it also shows up in the function get\_error\_matrix above). If it is not specified then the program simply does not simulate any error.



(Figure 2: An example worst case scenario for Grover search. In this instance, the algorithm is searching through a 32 element list for an element located at the very beginning of the list for 1024 shots. However, errorp=1 in this case (i.e. the maximum possible error is implemented). As a result the algorithm fails vastly more often than it succeeds.)

Updated quantum backend code:

<<quantum\_backend\_GPU.py>>



**Week 17**

**28/02/2023**

Held both our meeting with Ed and our group meeting today  
  
This week we will focus on testing our projects. The Shor's Algorithm/RSA Decryption project is supposedly now in a finalised working state.

Ana will focus on testing error variation in Grover's algorithm. I first need to work on fixing the extend\_binary function for Shor's algorithm to work properly and then I will focus on testing Grover adaptive search. I will also try to incorporate error into Shor's algorithm if possible in a similar manner to how I have for Grover's algorithm.

The issue with extend\_binary is that when I wrote it, it could only apply binary quantum gates to adjacent qubits. Sam had previously attempted to implement a fix for this, but it didn't work. I have written a fix for this that works by repeatedly applying SWAP gates so the target qubits are adjacent in a manner similar to bubble sort, applying the binary gate to them and then applying more swap gates so they are back in their original positions. This is admittedly not the most efficient way of doing it, but it works consistently.

This is achieved with the following code:

def extend\_adjacent\_binary(q=None,gate=None,bits=None):

"""

Extend binary gate to an N qubit state

q = indices of adjacent qubits the gate is applied to. Indexing of qubits starts from ZERO! (int)

gate = binary gate to be extended to a multi-qubit state. (2D complex numpy array, size 4\*4)

bits = number of qubits (int)

"""

if q is None:

raise SyntaxError("Input qubit indices not specified")

if gate is None:

raise SyntaxError("No gate specified")

if bits is None:

raise SyntaxError("Number of qubits not specified")

if q[1] != q[0]+1:

raise SyntaxError("Gate must be applied to ajacent qubits")

temp\_gate = np.array(1)

on\_second\_bit = False

for i in range(bits):

if on\_second\_bit:

on\_second\_bit = False

continue

if (i,i+1) == q: # Gates are combined together via tensor/kronecker product

temp\_gate = np.kron(temp\_gate,gate) # Insert the gate into the resulting matrix that acts on desired qubit

on\_second\_bit = True

else:

temp\_gate = np.kron(temp\_gate,np.identity(2)) # Insert an identity matrix to act on a different qubit

return temp\_gate

def get\_swapper(array,q,bits):

swapper = np.identity(2\*\*bits)

SWAP = np.array([[1,0,0,0],[0,0,1,0],[0,1,0,0],[0,0,0,1]])

for i,element1 in enumerate(array):

if element1 != q[0]:

continue

else:

# loop to compare array elements

for j in range(0, len(array) - i - 1):

if array[j] == q[0]:

continue

elif array[j] == q[1]:

break

else:

temp = array[j]

array[j] = array[j+1]

array[j+1] = temp

extended\_swap = extend\_adjacent\_binary(q=(j-1,j),gate=SWAP,bits=bits)

swapper = np.matmul(extended\_swap,swapper)

break

return swapper

def extend\_binary(q=None,gate=None,bits=None):

swapper1 = get\_swapper([i for i in range(bits)],q,bits)

extended\_gate = extend\_adjacent\_binary(q=(q[1]-1,q[1]),gate=gate,bits=bits)

swapper2 = get\_swapper([i for i in range(bits)][::-1],q[::-1],bits)

final\_gate = np.matmul(extended\_gate,swapper1)

final\_gate = np.matmul(swapper2,final\_gate)

return final\_gate

For example, if we wish to apply a CNOT gate to qubits 1 and 3 in a 3-qubit system, the code returns the 2D numpy array

[[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. 0. 0.]

[0. 0. 0. 0. 0. 0. 0. 1.]

[0. 0. 0. 0. 0. 0. 1. 0.]]

which is the correct form of the CNOT(1,3) gate.

**01/03/2023**

After working more with the code, it appears that the implementation of Shor's algorithm that has been written is fundamentally incorrect and lacking. This is because it assumes that there is only one ancillary qubit. This is completely incorrect, and as a result the controlled U gates are implemented incorrectly (they aren't even unitary!)  
  
As a result, I have done a complete re-write of the program. Like I did in the Grover's algorithm subproject I have implemented the algorithm as a class called "shor". I also defined a second class called "contfraction" which is used when calculating the period from the output of shor's algorithm. contfraction is used to convert floating point numbers into continued fractions which is necessary for the computation.

**Icon

Description automatically generatedThe full program would not reasonably fit here, so it is included as an attachment:**

<<shor.py>>

Overall, the structure of the program is determined by the function *main()*:

def main():

while True:

target = 39

main\_register\_bitnumber = 5

#a=7

J=shor(target,bits=main\_register\_bitnumber,verbose=True)

output = J.run\_algorithm()

if output[1]: # Check if the algorithm was skipped

print(output[0],"\n")

else:

phase = output[0]

print(phase)

p = J.get\_period(phase)

print(phase,J.get\_factors(p),"\n")

Here's what this is doing:

1) Define the target ***N*** to factorise. ***N*** can be represented as an ***n***-bit integer

2) (Optional) Choose the number of bits in the *working* register and the pivot ***a***. If these aren't given, there will be 2n bits in the working register and ***a*** is a random integer such that .

3) An instance J of the "shor" class is created with our chosen parameters. For an n-bit integer there will be n qubits in the ancillary register. For example, ***N***=15 is a 4 bit integer so there would be 12 qubits in total if we didn't fix the number of working qubits before.

4) Attempt to run shor's algorithm. If ***a*** was already a non-trivial factor then the algorithm is skipped and we just get the resulting factorisation, skipping the rest. Otherwise, we obtain the output of Shor's algorithm. Call this the "phase" for no special reason.

5) Using the output of shor's algorithm, we calculate the estimated period of factors.

6) Calculate the resulting factorisation from the given period. Due to quantum error, the result may sometimes be incorrect or trivial but it usually isn't. The accuracy of the algorithm improves as the number of qubits in the working register increases.

For shor's algorithm itself, there are two key components in the circuit: the controlled U gates to calculate modulo arithmetic with respect to the pivot ***a*** and then the inverse quantum fourier transform as applied to the working qubit register. This is represented by the following quantum circuit diagram:

Shape

Description automatically generated with medium confidence

(Figure 1: Quantum circuit diagram for shor's algorithm. The working qubit register is initialised in the states 0, the ancillary register is initialised in the state with final qubit equal to 1 (000...0001). The ancillary register is measured after all the controlled U gates are applied, which also causes collapse in the working register due to the entanglement the controlled U gates create. The state the working register collapses into gives a fourier transformed version of a function of the period, so the inverse quantum fourier transform is applied and the working register is then measured)

The controlled U gates are constructed in python using an implementation of the algorithm outlined in D. Candela *Undergraduate computational physics projects on quantum computing* (American Journal of Physics, 2015).

This algorithm is described by Candela for the case N=15 for 7 qubits (3 working, 4 ancillary) as follows:

Text

Description automatically generated

The python implementation of this algorithm for more general cases is as follows:

def construct\_CU\_matrix(self,control\_bit):

"""

CU matrix is constructed according to algorithm outlined in undergrad

quantum computing projects paper

"""

index = self.main\_bitnumber-(control\_bit+1) # Reverse numerical ordering of main register

if index == 0:

A = self.a%self.N

else:

A = (self.a\*\*(2\*\*(index)))%self.N

CU = np.zeros((2\*\*self.bits,2\*\*self.bits))

get\_bin = lambda x, n: format(x, 'b').zfill(n)

for column\_number in range(2\*\*self.bits):

k = get\_bin(column\_number,self.bits)

if k[control\_bit]== "0":

j = int(k,2)

else:

main = k[:self.main\_bitnumber]

ancil = k[self.main\_bitnumber:]

f = int(ancil,2)

if f >= self.N:

j=int(k,2)

else:

f2 = (A\*f)%self.N

new\_ancil = get\_bin(f2,self.ancillary\_bitnumber)

j\_binary = main+new\_ancil # Combine strings

j = int(j\_binary,2)

CU[j][column\_number] = 1

return CU

Next comes the implementation of the inverse quantum fourier transform (IQFT). Since the IQFT matrix is only applied to the working register, the resulting matrix must be followed by the required number of tensor products with the 2x2 identity to handle the ancillary qubit register.

There are then two possible ways of implementing IQFT. The first uses the fundamental definition of the QFT matrix. For an N bit state, the QFT matrix is given by:

Table

Description automatically generated

where w is the n-th complex root of unity. The inverse QFT is then implemented by using w-1 in the definition instead.

Including the aforementioned tensor products for the ancillary qubits for the ancillary register, this matrix is constructed in python using the following code:

def get\_IQFT\_matrix\_v2(self):

"""

Alternative method to get inverse quantum fourier transform matrix on main register

Uses mathematical definition of IQFT instead of building out of unary & binary gates

By default this method remains unused!

"""

if self.verbose: print("Computing IQFT matrix for {} bits in working register".format(self.main\_bitnumber))

N = 2\*\*self.main\_bitnumber

omega = np.exp(-2\*np.pi\*1j/N)

IQFT\_matrix = np.array([[0 for j in range(N)] for i in range(N)],dtype=complex)

for i in range(N):

for j in range(N):

IQFT\_matrix[i][j] = omega\*\*(i\*j)

IQFT\_matrix \*= 1/(np.sqrt(N))

for i in range(self.ancillary\_bitnumber):

IQFT\_matrix = np.kron(IQFT\_matrix,np.identity(2))

if self.verbose: print("Done!")

return IQFT\_matrix

Alternatively, QFT can be implemented using a combination of hadamard and controlled rotation gates via the following circuit diagram:

Diagram

Description automatically generated

(Figure 2: The quantum circuit diagram for the quantum fourier transform of N qubits. The inverse QFT is then obtained by replacing exp(2\*pi\*i/N) with exp(-2\*pi\*i/N)

This is implemented with the following python code:

def get\_IQFT\_matrix(self):

"""

Inverse quantum fouier transform function for given N qubit state.

Adjusted so it does not act on the ancillary qubits, only the main register

"""

if self.verbose: print("Computing IQFT matrix for {} bits in working register".format(self.main\_bitnumber))

circuit = np.identity(2\*\*self.main\_bitnumber)

for i in reversed(range(self.main\_bitnumber)):

gate1 = extend\_unary(targets=[i-1],gate=self.HADAMARD,bits=self.main\_bitnumber)

circuit = np.matmul(gate1,circuit)

for k in range(i): #Apply CROT gates to every qubit below the current one

if i-1 == k:

break

CROT = np.array([[1,0,0,0],[0,np.e\*\*(-2\*np.pi\*1j/(i-k)),0,0],[0,0,1,0],[0,0,0,1]])

gate2 = extend\_binary(q=(k,i-1),gate=CROT,bits=self.main\_bitnumber)

circuit = np.matmul(gate2,circuit)

if self.verbose: print("Applying tensor products to IQFT matrix for ancillary register...")

for i in range(self.ancillary\_bitnumber):

circuit = np.kron(circuit,np.identity(2))

return circuit

Both versions can be used in the final program, but by default the second version is used.

After the quantum algorithm is run, the working qubit register is measured. The resulting bitstring is treated as an integer number, with the bit that was adjacent to the ancillary bit corresponding to the smallest bit. This result is then divided by 2^L, where L is the number of working qubits. This causes the result to be normalised between 0 and 1. For the purposes of the program, this is called the "phase".

The program then converts this phase into a fractional representation **y/Q**, which it then further converts into a continued fractional expansion of the form:

A picture containing text, clock

Description automatically generated

(In theory this fraction could be infinite, but the program terminates the expansion after a finite number of terms)

The program then uses the terms in the expansion to create successive approximations **d/s** of the phase. It continues doing this until it finds an approximation d/s that satisfies:

1. S<N, where N is the target number to factorise
2. A picture containing text, clock, watch

   Description automatically generated

When these conditions are satisfied, s is then likely to either be the appropriate period or a multiple thereof. The program then checks through multiples of s (i.e. p=ns for n=1,2,3...) to find the given trial period p that satisfies a^p mod N = 1.

Finally, once the program finds the period it then computes the pair . These two integers will be factors of N and are statistically likely to be non-trivial factors, which is what the algorithm is searching for.

**02/03/2023**

I have combined our code for RSA encryption/decryption with our code for Shor's algorithm to compute the prime factors of the modulus of an RSA key. 8-bit RSA generates 8-bit integer moduli, so the working qubit register is restricted to a maximum of 5 qubits due to hardware limitations. While the algorithm will still work in this condition, it isn't ideal.

Regardless, once the program finds the factors of the moduli it then checks that they are valid non-trivial factors. Assuming they are, it then computes the private key as follows:

1. Call the public key e, the private key d and the modulus m
2. Factorise m into non-trivial factors p,q using Shor's Algorithm.
3. Compute d from p,q and e in the usual way for RSA key generation. This means setting s=(p-1)\*(q-1) and then calculating

The program then checks if the generated private key is the same as the true one to see if the program was successful.

**The full python code that implements this is attached here, alongside Sid's RSA code:**

<<RSA\_breaker.py>>

<<RSA.py>>

Using this code, I then tested how successful the algorithm was at breaking RSA encryption. To do this, the number of qubits in the working register was varied between 1 and 5. The algorithm was then run for 100 shots for each working register size. The key chosen was not fixed, instead the program was attempting to solve randomly generated RSA problems on every single shots. The program kept track of the number of successes vs failures for each shot of the algorithm and then computed the measured probability of a success for a given number of qubits.

Chart, box and whisker chart

Description automatically generated

(Figure 3: The experimental probability of successfully finding the correct private key for a given 8-bit RSA public key. On every shot of the program, the algorithm returned 1 if it was successful and 0 if it was unsuccessful. The program was then run for 100 shots. The error bars used here are then the standard error, not standard deviation as that would not be appropriate for this case)  
  
  
  
  
  
**03/02/2023**



I have tested the output of Shor's algorithm for factoring N=15 for various sizes *L* of the working qubit register.

**Chart, line chart

Description automatically generated**Chart, line chart

Description automatically generated

Chart, histogram

Description automatically generated**Chart, line chart

Description automatically generated**

(Figure 4: Measured probabilities of different outputs of Shor's Algorithm for factoring N=15 for working register sizes of L=3,4,5,6. A fixed value of a=7 was used in every single execution of the algorithm to ensure consistency, however this is not realistic to a real-world application of the algorithm. It can be seen that although there are *local* peaks in the distributions which correspond to measured periods *p*, which become more defined and frequent as the number of working bits increases. This is as expected for Shor's algorithm)

Chart, box and whisker chart

Description automatically generated

(Figure 5: Experimental probability of a successful factorisation of 15 into [3,5] using Shor's algorithm for different working register sizes. Again, a fixed value of a=7 was used in every single execution of the algorithm to ensure consistency. Overall, the algorithm seems to be consistently successful between 70-80% of the time for the register sizes shown here. Surprisingly, L=6 has a much lower experimental probability of success. This may be due to floating point error leading to failure in the algorithm. Since the matrices become quite large for L=6 and must still maintain unitarity, their components must hence become longer floating point numbers as they become smaller and smaller, which then leads to errors in the calculations. There is also the possibility that this is just a sampling error and that 20 trials, 100 shots is too small to determine the underlying probability distribution)

**Chart, line chart

Description automatically generated**