Week 11 (Test Page)

Monday, April 18, 2022
12:29 PM

Hello world! (17/01/2023)

✓ [illegible] 23/01

Week 12

24/01/2023

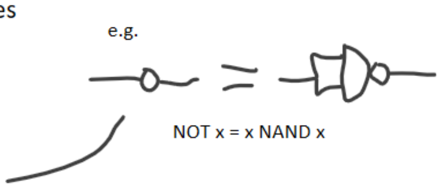
Classical Logic Basics

A bit is a 0 or 1, i.e. an element of the set {0,1}.
2 bits would be a (0,0) or (1,0) or etc., i.e. an element of {0,1}x{0,1}
Etc.

We can then map bits to other bits with logic gates, which mathematically are just functions of bits. A unary logic gate maps one bit to another bit. A binary logic gate maps two bits to one bits. In principle all logic gates can be built out of these.

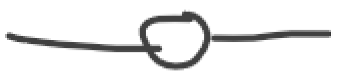
e.g. NOT x is a unary gate. NOT(0) = 1, NOT(1) = 0
e.g.2 binary gates
OR: 0 OR 0 = 0, 0 OR 1 = 1, 1 OR 0 = 1, 1 OR 1 = 1
AND: 0 AND 0 = 0, 0 AND 1 = 0, 1 AND 0 = 0, 1 AND 1 = 1
XOR: 0 XOR 0 = 0, 0 XOR 1 = 1, 1 XOR 0 = 0, 1 XOR 1 = 1

Since a bit is an element of a 2-element set, there are 2^2=4 possible unary gates and 2^(2*2)=16 possible binary gates. Can also construct gates by combining them, e.g. NAND = NOT(AND). w/o proof we can just say that ALL possible logic gates can be built out of NAND gates.

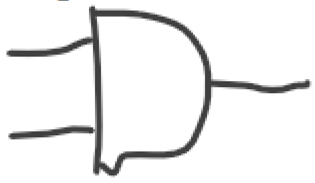


Can represent logic gates with diagrams!

e.g. NOT



e.g AND



e.g. NAND = NOT(AND)

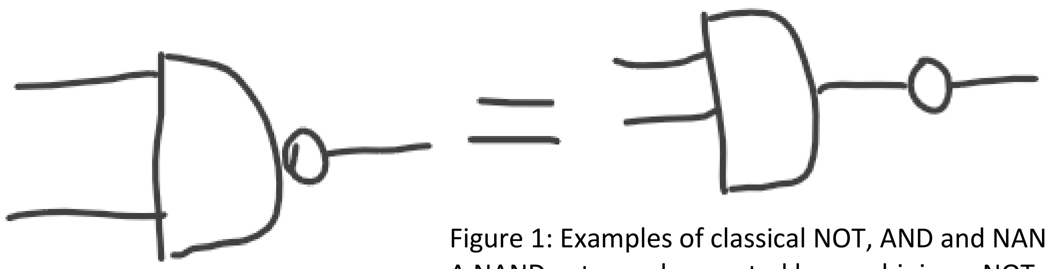


Figure 1: Examples of classical NOT, AND and NAND gates.
A NAND gate can be created by combining a NOT and NAND gate

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>

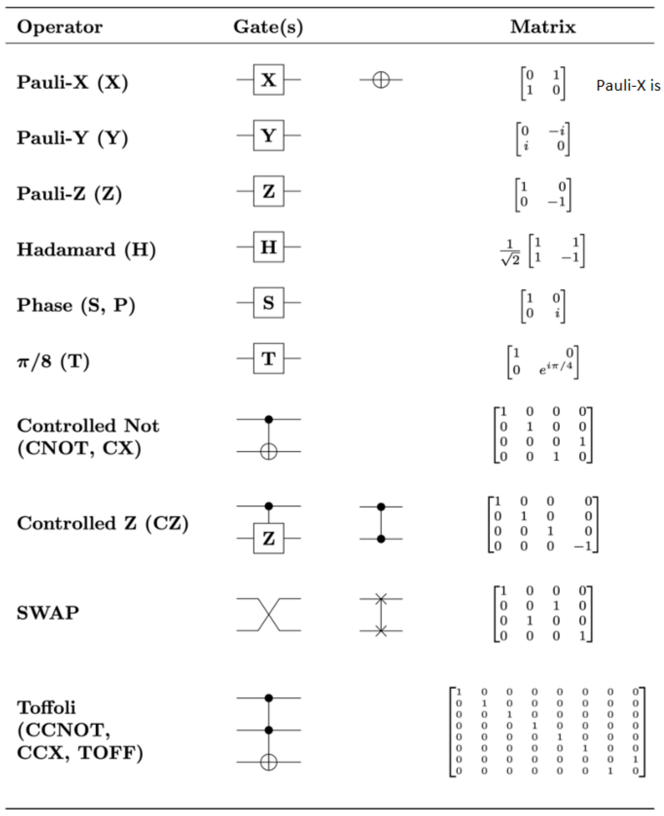
Since this is a tensor product space, the possible qubit states may or may not be decomposable into tensor products |A>⊗|B>. If we can write this, then the state is separable and the two qubits are not correlated (i.e. they are statistically independent). If we can't separate the states into a tensor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

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

$$\begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix}$$

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



Operator	Gate(s)	Matrix
Pauli-X (X)	$\boxed{X}$, $\oplus$	$\begin{bmatrix}0&1\\1&0\end{bmatrix}$
Pauli-Y (Y)	$\boxed{Y}$	$\begin{bmatrix}0&-i\\i&0\end{bmatrix}$
Pauli-Z (Z)	$\boxed{Z}$	$\begin{bmatrix}1&0\\0&-1\end{bmatrix}$
Hadamard (H)	$\boxed{H}$	$\frac{1}{\sqrt{2}}\begin{bmatrix}1&1\\1&-1\end{bmatrix}$
Phase (S, P)	$\boxed{S}$	$\begin{bmatrix}1&0\\0&i\end{bmatrix}$
$\pi/8$ (T)	$\boxed{T}$	$\begin{bmatrix}1&0\\0&e^{i\pi/4}\end{bmatrix}$
Controlled Not (CNOT, CX)		$\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{bmatrix}$
Controlled Z (CZ)	$\boxed{Z}$	$\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&-1\end{bmatrix}$
SWAP		$\begin{bmatrix}1&0&0&0\\0&0&1&0\\0&1&0&0\\0&0&0&1\end{bmatrix}$
Toffoli (CCNOT, CCX, TOFF)		$\begin{bmatrix}1&0&0&0&0&0&0&0\\0&1&0&0&0&0&0&0\\0&0&1&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&0&0&1&0&0&0\\0&0&0&0&0&1&0&0\\0&0&0&0&0&0&0&1\\0&0&0&0&0&0&1&0\end{bmatrix}$

Pauli-X is the quantum equivalent of a NOT gate!

Figure 2: A list of unary and binary quantum logic gates. These are combined together to act on multiple qubits via tensor/kronecker products
(Image taken from https://commons.wikimedia.org/wiki/File:Quantum_Logic_Gates.png)

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

I've written a simple program in python that can apply unary quantum gates to different qubits in an N-qubit system. These can then be combined together to build some simple quantum circuits. In the version attached here it just applies PAULI-X gates to two separate qubits, however it can be modified with minimal effort to include any other unary gates and any number of qubits.

unary_gate

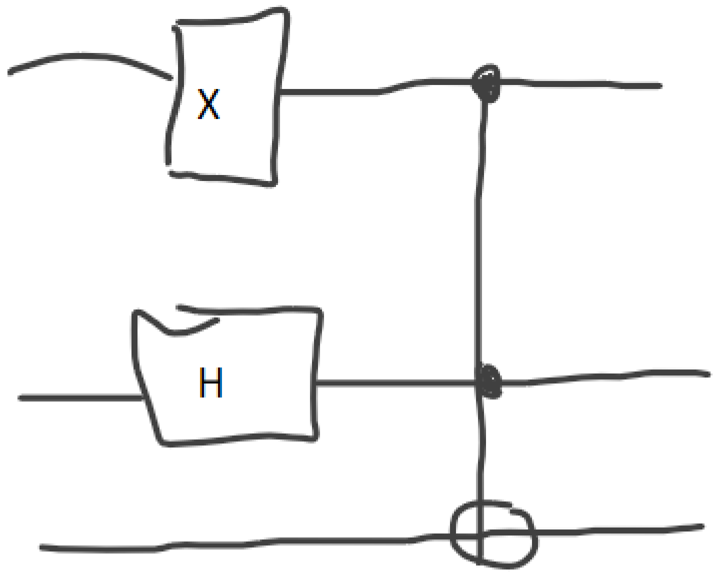
✓ thanks, I can read this OK.

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*.

Excellent - regular entries and a good level of detail. [illegible] 30/01.

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

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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"

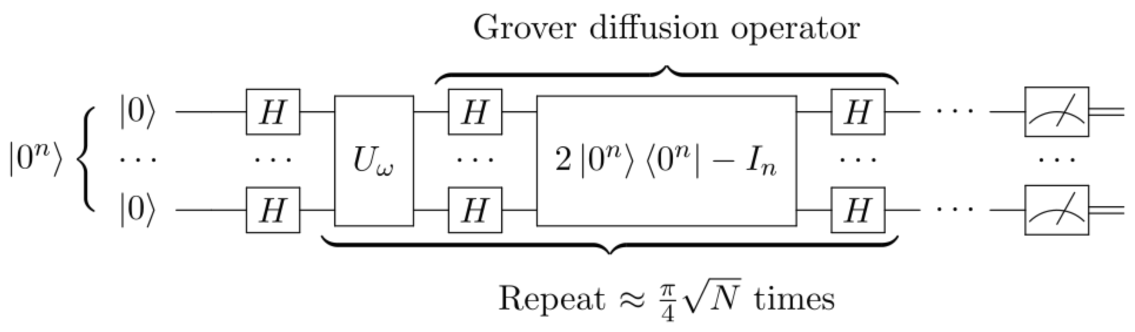
Let x be an arbitrary element of a database and we are searching for a specific element x_i in the database that satisfies a search requirement. This search requirement can be codified by a function called a quantum oracle which returns a Boolean value

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.



Grover diffusion operator
$|0^n\rangle$
$|0\rangle$
$|0\rangle$
H
U_ω
$2|0^n\rangle\langle 0^n| - I_n$
Repeat $\approx \frac{\pi}{4}\sqrt{N}$ times

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:

$$\text{ESI} = \prod_{i=1}^{n}\left(1 - \left|\frac{x_i - x_{i0}}{x_i + x_{i0}}\right|\right)^{\frac{w_i}{n}}$$

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.

Excellent! [illegible] 06/02

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:

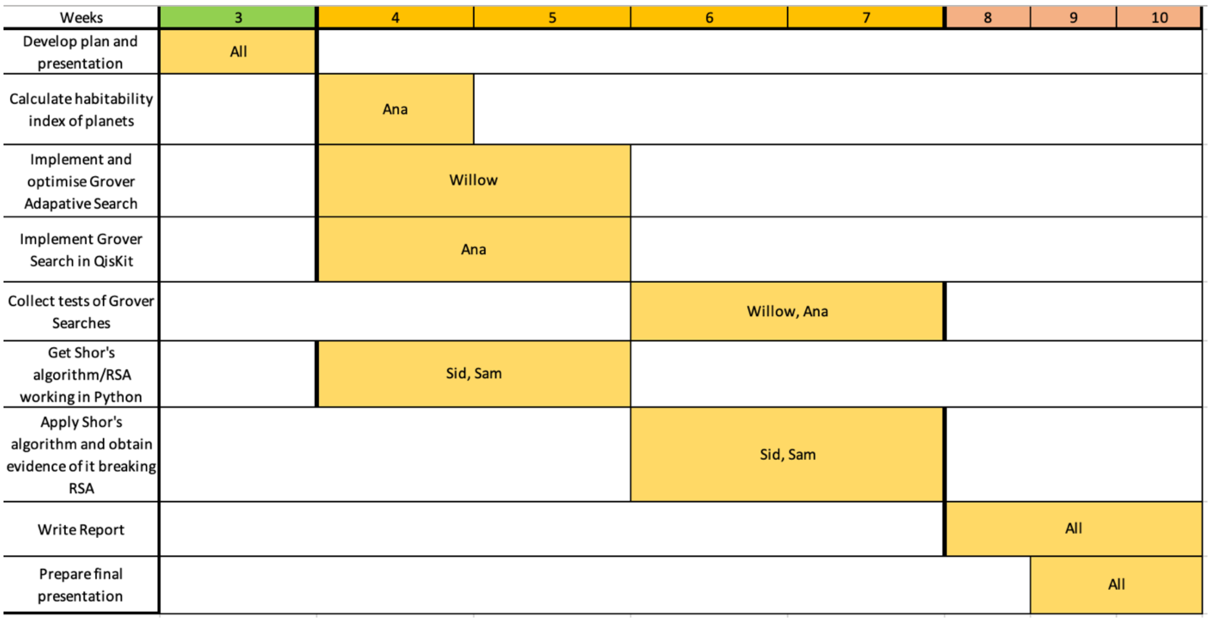
$$P_f\,|x\rangle = (-1)^{f(x)}\,|x\rangle$$

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:



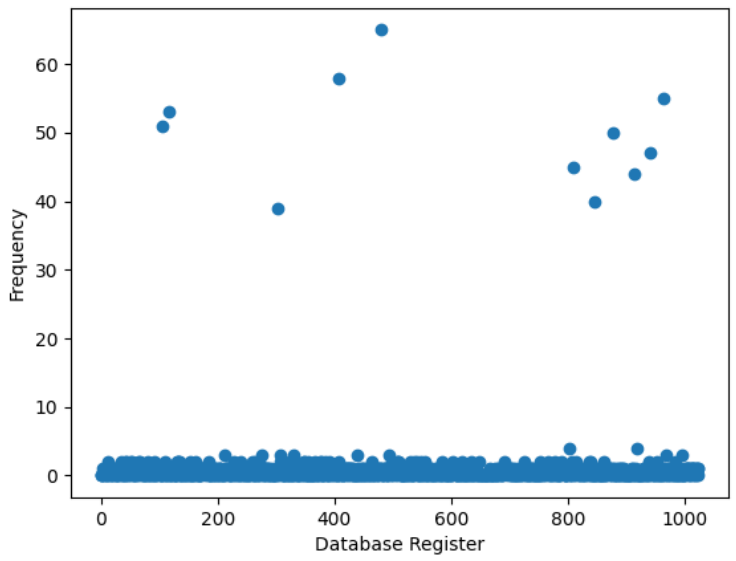
Weeks	3	4	5	6	7	8	9	10
Develop plan and presentation	All							
Calculate habitability index of planets		Ana						
Implement and optimise Grover Adapative Search		Willow						
Implement Grover Search in QisKit		Ana						
Collect tests of Grover Searches				Willow, Ana				
Get Shor's algorithm/RSA working in Python		Sid, Sam						
Apply Shor's algorithm and obtain evidence of it breaking RSA				Sid, Sam				
Write Report						All		
Prepare final presentation							All	

(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:



(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.

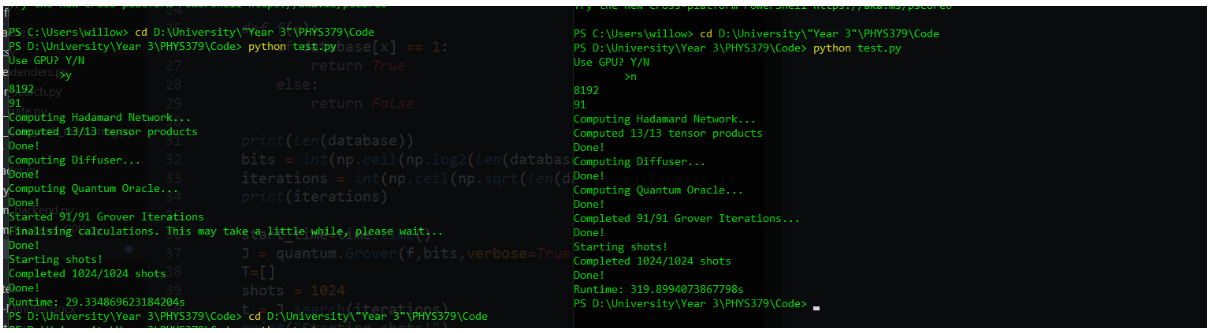
3. The CPU and GPU have separate memory pools. On my laptop there is 16GB DDR4 of RAM available to the CPU and 6GB GDDR6 of VRAM available to the GPU. This means transferring data between the CPU and GPU introduces extra latency, which means the GPU version may actually take longer to run than the CPU 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_...

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.



```
PS C:\Users\willow> cd D:\University\"Year 3"\PHYS379\Code
PS D:\University\Year 3\PHYS379\Code> python test.py
Use GPU? Y/N
        >y
8192
91
Computing Hadamard Network...
Computed 13/13 tensor products
Done!
Computing Diffuser...
Done!
Computing Quantum Oracle...
Done!
Started 91/91 Grover Iterations
Finalising calculations. This may take a little while, please wait...
Done!
Starting shots!
Completed 1024/1024 shots
Done!
Runtime: 29.334869623184204s
PS D:\University\Year 3\PHYS379\Code> cd D:\University\"Year 3"\PHYS379\Code
```

```
PS C:\Users\willow> cd D:\University\"Year 3"\PHYS379\Code
PS D:\University\Year 3\PHYS379\Code> python test.py
Use GPU? Y/N
        >n
8192
91
Computing Hadamard Network...
Computed 13/13 tensor products
Done!
Computing Diffuser...
Done!
Computing Quantum Oracle...
Done!
Completed 91/91 Grover Iterations...
Done!
Starting shots!
Completed 1024/1024 shots
Done!
Runtime: 319.8994073867798s
PS D:\University\Year 3\PHYS379\Code>
```

(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.

<u>**09/02/2023**</u>

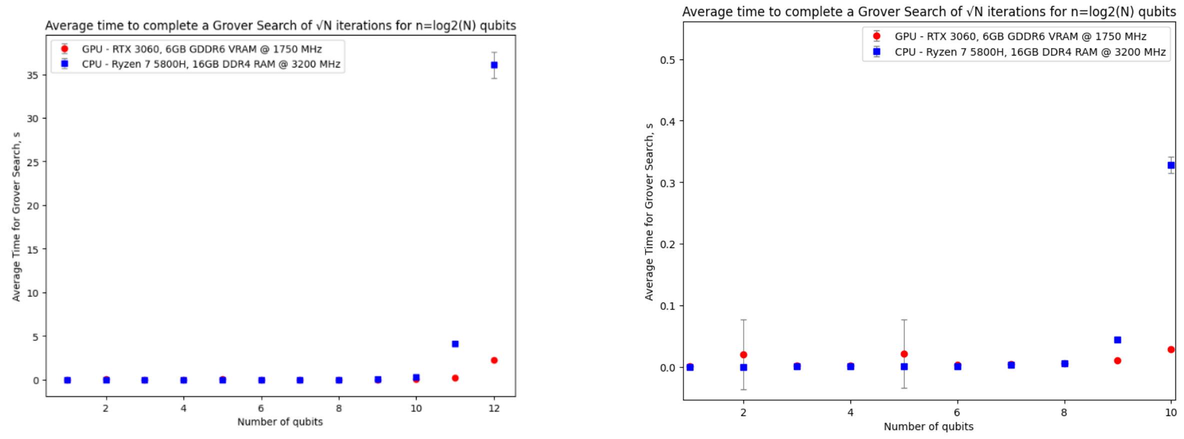
I have attempted to convert the matrices into sparse matrices using scipy's sparse matrix library and cupy's variant of this. I had success with converting the diffuser and quantum oracle into sparse matrices, however they do not conform to the same sparse matrix types. The quantum oracle makes sense 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



(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.

Excellent! [illegible] 13/02

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)

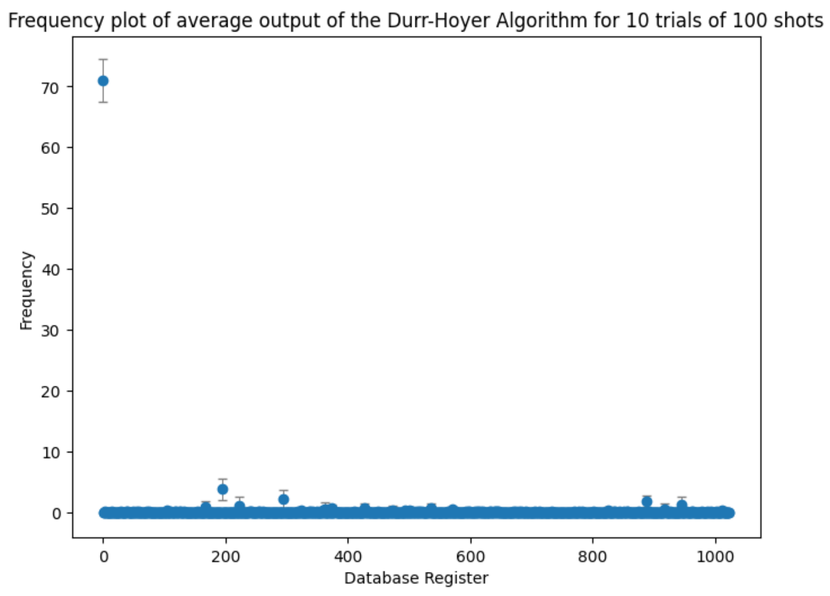
(By threshold Yn, this refers to having an oracle function which returns 1 if y<Yn for every element y in the database f. x and Xn refer to registers of elements in the database)
Baritompa et al generally makes the recommendation of choosing λ=1.34. For now I'll choose this but it will probably be useful 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 $X_{n+1}=X_n$ 5 times in a row, set $\lambda=1.34$) I have tested the algorithm for 100 shots. I repeated this "100 shot test" ten times and then plotted the average results.



(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 $y<Y_n$ 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" $\boldsymbol{\mu}$ (so the previous work done so far had $\mu=5$).

<u>Meetings</u>
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.

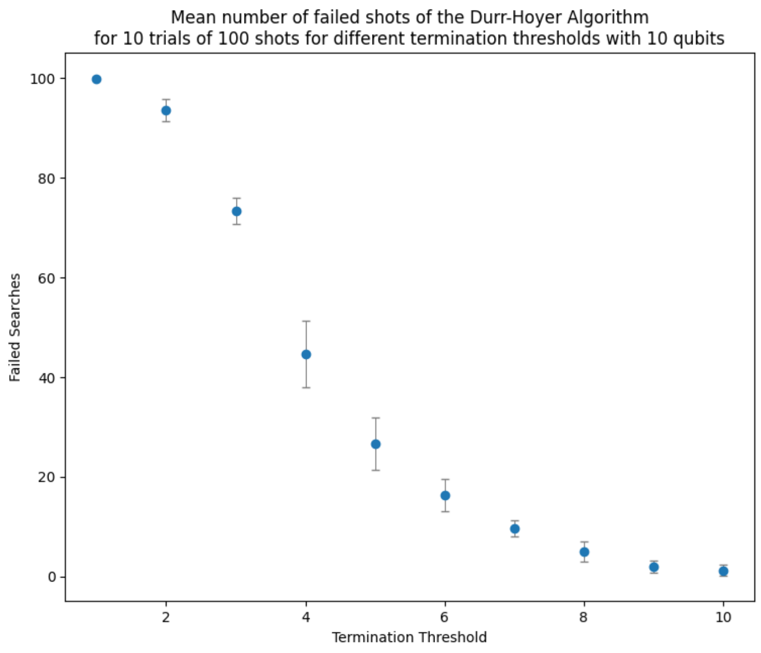
On another aside I have actually thought about the fact that what I have written so far searches for the minimum value entry. Since we are looking for the maximum value entry in the ESI database I have clearly made a mistake here. Luckily this is an easy fix (just need to flip the order of the inequality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

<u>15/02/2023</u>

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

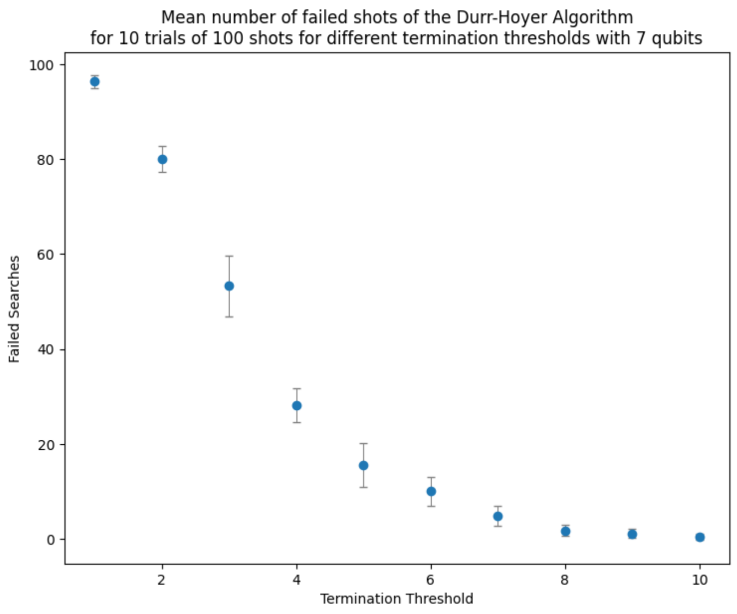


(Figure 2: On the left is the full table of fail rates for different values of µ. On the right is a scatter plot of the data. Assuming a maximum tolerance of 5% failure, it is clear that the ideal threshold for a 10 qubit search is µ=9. µ=8 would also work pretty well, however its uncertainty means it 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)

<u>16/02/2023</u>

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.

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:

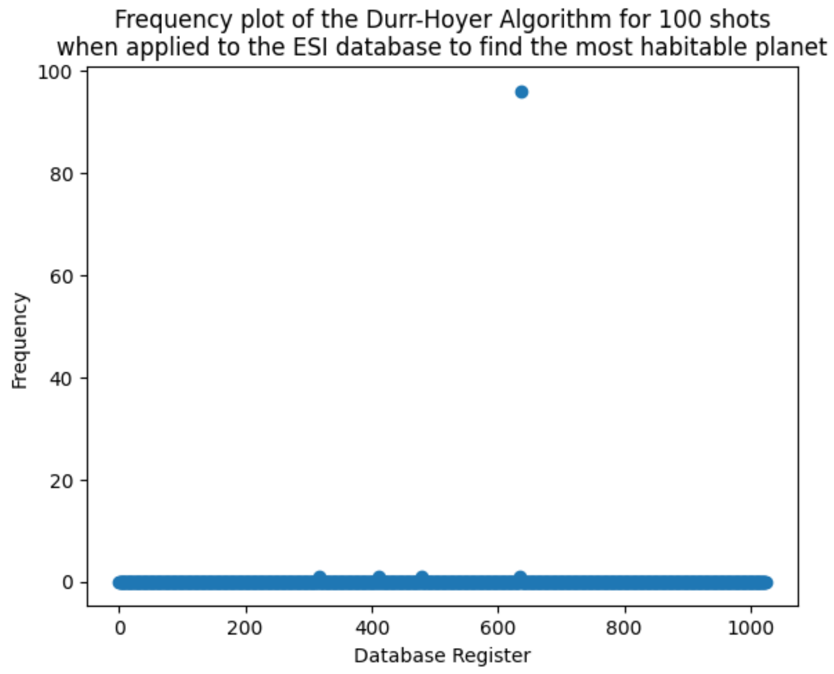
```
def import_ESI_data():
    with open("esi.csv",newline="") as file:
        data_reader = csv.reader(file,delimiter=",")
        data = [row for row in data_reader]
        del data[0]                # Remove Names/ESI line at the start
        of the file
        data_length = len(data)
        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_ad...

esi

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).



(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.

Excellent! [illegible] 20/02

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.

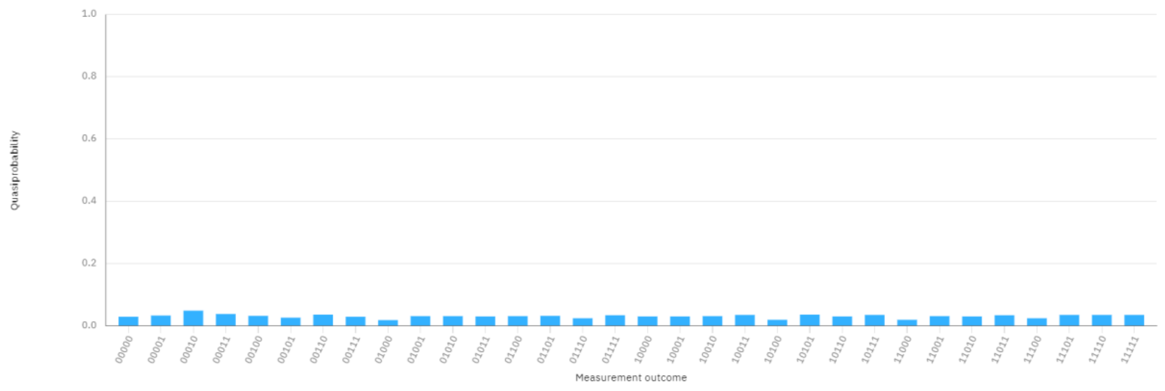
```
class Grover:

    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_...

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).



(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...

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.

<u>21/02/2023</u>

<u>Held our Group Meeting Today</u>
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.

<u>22/02/2023</u>

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

$$\begin{aligned} R_{\hat{n}}(\varphi) &= \exp[-i\varphi(n_x X + n_y Y + n_z Z)/2] \\ &= \cos(\varphi/2) I - i \sin(\varphi/2)(n_x X + n_y Y + n_z Z). \end{aligned}$$

where I is the identity 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
```

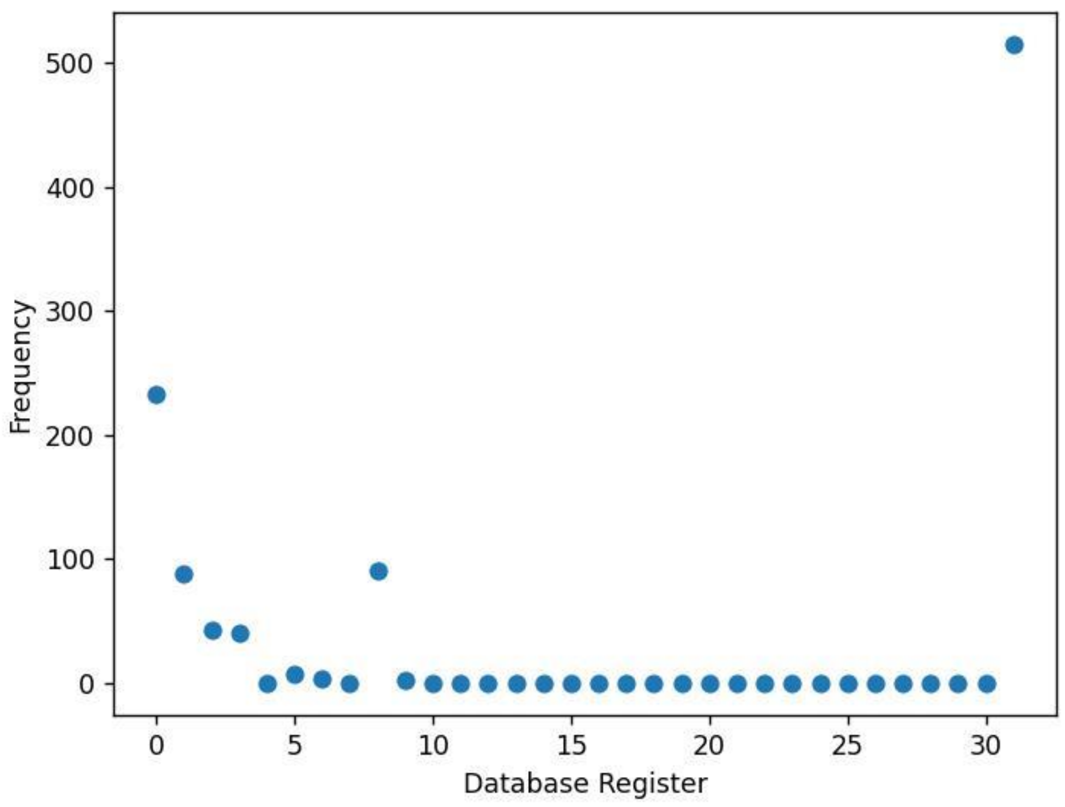
(This is the GPU version. For the CPU version all references to the cupy module labelled as cp must be replaced with references to numpy)

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_...

Excellent! [illegible] 27/02

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 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.

The full program would not reasonably fit here, so it is included as an attachment: ✓

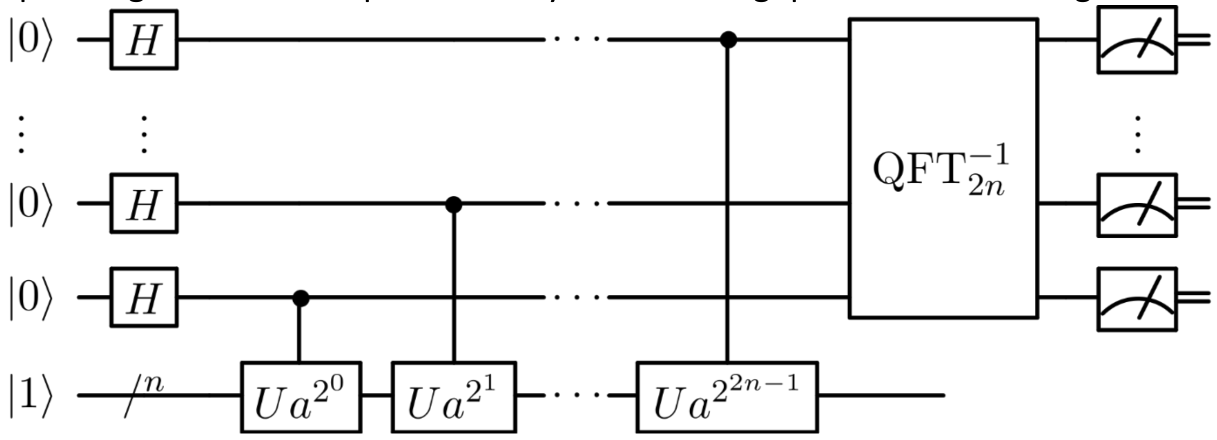
shor

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 $1 \leq a < N$.
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:



(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:

(1) Compute $A_0 = a \pmod{15}$, $A_1 = a^2 \pmod{15}$, and $A_2 = a^4 \pmod{15}$.

(2) Start with the first controlled gate in Fig. 8, controlled by ℓ_0. In each column $k = 0 \ldots 127$ of the matrix,[29] the entries are all 0's except for a 1 in some row j. To compute j:

(a) Write the column k as a 7-bit binary number $k = \ell_2\ell_1\ell_0 m_3 m_2 m_1 m_0$ as on the left side of Fig. 8.

(b) If $\ell_0 = 0$, then $j = k$. This puts the 1 in column k on the diagonal, as in an identity matrix. (Conceptually if $\ell_0 = 0$, the gate does nothing to the f-register.)

(c) If $\ell_0 = 1$, then express the four-bit binary number $m_3 m_2 m_1 m_0$ as an integer f. If $f \geq 15$ (i.e., if $f \geq C$), then again set $j = k$.

(d) If $\ell_0 = 1$ and $f < 15$, compute a new f value $f' = A_0 f \pmod{15}$. Write this in binary as $f' = m'_3 m'_2 m'_1 m'_0$. Then j is given by the 7-bit binary number $j = \ell_2 \ell_1 \ell_0 m'_3 m'_2 m'_1 m'_0$. [Conceptually if $\ell_0 = 1$ then the f-register is multiplied by $A_0 \pmod{15}$.]

(3) Use the same procedure to compute the other two controlled gates, substituting ℓ_1, ℓ_2 for ℓ_0 and A_1, A_2 for A_0.

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:

$$F_N = \frac{1}{\sqrt{N}} \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \cdots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2(N-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \cdots & \omega^{3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \omega^{3(N-1)} & \cdots & \omega^{(N-1)(N-1)} \end{bmatrix}$$

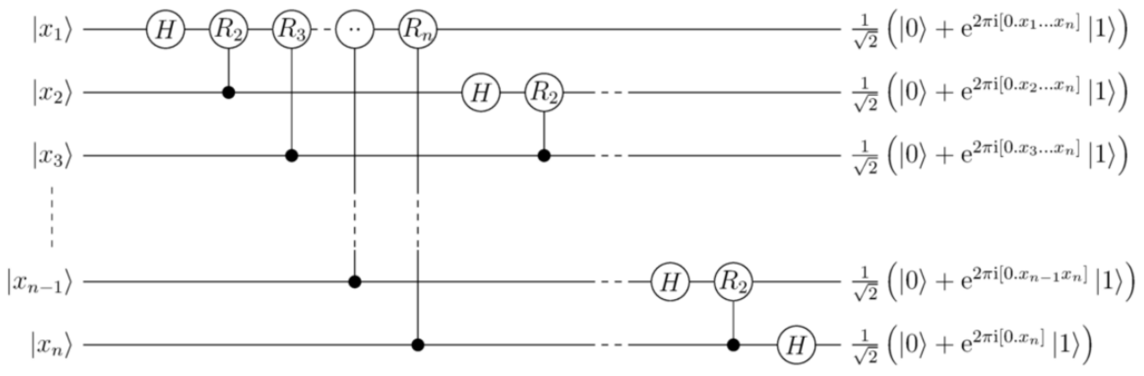
where w is the n-th complex root of unity. The inverse QFT is then implemented by using w^{-1} in the definition instead.
Including the aforementioned tensor products for the ancillary qubits for the ancillary register, this matrix is constructed in python using the following code:

```
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:

$$H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \qquad \text{and} \qquad R_n = \begin{pmatrix} 1 & 0 \\ 0 & e^{2\pi i/N} \end{pmatrix}$$

with $\omega_N = e^{i2\pi/N}$ the N-th root of unity. The circuit is composed of H gates and the controlled version of R_n



$|x_1\rangle$
$|x_2\rangle$
$|x_3\rangle$
$|x_{n-1}\rangle$
$|x_n\rangle$
H
R_2
R_3
R_n
$\frac{1}{\sqrt{2}}\left(|0\rangle + e^{2\pi i[0.x_1\ldots x_n]}|1\rangle\right)$
$\frac{1}{\sqrt{2}}\left(|0\rangle + e^{2\pi i[0.x_2\ldots x_n]}|1\rangle\right)$
$\frac{1}{\sqrt{2}}\left(|0\rangle + e^{2\pi i[0.x_3\ldots x_n]}|1\rangle\right)$
$\frac{1}{\sqrt{2}}\left(|0\rangle + e^{2\pi i[0.x_{n-1}x_n]}|1\rangle\right)$
$\frac{1}{\sqrt{2}}\left(|0\rangle + e^{2\pi i[0.x_n]}|1\rangle\right)$

(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:

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots + \cfrac{1}{a_n}}}}$$

(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. $\left|\frac{y}{Q} - \frac{d}{s}\right| < \frac{1}{2Q}$.

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 $\left[\gcd\left(a^{\frac{p}{2}} - 1, N\right), \gcd\left(a^{\frac{p}{2}} + 1, N\right)\right]$. 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 $d = e^{-1} \mathrm{mod}\, s$

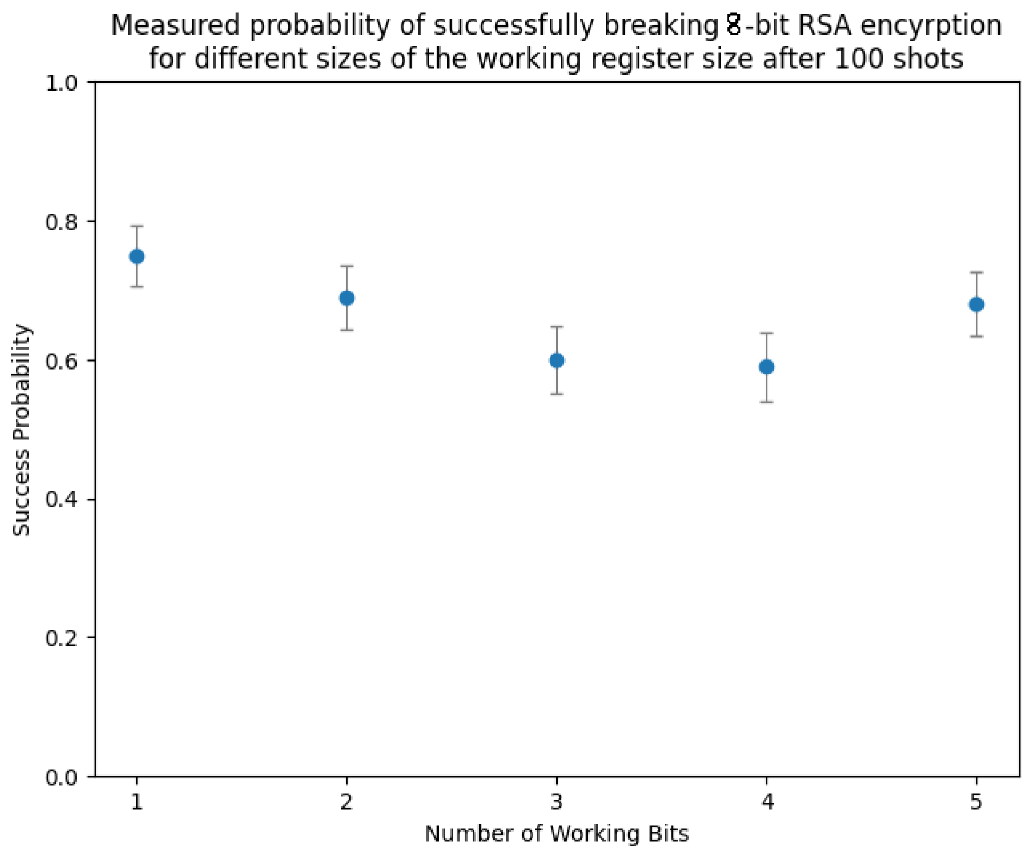
The program then checks if the generated private key is the same as the true one to see if the program was successful.

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

RSA_break...

RSA

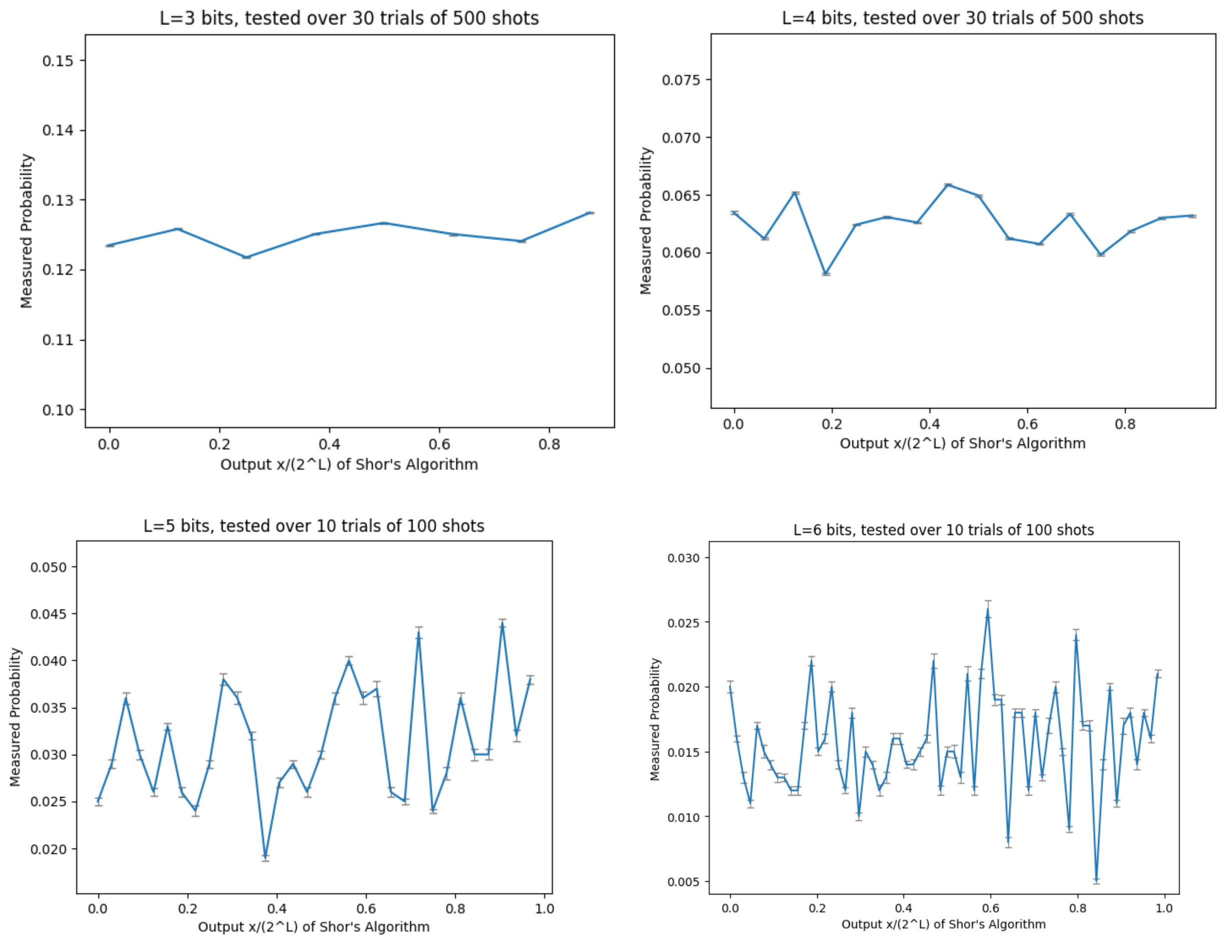
Using this code, I then tested how successful the algorithm was at breaking RSA encryption. To do this, the number of qubits in the working register was varied between 1 and 5. The algorithm was then run for 100 shots for each working register size. The key chosen was not fixed, instead the program was 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.



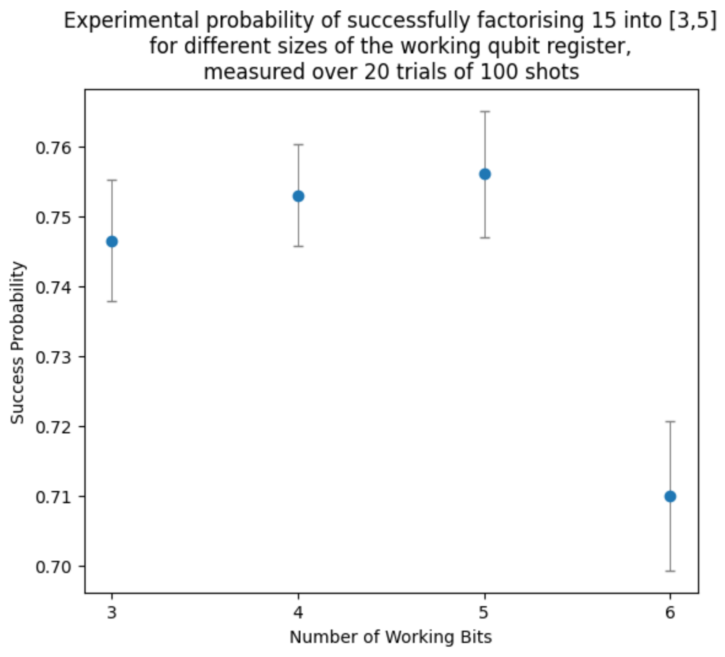
(Figure 3: The experimental probability of successfully finding the correct private key for a given 8-bit RSA public key. On every shot of the program, the algorithm returned 1 if it was successful and 0 if it was unsuccessful. The program was then run for 100 shots. The error bars used here are then 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.



(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)

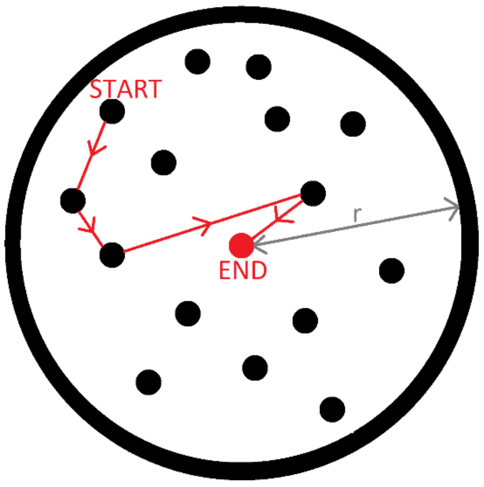


(Figure 5: Experimental probability of a successful factorisation of 15 into [3,5] using Shor's algorithm for different working register sizes. Again, a fixed value of a=7 was used in every single execution of the algorithm to ensure consistency. Overall, the algorithm seems to be consistently successful 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)

Excellent! [illegible] 06/03

Week 18
06/03/2023

Today I started working on the section in our report explaining Grover Adaptive Search (GAS) and the Durr-Hoyer algorithm. I have discussed how the efficiency of Grover search means it is well suited to solving optimisation problems and how GAS implements this, how the Durr-Hoyer algorithm is designed and the advantages the GAS brings to optimisation problems (e.g. most GAS algorithms typically require little to no knowledge of the global behaviour of a function to solve optimisation problems, instead they find the global minima/maxima of a function by simply taking a random walk through the input space guided by successive rapid Grover searches). I now need to discuss the origin of the choice of $\lambda = 1.34$ in the Durr-Hoyer algorithm as recommended by Baritompa et al. I also need to discuss the choice of termination condition and the origin of the termination threshold μ, which I previously tested and set equal to 9 for the 10 qubit GAS implementation.



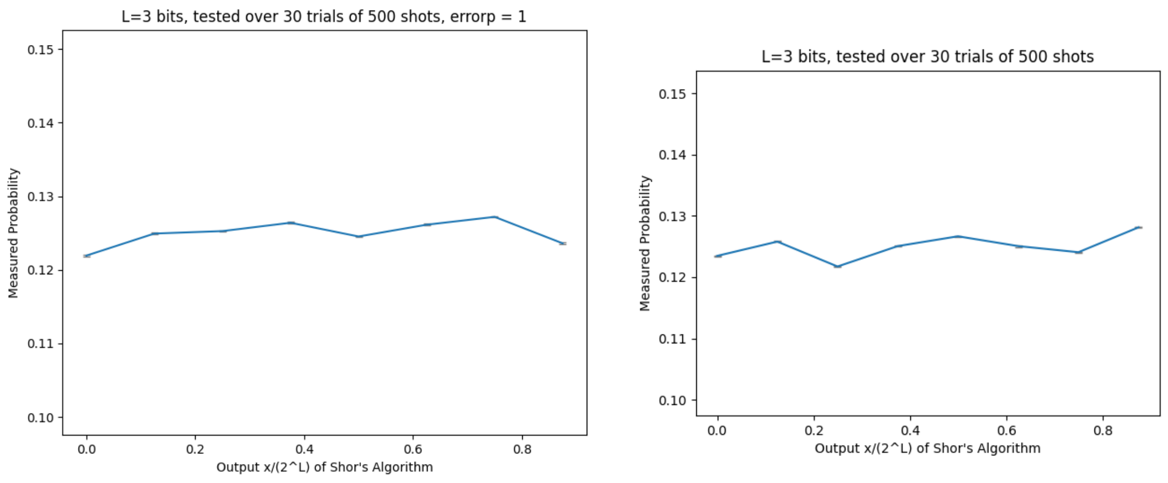
(Figure 1: A diagrammatic representation of a Grover Adaptive Search to find the element at the minimum distance from the circle's centre. The algorithm starts at a random point and then uses Grover searches to find random elements at successively lower distances from the centre, thereby taking a guided random walk to the optimal element where it ends.)

07/03/2023

Held our group meeting today, where I explained the status of the Shor's Algorithm project and what had gone wrong

Following on from our meeting today, I will focus on testing errors in our implementation of Shor's Algorithm and RSA cracking using the same error simulation method I wrote in the Grover's Algorithm project.

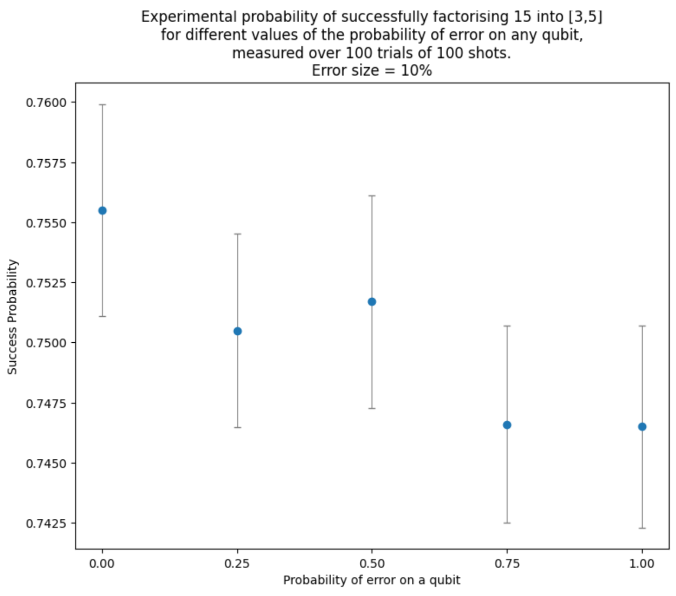
To implement error, I have added a parameter *errorp* to the shor.run_algorithm() function. If this parameter is not specified when the function is called, the program does not simulate error. Otherwise, if the parameter is set to a floating point number between 0 and 1 the program treats this number as the probability of introducing an "error gate" in the same manner as done in the Grover search program. These error gates can appear at any single point in the circuit for Shor's algorithm.



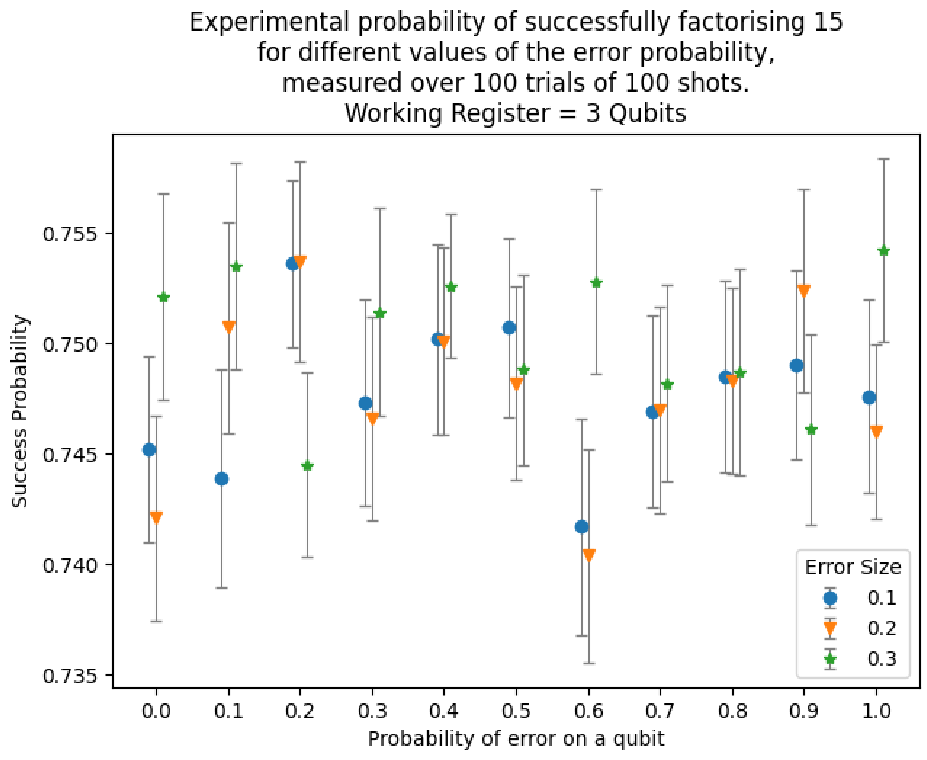
(Figure 2: (Left) Measured probabilities of different outputs of Shor's Algorithm for factoring N=15 for a working register size L=3. The probability of an error occurring on a qubit at every point in the circuit*, errorp*, is equal to 1 and the error_size is 10%. (Right) The same data but for errorp=0 as measured last week. Although these two graphs look similar they are very clearly not the same, and the peaks in the probability distribution for the output with error simulation are slightly less pronounced than those in the original version)

<u>08/03/2023</u>

Today I ran a range of tests on how probability of a successful factorisation into prime factors of various integers was affected by varying errorp and the error size.



(Figure 3: Initial graph generated to test that the program works. errorp is incremented by 0.25 between 0 and 1. There are 3 working qubits, the target is N=15 and the error size is 10%. The variation in the probability of success is very small for this error size, however there is a very marginal downwards trend as errorp increases)

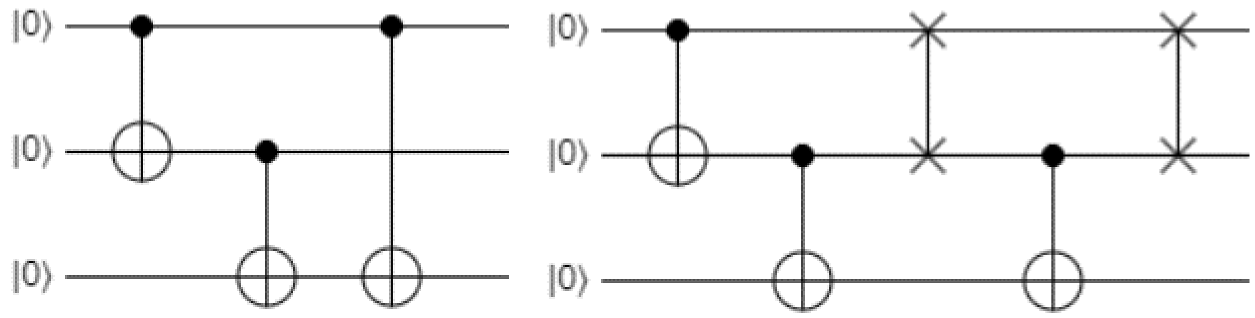


(Figure 4: Tests of the success probabilities for different small values of the error size. There is very little variation in the probability of success overall. The fact the probability of success for errorp=0 varies slightly as well is due to sampling bias, implying much of this variation may also be due to sampling bias. Overall, it is evident that our implementation of Shor's Algorithm is remarkably stable)

09/03/2023

Today I focused on writing the report. I have written up a section explaining the error simulation algorithm and I have started work on a section that explains how a quantum circuit may be simulated mathematically through a combination of vectors, matrix multiplication and tensor products. There is also now a discussion of the geometric picture of a qubit state vector via the Bloch Sphere, which is crucial to our implementation of errors as our effectively we are simulating error by performing randomised rotations of the Bloch Sphere.

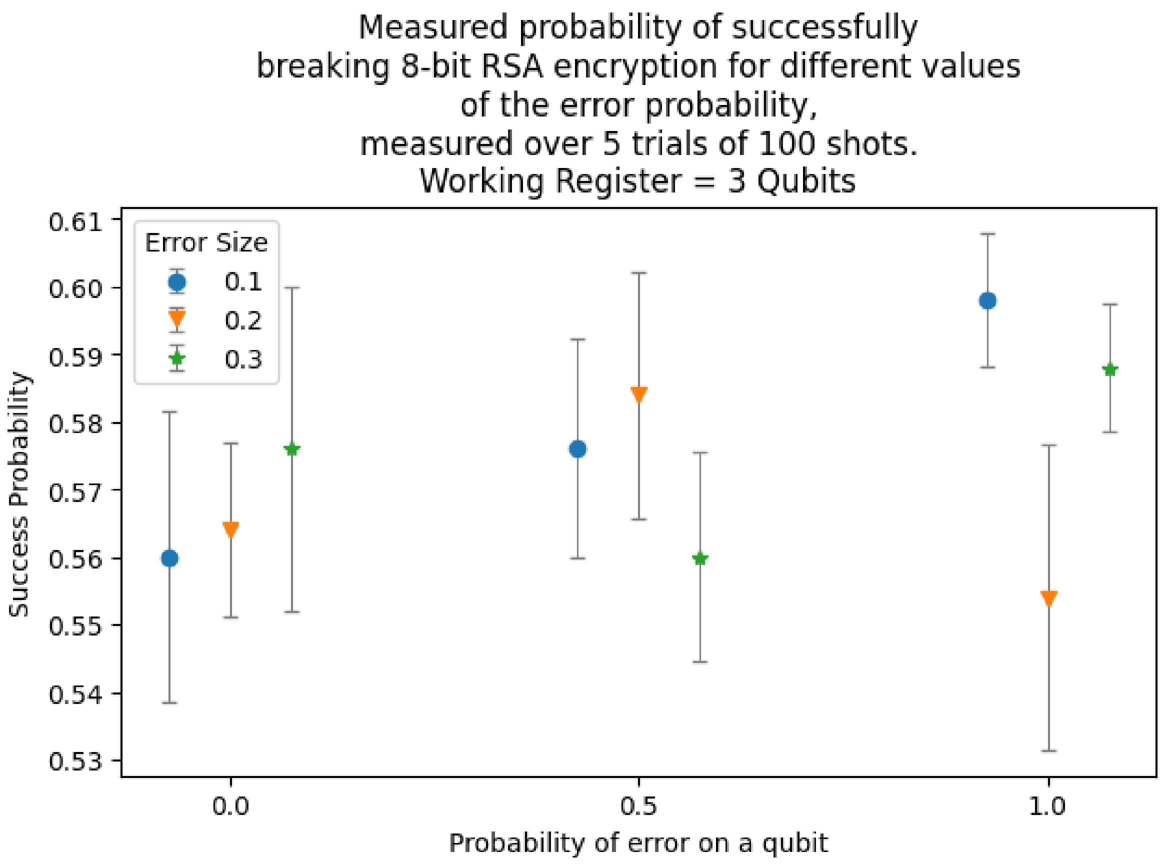
I have also included a discussion of how we dealt with *non-adjacent* multi-qubit gates through the use of SWAP gates.



(Figure 5: Two quantum circuits. The first applies a CNOT gate to qubits 1&2, then to 2&3, then to 1&3. Computing the third gate in this circuit is not as simple as computing its tensor product with the identity matrix as is done for the first and second gates. Our solution to this problem is to use SWAP gates to swap the first and second qubits, apply an *adjacent* CNOT gate and then swap the qubits back. This is shown in the second circuit diagram and it is functionally equivalent to the first.)

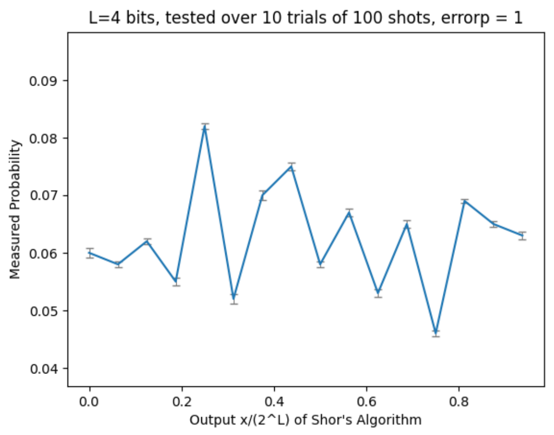
I have also improved the structural layout of the report. We need to restructure the discussion of Shor's Algorithm and rewrite the section on Shor's Algorithm as well. We also had a section of security of encryption algorithms with respect to quantum computers which for now has been put on hold as it was not very focused and needs refining (much of it would also make sense to go in our "conclusions" section as part of a wider discussion of our work).

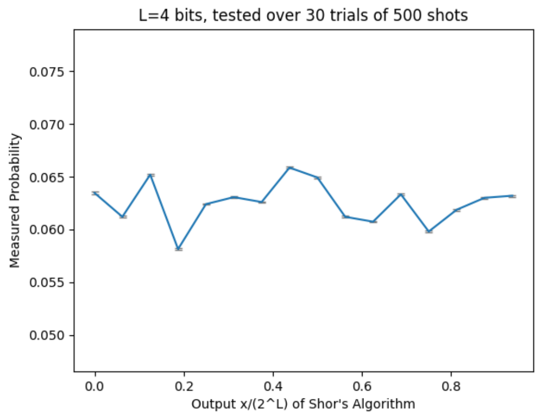
I have also continued testing our implementation of Shor's algorithm with respect to variation in error probabilities. Progress on this is quite slow unfortunately as the program is not optimised to run on a GPU as it was for Grover's Algorithm, which means each test takes quite a long time.



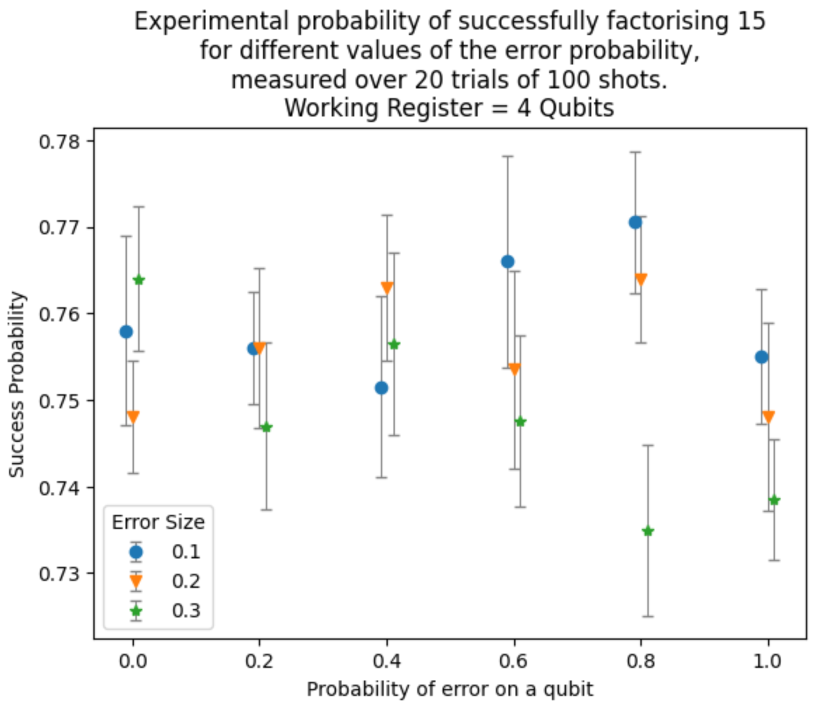
(Figure 6: Tests of the success probabilities for breaking RSA encryption with a working register of 3 qubits. Overall the probabilities are quite low, which is unsurprising since this is 8-bit RSA so a much larger working register would be ideal for it to work)

TOMORROW: CHECK STATUS OF L=4 JOB ON VIRTUAL DESKTOP.





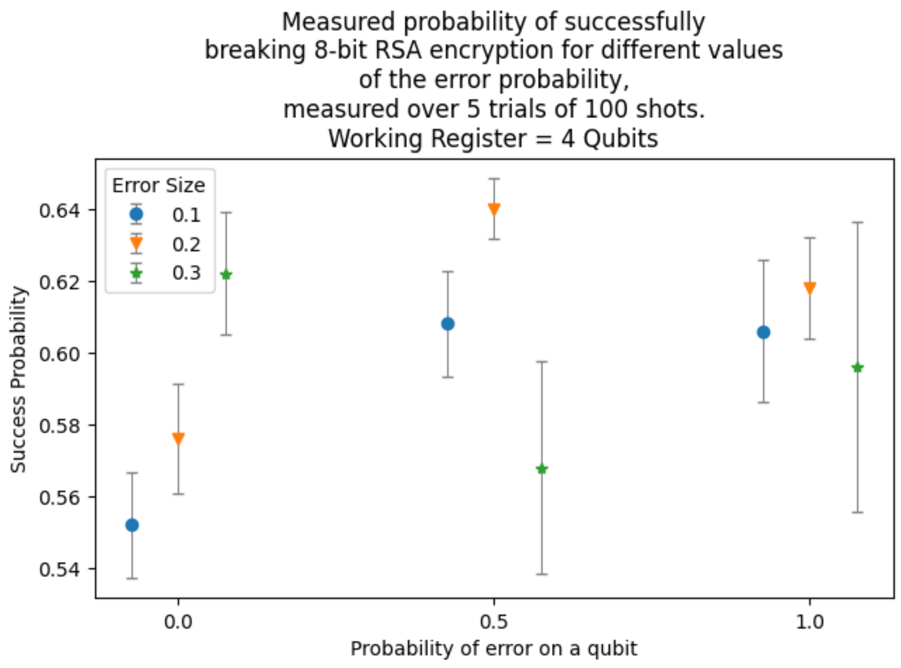
(Figure 7: (Left) Measured probabilities of different outputs of Shor's Algorithm for factoring N=15 for a working register size L=4. The probability of an error occurring on a qubit at every point in the circuit*, errorp*, is equal to 1 and the error_size is 10%. (Right) The same parameters but for errorp=0 as measured last week, albeit with a larger dataset.)



(Figure 8: A test of varying error when factorising N=15, but for L=4 qubits. There is more spread, but this may be an artifact of the lower number of trials (we don't know). Success probabilities are generally in the same region as L=3, around 75%)

10/03/2023

Yesterday evening around 7:10pm I submitted a job to the virtual desktop, which was a test for varying errorp when breaking RSA with 4 qubits. This finished at 11:22am this morning.



(Figure 9: Tests of the success probabilities for breaking RSA encryption for L=4 qubits in the working register for different values of errorp. Overall the rate of success is still not great - as expected for such a small working register, but we are limited here - but there is on average a 5-10% improvement over the 3-qubit version seen in figure 6).

I've also continued writing the report. Our introduction and simulation method discussion is now complete, as is the theory section for the grover search. Our results section for the grover search is now being worked on. The shor's algorithm theory section is also now written, but I need to take a second pass at what Sid and Sam have written. I also need to include the data I have gathered this week into the results section of the report.

Excellent! [illegible] 13/03