**Quantum Fourier Transform:**

The aim of QFT is to move a qubit state from the computational basis states to a Fourier state and angles in between these states.

Depending on the amount of input qubits and their states, we will place the qubits into Fourier states with specific phases applied onto them.

**Properties:**

1. Inverse QFT, will always return the original qubit states (wrote a test for this, works as intended)
2. The first qubit can be in the states after QFT algorithm has been applied, and the second can be in the states. The problem with this, is that it may be difficult to assert the phase of the qubits.

The phase of the output is dictated also by the length in qubits. QFT on qubits ‘1’ is different to QFT on ‘0001’.

The equation for the phase is where ‘’ is the real value in qubits, and N is the position of the qubit (starting from 1)

So, for ‘0001’ we expect the phases to be:

for the first qubit (on the left)

90 for the second

45 for the third

22.5 for the fourth

1. A simpler property that can be used to do with phase, no matter the qubit value, we know that the phase for each qubit must be a multiple of certain amount of **c,**

**c**

1. If the inputted qubits are all 0, then the phase of all qubits should be 0 **c**
2. The LSB qubit will always be less than

**Three similar implementations of the QFT algorithm**

|  |  |  |
| --- | --- | --- |
| **Qiskit** | **CirQ** | **Q#** |
|  |  |  |
| **Defining the function -** The input parameters are:  Circuit variable (circuit)  The number of qubits to use (n) | **Defining the function -** The input parameters are:  Circuit variable (circuit)  The qubit variables to use (qubits)  The number of qubits to use (n) | **Defining the function -** The input parameters are:  The qubit variables to use (qubits)  The number of qubits to use (n) |
|  |  |  |
| **The base case -** All three implementations use recursion, so we need a base case to stop the recursion.  When the n variable reaches 0, we know that the circuit has been completed, so we return the circuit. | **The base case –** In CirQ we need to apply the measurement gate to each qubit to get the results, so in the base case we apply a measure gate before returning the circuit. | **The base case –** In Q#, the concept of a circuit does not exist, we can only apply gates to qubits and measure them.  The method returns a list containing the measured results. |
|  |  |  |
| Apply the Hadamard gate on the circuit, on qubit n.  Before it reaches the base case, it will have applied the Hadamard gate to each qubit. | Apply the Hadamard gate to the **qubit object,** of the index n and add it to the circuit. | Apply the Hadamard gate to the qubit object of the index n. |
|  |  |  |
| In Qiskit we use the Controlled Phase gate to apply the rotation in the Fourier basis. CP =  This loop will add multiple gates, each applying a different rotation. The control qubit changes, but the target qubit will remain the same in this loop.  The Syntax for this gate is:  circuit.cp(rotation, control qubit, target qubit) | In CirQ we use the Controlled Z gate, as there is no Controlled Phase gate.  Fourier basis. CZ =  Notice that we do not multiply the numerator by , this is because = -1, essentially, they cancel out applying the same rotation.  The syntax for the gate is the same as qiskits’, but we need to remember to add it to the circuit. | In Q#, we use the R1 gate, which is identical to Qiskit’s Phase gate. They are just named differently in both languages.  However in Q# we need to add the controlled property before the R1 Operation.  This allows us to pass in a separate list of qubits to the R1 operation, to use as control qubits.  The syntax for this is:  (Controlled R1)([control qubit], (rotation, target qubit)); |
|  |  |  |
| Recursively call the method | 🡨 | 🡨 |

Aside from the recursive method calls, we need to do some setup in each programming language, to drive the program.

|  |  |  |
| --- | --- | --- |
| Qiskit | CirQ | Q# |
|  |  |  |
| We need to choose a simulator, Aer, provides with locally run simulators.  IBMQ can also be used, to run with physical quantum computers, but you need an account to set it up.  [More info](https://medium.com/qiskit/qiskit-backends-what-they-are-and-how-to-work-with-them-fb66b3bd0463) |  | We need to annotate an operation with @EntryPoint(), it will work as a “main” method for the namespace (class) |
|  |  |  |
| Set up the 4 qubit classical and qubit registers, and make a quantum circuit with those registers | In CirQ there are different types of qubits that you can declare (like gridqubits e.t.c)  We just use 4 LineQubits for CirQ, they seem the most appropriate.  Also, we create a circuit variable. | The ***use*** keyword allows us to initialise the array of qubits to use. |
|  |  | Calls the method that sets up the “circuit” (operations in Q#) |
| print(circuit) result: | print(circuit) result:  The circuit is identical as you can see. | There is no method to print a circuit in Q# ☹, I have tried to run the %trace magic command in jupyter, but it did not seem to work.  The next best thing that can be done is DumpMachine():  This allows us to see the probability distributions of the superposition state |
| job.result().get\_counts():  (There are more results but I cannot fit them in this table, they are roughly evenly distributed) | simulator.run(circuit, repetitions=10):  (The results are sorted by index, each comma separates a different qubit, and each value with a different index is from a different experiment or ‘shot’) | In the terminal you can use    To run the simulations |