KØBENHAVNS UNIVERSITET

Measurements in quantum simulations

Mikkel Willén (bmq419)
Supervisor: Martin Elsman

March 24, 2025

Abstract

This report presents extensions to the DQ quantum-circuit simulator to support projective measurements and explores their role in enabling quantum-classical interaction within an an SML host. We begin by motivating the need for measurement support. Without it, algorithm like Shor's algorithm and quantum error correction cannot be fully simulated, and classical control based on measurement outcomes remains out of reach. To address this gap, we have implement a suite of measurement primitives in DQ. This includes bit-extraction, probability computation, random sampling, state collapse and normalization, and integrates them into a higher-level routine for single measurements. We also demonstrate a fully parameterized Quantum Fourier Transform module and introduce a `Repeat` constructor for compact circuit repetition.

Our evaluation confirms expected quantum behavior in two targeted tests, that test the collapse of a Bell state entanglement and stability under consecutive measurements. We discuss the trade-offs of using SML as a classical host versus embedding a dedicated quantum language, and outline how measurement support lays the groundwork for simulating Shor's algorithm and simple quantum error correction codes. Finally, we identity avenues for future work. This includes multi-qubit measurements, full Shor algorithm implementation, QECC integration, a domain specific language for classical control flow, a monadic state-vector API, and noise modeling, to further close the gap between simulation and practical quantum-classical interaction.

Contents

1 Introduction

Quantum simulation frameworks let us develop and test quantum algorithms on a classical computer before running them on real hardware. Many important algorithms, such as quantum error correction routines and Shor's factoring algorithm, depend on measuring qubits partway through a computation and then using those measurement results to decide which operations to perform next. Without built-in support for projective measurements and the resulting state collapse, a simulator cannot show how these hybrid quantum-classical algorithm actually behave.

The DQ framework is a state-vector simulator written in Standard ML, and also includes a Futhark back end for higher performance. It offers a clear way to build and run quantum circuits, and to inspect exact amplitudes and probability values. However, up to now DQ only provides probability extraction. It does not let the user perform a measurement that collapses the state and returns a classical bit for further control flow. This gap prevents users from prototyping or validating any algorithm that needs mid-circuit measurements or measurement dependent branching.

To fill this gap, we have added a set of measurement function to DQ. These include functions to compute the the probabilities for a qubit, draw a random outcome according to those probabilities, collapse the state vector to the observed outcome, and normalize the result so it remains a valid quantum state. Furthermore, we have added a `Repeat` constructor, which repeats a circuit a specified number of times, and we have also implemented Quantum Fourier Transform. With these additions, a user can write SML code that measures a qubit, gets back a Boolean result, and uses that result in conditional statements. We state the following hypothesis:

Extending the DQ quantum simulation framework with projective measurement functionality will enable accurate and consistent simulation of quantum-classical interactions, such that measured outcomes can be used to influence classical control flow without deviating from the expected theoretical distribution and post-measurement state behavior.

The rest of this report is as follows. Section 2 provides background on the DQ framework and the theory behind quantum measurements, the Quantum Fourier Transform and Shor's algorithm. Section 3 describes how we implemented measurement support in DQ, and also how we implemented QFT and the `Repeat` constructor. Section 4 presents tests that verify correctness of state collapse, discusses the choice of using SML for classical control flow versus making a dedicated embedded language, and reflects on how measurements make it possible to simulate Shor's quantum subroutine and basic error correction codes. Section 5 outlines directions for future work, and sections 6 offers our conclusion.

The code for the project was attached as a zip file at handin, but can also be found at: `https://github.com/mikkelwillen/dq`

2 Background

2.1 DQ Quantum simluation framework

The DQ (DIKU Quantum) framework is a state-vector quantum circuit simulator. It supports simulation by representing quantum states as vectors of complex amplitudes and applying quantum gate operations as linear transformations on these state vectors. DQ offers multiple simulations modes, a semantics-based simulator, which directly uses the unitary matrices denoted by circuits, to a more optimized vectorized simulator, that applies gates as in-place operations on a global state vector [4]. Furthermore, DQ can leverage GPU acceleration by generating optimized data-parallel code using the Futhark language [4]. This allows users to run larger simulations on GPU hardware for improved performance. All these simulation modes produces equivalent results, ensuring that the framework can execute quantum circuits and obtain state vectors or outcome probabilities accurately regardless of the chosen backend.

DQ is implemented primarily in SML (Standard ML) and the core library is written only in SML and can be built with SML compilers such as MLKit or MLton [4]. In addition to the SML core, DQ

includes a Futhark library of quantum gates of operations, which can be invoked to execute circuits on parallel hardware. DQ provides an abstraction for quantum circuits that is modular. Circuits are defined using a combinator-based representation. Simple gate operations can be composed sequentially and in parallel to build more complex circuits. For example, one can construct a two-qubit circuit that applies certain single-qubit gates in parallel, using a tensor-product combinator, and then sequence that with two-qubit entangling gates, using a composition combinator, to form a complete circuit. This compositional approach is implemented directly in SML, allowing circuits to be treated as first-class values in the program. The framework cleanly separates the specification and visualization of circuits from their execution semantics [4]. In practice, this means a user can define a quantum circuit and obtain a visual diagram of it, before running any simulations.

The gate set supported by DQ encompasses the standard universal gates commonly used in quantum algorithms. This includes single-qubit gates, such as the Pauli X, Y and Y gates and the Hadamard H gate, as well as classical two-qubit gates, like the controlled-Not, denoted CX and the SWAP gate [4]. The identity gate is also available, enabling flexible circuit design, for example, to align qubit wires when composing circuits in parallel.

In addition to manual circuit construction, DQ offers tools for importing or converting circuits from external representations. It includes a utility to parse circuits described in the standard "QSIM" format and convert them into DQ's internal form or directly into Futhark code [4].

Once a quantum circuit is defined, DQ can evaluate the circuit on a given initial state and compute the resulting state vector or measurement outcomes. The framework allows initialization of arbitrary basis states. DQ calculates the amplitudes of each basis state in the output state vector, which can then be squared to yield the measurement probabilities. DQ does not provide functionality to perform sampling or repeated sampling, but instead focuses on exact state evolution and probability extraction.

2.2 Quantum measurement and state collapse

In quantum computing, measurement is the process by which a quantum state is converted into a classical outcome. According to the Born rule, the probability of obtaining a particular outcome upon measurement is given by the squared magnitude of the state's amplitude for that outcome [14]. For example, a single qubit in state $\alpha|0\rangle + \beta|1\rangle$ will yield the classical bit value 0 with probability $|\alpha|^2$ and 1 with probability $|\beta|^2$. Performing a measurement disturbs the quantum system. The wave function collapses to the eigen-state corresponding to the observed outcome [14]. In the qubit example, if the outcome is 0, the post-measurement state becomes $|0\rangle$, and similarly $|1\rangle$ if the outcome is 1. This collapse reflects the transition from a quantum superposition to a single classical reality as a result of observation.

The collapse means that after a measurement, the quantum state no longer contains information about the other possible outcomes. Repeated measurements of the same state are the needed to empirically estimate the probability distribution of outcomes. Each individual measurement yields one random outcome, and only over many trials can one recover the underlying probabilities $|\alpha|^2$ and $|\beta|^2$. Thus, a quantum measurement provides a single sample from the state's probability distribution and yields a classical result, irreversibly altering the state in the process.

2.3 Quantum Fourier Transform

In quantum computing, the Quantum Fourier Transform (QFT) is a linear transformation on quantum states analogous to the classical Discrete Fourier Transform (DFT) [10]. Acting on an n-qubit register, where the dimension is $N = 2^n$, the QFT maps each computational basis state $|x\rangle$ into a superposition of all basis states with Fourier-phase coefficients [10]. In particular the QFT performs the transformation

$$|x\rangle \mapsto \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{2\pi i \frac{xk}{N}} |k\rangle$$

which is exactly the DFT on the state amplitudes, up to normalization [10]. This change of basis is unitary, and if a quantum state encodes a periodic structure, the QFT causes interference that constructively amplifies amplitudes at the multiples of the period and cancels others, making the period observable upon measurement [11]. QFT is used in quantum algorithms that find hidden periodicities, most notably Shor's factoring algorithm and quantum phase estimation [10]. QFT can be performed efficiently on a quantum computer. It requires on the order of $O(n^2)$ one- and two-qubit gates for an n-qubit input [10], which is exponentially faster than the $O(N \log N)$ operations needed for a classical Fourier Transform on N data points, where $N = 2^n$ [10].

2.4 Shor's algorithm

Shor's algorithm, invented by Peter Shor, 1994, is a quantum algorithm for integer factorization [13]. Given a large composite number N, the algorithm finds its nontrivial prime factors efficiently, in polynomial time, whereas the best known classical factoring methods run in sub-exponential time, which is nearly exponential time[5]. The ability to factor large N quickly has profound implications for cryptography, as it would break RSA encryption. Shor's algorithm consists of a classical number-theoretic reduction combined with a quantum period-finding subroutine [13]. At a high level, it exploits the fact that factoring can be recast as finding the period of a certain modular exponential function. The steps below outline the algorithm's structure.

1. Pick a random integer a, such that $1 < a < N$ [13]. Compute $\gcd(a, N)$ using Euclid's algorithm. If $\gcd(a, N) \neq 1$, then we have found a nontrivial factor of N, and the algorithm can terminate [13]. Otherwise a is a co-prime to N, and we proceed.

2. Use a quantum computer to determine the period r of the function $f(x) = a^x \bmod N$. This means finding the smallest positive integer r such that $a^r \equiv 1 \pmod{N}$. The quantum circuit prepares a superposition of states encoding many exponents x, computes $f(x)$ in a second register, and then performs a QFT to reveal the period r via interference [13]. The period r can then be extracted from the output using the continued fractions algorithm on the measured outcome [13].

3. If the period r is odd or if $a^{r/2} \equiv -1 \pmod{N}$, then the procedure fails to find factors this round [13]. In this case we need to go back to step 1. If instead r is even and $a^{r/2} \not\equiv \pm 1 \pmod{N}$, then we can retrieve the prime factors. In fact, in this case one of $\gcd(a^{r/2}+1, N)$ or $\gcd(a^{r/2}-1, N)$ will yield a nontrivial factor of N, and similar for the other factor [13].

By repeating the above steps as needed, the correct period r, or the factors, will be obtained with high probability. The quantum speedup arises from the ability to find the period in step 2 exponentially faster than any known classical procedure. Overall, Shor's algorithm runs in time polynomial in $\log N$ or n [13]. Shor's approach can also be adapted to solve the discrete logarithm problem and other problems reducible to finding hidden periods [13].

3 Implementation

3.1 Measurements

To support projective measurements within the DQ quantum simulation framework, a collection of functions has been added that enables sampling from a quantum state's probability distribution and performing state collapse accordingly. These functions allows users to simulate measurements in the computational basis, return classical outcomes, and obtain updated quantum states consistent with the measurement results.

3.1.1 Bit extraction and complex scaling

First two helper function were made, that provide basic utilities. The first function, `bitAt`, extracts the bit value at a specific index in the state, by determining the qubit value.

```
fun bitAt (n, i) = (n div (IntInf.pow (2, i))) mod 2 = 1
```

Another utility function is `complexDivReal`, which divides a complex number by a real scalar, and is used for normalization of state vectors.

```
fun complexDivReal (z, r) = C./ (z, C.fromRe r)
```

3.1.2 Computing measurement probabilities

To calculate the probabilities of measuring 0 or 1 on a given qubit, the function `calcProbability` iterates over the entire state vector. For each index, it identifies the value of the measured qubit and adds the squared magnitude of the amplitude to the corresponding accumulator.

```
val prob = Math.pow (Complex.re (Complex.abs amplitude), 2.0)
val bit = bitAt (IntInf.fromInt idx, qubitIndex)
val (newP0, newP1) =
  if bit then (p0, p1 + prob) else (p0 + prob, p1)
```

This results in two totals `p0` and `p1`, which represents the probabilities of observing 0 and 1, respectively.

3.1.3 Sample based on probability

The function `sampleBits` generates Boolean values by comparing a pseudorandom number to the probablity threshold `prob` provided as an argument.

```
val randVal = R.random rng
val randValShifted = randVal * 100000.0 - Real.fromInt (Real.trunc (randVal
    * 100000.0))
val result = randValShifted < prob
```

This mechanism ensures variability in the simulated measurement results, reflecting the probabilistic nature of quantum computing.

3.1.4 State collapse

After sampling a result, we can collapse the quantum state to reflect the observed outcome. This is done by zeroing out all amplitudes that are inconsistent with the result.

```
val bit = bitAt (IntInf.fromInt idx, qubitIndex)
if bit = measuredBit then amplitude else C.fromRe 0.0
```

This process yields a project, but unnormalized quantum state.

3.1.5 State normalization

To maintain consistency, the collapsed state is normalized to ensure the aggregate probability is 1, meaning we have a valid quantum state.

```
val normSquared = Vector.foldl (fn (amp, acc) => acc + Math.pow (C.re (C.
   abs amp), 2.0)) 0.0 state
val norm = Math.sqrt normSquared
```

Each amplitude is then scaled by the inverse of the computed norm.

3.1.6 Single measurement

The function `measureQubit` perform single-shot measurement, and ties together probability computation, sampling, collapse and normalization.

```
val (_, prob1) = calc_probability stateSize state qubitIndex
val lst = sampleBits 1 prob1 rng
val measuredBit = List.hd lst
val collapsed = collapseState stateSize state qubitIndex measuredBit
val normalized = normalizeState collapsed
(measuredBit, normalized)
```

The function returns both the measured bit, as a boolean value, and the updated quantum state after collapse and normalization.

3.1.7 Repeated measurements

The function `measureNQubits` simulates repeated measurements on the same qubit and determines the most frequent outcome, which is then used for collapse.

```
val lst = sampleBits n prob1 rng
fun countBits (list: bool list) =
    List.foldl (fn (bit, (count0, count1)) =>
        if bit then (count0, count1 + 1)
        else (count0 + 1, count1)) (0, 0) lst

val (count0, count1) = countBits lst
val measuredBit = count0 < count1
```

Similarly, `measureNQubitsDist` performs repeated sampling and returns a list of boolean outomes without collapsing the state.

```
val list = sampleBits n prob1 rng
```

This simulates empirical estimation of measurement statistics, and does not affect the state vector.

It is worth noting, that on an actual quantum computer, repeated measurements cannot be peformed on the same quantum state, since the state collapses after the first measurement. Instead, the quantum circuit must be executed multiple times from the initial state, with each run producing one measurement outcome. The distribution is then estimated by aggregating results across many independent runs.

3.2 Quantum Fourier Transform

We have also implemented a functional implementation of the QFT using the DQ framework. The QFT is implemented as a reusable circuit module, parameterized by the number of qubits k and the classical integer input n to be encoded.

3.2.1 Utility functions

A number of utility functions are defined to help with circuit constructions, repetitions and readability.

The function `pow2 n` computes 2^n recursively.

```
fun pow2 n = if n = 0 then 1 else 2 * pow2 (n - 1)
```

The expression `repeatV n g` constructs a vertical repetition of gate g applied to n qubits. This is used to apply identity gates for circuit alignment and also for bulk applications of gates like Hadamard.

The `wrap i c j` function pads the circuit c with i identity gates on top and j below, to ensure its total height is uniform with the surrounding gates.

```
fun wrap i c j =
    if i < 0 orelse j < 0 then die "wrap"
    else if i = 0 andalso j = 0 then c
    else if i = 0 then c ** repeatV j I
    else if j = 0 then repeatV i I ** c
    else repeatV i I ** c ** repeatV j I
```

The functions `swap k i` and `swap k a b` are used to swap qubits. The first function, `swap k i`, swaps qubit i with its adjacent qubit below in a k-qubit circuit using a wrapped SW gate, while `swap k a b` swaps arbitrary qubits indices a and b by generating a sequence of adjacent swaps from a to b and back using the `swap` function.

```
fun swapN k a b =
  let
    fun recSwap acc up =
    if acc = a then
      swap k acc
    else if acc = b - 1 then
      swap k acc ++ recSwap (acc - 1) false
    else if up then
      swap k acc ++ recSwap (acc + 1) up
    else
      swap k acc ++ recSwap (acc - 1) up
  in
    if a = b - 1 then
    swap k a
    else
    swap k a ++ recSwap (a + 1) true
  end
```

The function `cntrl n g` recursively applies C to create a controlled gate with n control qubits

```
fun cntrl n g = if n <= 0 then g else C (cntrl (n - 1) g)
```

resulting in a circuit of height $n + 1$.

3.2.2 Register initialization

The classical input value n is encoded into an initial quantum state using a sequence of Pauli-X gates. Each bit of n determines whether an X gate should be applied to the corresponding qubit. The encoding proceeds from least to most significant bit to match QFT bit order.

```
fun encode_num k n =
  let
    fun enc i acc =
      if i >= k then acc
      else
        let
          val bit = (n div pow2 i) mod 2
          val gate = if bit = 1 then X else I
        in
          enc (i + 1) (gate ** acc)
        end
  in
    let
      val bit = n mod 2
      val gate = if bit = 1 then X else I
    in
      enc 1 gate
    end
  end
```

This produces a tensor product of X and I gates applied to an initial ket vector.

3.2.3 QFT phase rotations

The QFT itself is constructed from a layered application of Hadamard and controlled-R_z gates. The `qft_rots` function implements the standard decomposition. For each qubit, a Hadamard gate is applied, followed by a series of controlled phase shifts from that qubit to all lower-indexed qubits. Phase shifts are implemented as parameterized $R_z(\theta)$ gates, with angles decreasing exponentially by powers of two.

```
val theta = Math.pi / Real.fromInt (pow2 (j - i))
val rz = RZ(theta)
val ctrl = wrap i (cntrl (j - i) rz) (k - j - 1)
```

where the $R_z(\theta)$ gates is defined as follows:

$$R_z(\theta) = \begin{bmatrix} e^{-\frac{i\theta}{2}} & 0 \\ 0 & e^{\frac{i\theta}{2}} \end{bmatrix}$$

Each gate is wrapped with I gates to maintain uniform circuit height k, ensuring compatibility when composed with `++`.

3.2.4 Bit-reversal via swaps

The output of the QFT is in reversed bit order. This is corrected via a series of SW gates implemented in the `qft_swaps` function. The swaps are constructed so the each pair of qubits $(i, k-i-1)$ is swapped via a nested application of the `swapN` function.

```
fun qft_swaps k =
  let
    fun flip i acc =
      if i >= k div 2 then acc
      else
        let
          val s = swapN k (i + 1) (k - i) ++ acc
        in
          flip (i + 1) s
        end
  in
    let
      val s = swapN k 1 k
    in
      flip 1 s
    end
  end
```

This ensures the final output state has qubits in canonical order, so it fits with post-measurements expectations.

3.2.5 Circuit composition

The full QFT circuit is composed by first applying the encoding circuit and then the QFT rotations and swaps.

```
fun qft_circuit k n =
  let
    val init = encode_num k n
    val qft = qft_rots k ++ qft_swaps k
  in
```

```
    init ++ qft
  end
```

This is outputs the full circuit for QFT, to be used in the DQ framework.

3.3 The repeat constructor

In the DQ framework, the `Repeat` constructor has been implemented for repetition of a subcircuit multiple times in sequence. It is defined by the type `Repeat of int * t`, where `n` is the number of repetitions and `t` is the subcircuit to be repeated. This is useful for expressing repeated quantum operations, such as the oracle and diffusion operations in Grover's algorithm.

For visualization and debugging purposes,the `Repeat` construct is integrated into the pretty-printing logic of circuits. When pretty-printing a repeated subcircuit, it is represented in the form `n x [t]`, indicating that the subcircuit `t` is repeated `n` times.

```
| Repeat(n,t) => maybePar (p > 5) (Int.toString n ^ " x [" ^ pp 5 t ^ "]")
```

This enhances readability, especially when dealing with circuits containing large numbers of repeated blocks.

To produce a visual layout of a repeated circuit, the `Repeat` case is implemented for the standard rendering function.

```
| Repeat(n,a) => Diagram.rep (n, height a, dr a)
```

This renders the subcircuit within vertical bars, and before the left most, top middle qubit, the number of repetitions are printed. The latex rendering has not been implemented yet.

In the semantics layer, the `Repeat` operator is implemented as repeated matrix multiplication. This reflect the sequential application of the same unitary operation `n` times.

```
| Repeat(n, t) =>
    let val m = sem t
    in foldl (fn (_, a) => matmul(m, a)) m (List.tabulate(n - 1, fn _ => m))
   )
    end
```

Here, the semantic function `sem t` evaluates the matrix representation of the subcircuit `t`. The initial matrix is then repeatedly multiplied by itself $n-1$ times to produce the full transformation of the repeated block. This ensures that the repeated circuit behaves identically to `t` applied `n` times in succession.

4 Evaluation and discussion

4.1 Measurements

Quantum measurement has been implemented by calculating the outcome probabilities using the Born rule, and then simulating the act of measurement by random sampling and state collapse.

4.1.1 Test

To test the correctness of our measurement implementation in DQ, we created two small experiments that check important properties of quantum measurements. These tests are not based on collecting large amounts of data, but on verifying expected behavior in quantum circuits.

Entanglement and measurement

The first test, in `entangle_ex1.sml`, checks whether measurement correctly collapses entangled states. The program creates a Bell state using a Hadamard on the first qubit followed by a CNOT gate:

```
val () = run ((H ** I) ++ (C X))  (ket[0,0])
```

This prepares the state $(|00\rangle + |11\rangle)/\sqrt{2}$, where the two qubits are correlated. When we measure qubit 0, we expect qubit 1 to always have the same value. The test confirms this. Measuring the first qubit yields either 0 or 1, and the distribution printed from that resulting state show that only $|00\rangle$ or $|11\rangle$ respectively, remain.

Repeated measurement

In the second test, in `measure_ex2.sml`, we check that measuring the same qubit twice gives the same result. In the program, we prepare a 4-qubit state and then measures the same qubit twice in a row.

```
fun run c k =
     let val v0 = init k
         val v1 = eval c v0
         val (bit, newState) = measureQubit v1 2
         val (bit2, newState2) = measureQubit newState 2
     in print ("State after first measurement:\n" ^ pp_dist (measure_dist
    newState) ^ "\n\n")
        ; print ("State after second measurement:\n" ^ pp_dist (measure_dist
     newState2) ^ "\n\n")
     end)

val () = run ((H ** H ** H ** H) ++ Repeat(2, T ** Z ** T ** I)) (ket
    [1,0,1,0])
```

Here we measure qubit 2, then measure it again. Accoding to quantum mechanics, after the first measurement the qubit should be in a definite state, and the second measurement should always return the same result. Our implementation behaves as expected, since the first measurement collapses the state, and then second gives the same bit every time. The printed distributions before and after confirm that no further change occurs after the first measurement.

Discussion

These two tests show that the measurement implementation behaves as expected in basic scenarios. The entanglement test checks that the result of one qubit affects the other, as it should in entangled states. The second test confirms that once a measurement collapses a qubit, repeated measurement return the same value.

We have not included dedicated tests for the sampling functions, as such tests would primarily evaluate the behavior of the underlying random number generator, rather than the correctness of our quantum measurement logic. Since sampling involves inherent randomness, verifying its behavior is challenging, and since fluctuations in the output are expected, it does not necessarily indicate a bug. To properly test sampling statistically, we would need to run a large number of trials on both the sampling functions and the random number generator and compare the results of the two.

While the initial tests confirm that our measurement implementation behaves as expected in simple and controlled cases, further testing is needed to build confidence in its correctness and robustness. The current tests do not cover edge cases, more complex entangled states or interactions with larger circuits and intermediate measurements. Future tests could include randomized circuits with known expected measurement distributions, checking statistical outcomes over many runs, or we could also introduce test cases that involve partial measurements in larger entangled systems, such as simulating Shor's algorithm or quantum error correction, both described below.

4.2 Classical SML host vs embedded quantum language

One important design aspect of measurements in DQ, is that all of SML is used as the classical host language. All control flow, loops, conditionals, and classical data structures exist in the SML layer, separate from the quantum state that SML functions operate on. This reflects a quantum coprocessor model, where a classical computer requests quantum operations and reads results.

Using SML directly for classical computations has some advantages. First, we get rich control structures, like conditionals, pattern matching, recursion/loops, and we get data types, modules and libraries "for free", without having to re-implement these features in an embedded language. For example, to apply a quantum operation conditionally based on prior measurement result, we can simply use an SML `if` on the boolean result and call the appropriate quantum function in each branch. A lot of current quantum programming systems use classical logic in this way, where measurements are handled classically, and subsequent quantum operations are chosen accordingly [12].

Another benefit is flexibility and clarity. Since DQ the measurement functionality in DQ are just SML functions, like `measureQubit` returning a bool and a new state, the user can compose them with any other SML functions. All classical data, like random number generators, can be managed with existing tools.

The pure host-language approach also has some downsides, particularly when considering more advanced use cases or ensuring quantum-specific properies. For example, since we do not have a dedicated quantum language layer with its own type system, it is much more difficult to enforce constraints like linearity or no-cloning of quantum data. The quantum state in DQ is represented as a plain `Vector` of complex amplitudes, and SML's type system does not inherently prevent a user from e.g. storing two copies of a `state` or reusing an old state after measurement, or new operations on the state. An embedded quantum language could introduce type constructors or qualifiers to enforce linear usage, to ensure a quantum state has only one reference.

Another drawback is that by using SML's own control flow for everything, we lack the ability to represent a quantum program's structure as a first-class object. This makes it harder to analyze and optimize quantum programs. If we write an SML function with an `if` that calls some quantum operations, the branching is happening on the fly during execution, and we do not have an intermediate representation of a quantum circuit with a conditional. This means we cannot easily apply transformation or optimizations on the whole program. An embedded language approach could define an AST or monadic representation for quantum computations. Such an embedded DLS could also allow constructing a tree of operations, where classical control structures are nodes in that tree, instead of executing them immediately.

If we want to make DQ an embedded language with SML, we would likely introduce type constructors for classical control-flow and other classical operations. We could create a type called `QProgram` and add constructors like `QIF(cond_fn, then_block, else_block` and `QWhile(cond_fn, block`, where `cond_fn` specifies how a classical condition is obtained, like from a measurement. This would allow us to treat the entire quantum-classical program as data, so we could perform optimizations or translate it to other forms, like for the Futhark language. Some frameworks provide library functions to inject classical conditions into an embedded circuit description [2]. These work in a simliar way to what a `QIf` constructor in an AST would do. They produce a new quantum program fragment that includes a branch condition on a measurement result, rather than using the host language's `if`.

Thus our current approach prioritizes simplicity and direct execution, at the cost of not having static quantum type safety, and having a much harder time with program analysis.

4.3 Shor's algorithm

Shor's algorithm is a quantum algorithm for factoring an integer N by exploiting a quantum method to find the period of a function related to N [13]. To implement Shor's algorithm, we need both measurements and QFT among other things.

4.3.1 Simulating the Quantum subroutine in Shor's algorithm

If we were to implement Shor's algorithm, we would first allocate two quantum registers. The first register needs enough qubits to represent a sufficient range of exponents. If N is an n-bit number, we need $2n$ qubits in the first register to encode values from 0 to $2^{2n} - 1$, since we need enough

binary digit to approximate a fraction with denominator up to N. The second register, should have n qubits, enough to represent values module N. We initialize the first register to $0 \dots 0\rangle$ and the second register to $|1\rangle$, since $a^0 \mod N = 1$. Next we apply Hadamard gates to all qubits in the first register, creating a uniform superposition of all possible exponent states[13].

The next step is to compute $a^x \mod N$ for the superposed exponents x and store the reulst in the second register. This is implemented by a series of controlled operations from the first to the second register. Essentially, we implement a unitary U that performs

$$U \,:\, |x\rangle_1|0\rangle_2 \mapsto |x\rangle_1|a^x \bmod N\rangle_2$$

This can be done by breaking x into bits and performing modular exponentiation by repeated squaring[13]. Now the state is in an entangled superposition, where all the amplitudes are equal in magnitude, but the values in the first register that lead to the same $a^x \mod N$ in the second register are now linked. These values x form arithmetic progressions separated by the unknown period r. For example, if $a^x \mod N$ yields a value y, then all x congruent modulo r with yield the same y. This means the first register's state is periodic with period r, up to a phase [13].

We know apply the inverse QFT to the first register. The inverse QFT concentrates amplitude onto the basis states that correspond to multiples of the reciprocal of that period. If the first register, of m qubits, spans integers $0 \dots 2^m - 1$, then after a inverse QFT, the state of the first register will be most likely found in one of the basis states $|k\rangle$ that are closest to an integer multiple of $\frac{2^m}{r}$.

When measured, the output of QFT is, with high probability, a classical value k that is near to $\frac{j}{r}2^m$ for some random integer $j \in 0, 1, \dots, r-1$ [13]. We then apply the continued fractions algorithm to approximate the ratio $\frac{j}{r}$ and thus solve for the unknown period r [13], by finding a fraction $\frac{j}{r} \approx \frac{k}{2^m}$ in the lowest terms. Once a candidate r is obtained, the final step is to attempt factoring. First we check if r is even, and if so, compute $g = \gcd(a^{r/2} - 1, N)$. If g is nontrivial, then g and $\frac{N}{g}$ are the factors, and we are done. If not, the quantum subroutine has failed, and we need to pick a new random a [13].

Shor's algorithm demonstrates why implementing measurements is useful. The algorithm's quantum portion prepares a superposition encoding the answer in the relative phases of the amplitudes. However, the result remains encoded in a quantum state and must be extracted via measurement. After the inverse QFT, the desired information is embedded in the probability distribution of the first register's basis states. We need measurement to collapse the wavefunction and yield a classical bit-string with the result. Without the ability to measure qubits, the factors of N would remain inaccessible in a superposed form.

4.4 Quantum error correction

Measurements are important in both quantum error correction (QEC) and quantum error correcting codes (QECC), as measurements allow extraction of syndromes without disturbing the logical quantum information, and thus, we will look into how to implement both in DQ.

4.4.1 Sources of error in quantum computers

Quantum hardware is inherently susceptible to various forms of noise and imperfections. These errors typically arise from decoherence, imperfect gate operations and measurement inaccuracies.

Decoherence occurs as qubits lose their quantum coherence over time due to uncontrolled interactions with the environment. This manifests as relaxation, which is energy loss from excited to ground state, and dephasing, which is randomization of relative phase. This means the qubit's state decays or randomizes, behaving as if it becomes entangled with the environment [1]. Decoherence sets a finite coherence time during which quantum information can reliably be measured.

Applying quantum logic gates is an analog physical process, and imperfections in control pulses or calibration cause gates to enact slightly wrong rotations or entangle the qubit with unwanted degrees of freedom, leading to gate infidelity. A gate error occurs when a gate changes the qubit

state incorrectly, typically quantified by a gate fidelity, meaning the probability of the gate acting without error [1]. Crosstalk between qubits during multi-qubit operations can also induce gate errors on neighboring qubits [1].

Measuring a qubit is also prone to error. A qubit in state $|0\rangle$ might be reported as "1" or vice versa, with some probability due to noise in the readout apparatus [1]. Such measurement error rates are significant in current devices, meaning the classical output bits can be wrong even if the qubit's state was correct before measurement.

These physical processes typically induce logical errors in qubits that can be abstracted by simple error channels. The most prominent error types types are bit-flip and phase-flip errors, as well as their combinations. A bit-flip error, corresponding to a Pauli X gate, flips a qubit's state $|0\rangle \leftrightarrow |1\rangle$, while a phase-flip error, corresponding to a Pauli Z gate, flips the sign of the $|1\rangle$ component, sending $|1\rangle$ to $-|1\rangle$ but leaving $|0\rangle$ unchanged [1]. More general noise can be modeled by depolarizing errors, in which a qubit's state is randomized to any other state. In a depolarizing channel [7], with some probability p the qubit undergoes a random Pauli X, Y or Z error, each with probability $\frac{p}{3}$, and with probability $(1-p)$ it remains correct [9]. Depolarizing noise captures the net effect of many small error sources, effectively shrinking the qubit's Bloch sphere towards a mixed state.

4.4.2 Error mitigation

Because quantum computers have to deal with the above errors, QEC is used to actively detect and correct errors, mitigating their effect on the computation. The general idea of QEC is to encode a single logical qubit of information into a larger entangled state of multiple physical qubits, in such a way that if some of the qubits suffer errors, the collective logical information can still be recovered [1]. Unlike classical error correction, where one can directly copy bits and perform a majority vote, quantum computing must follow the no-cloning theorem, which states, that it is impossible to create an identical and independent copy of an arbitrary unknown quantum state [6]. Thus, quantum error correction uses syndrome measurements on ancillary qubits to gain information about errors without measuring the data qubits' logical state [9]. An error on any one physical qubit leaves a detectable mark in the syndrome bits, which can then be used to infer and correct the error without destroying the quantum state [8] [1].

QECC come in many forms, but most are based on the same principle of distributing quantum information across multiple qubits and using multi-qubit measurements to detect errors. Typically, a QECC involves three conceptual steps; encode the logical qubits into an entangled multi-qubit state, periodically detect whether an error has occurred, via syndrome measurements, and if so, correct the error by a recovery operation. Most QECC use stabilizers to perform error correction, where a set of commuting multi-qubit observables, the stabilizers, are periodically measured. Each stabilizer check yields a binary syndrome bit, ± 1 eigenvalue, indicating whether an error has flipped the eigenvalue of that stabilizer [1]. The combined collection of syndrome bits pinpoint the type and location of error, after which a targeted correction operation, Pauli X, Y or Z on a specific qubit, is applied to reverse the error [1]. Since Pauli gates are their own inverses, applying, e.g., and X gate will undo an X error [1]. In this way, as long as errors are sufficiently infrequent, the logical qubit can be restored to its intended state, thus prolonging the effective coherence of quantum information.

While QEC is important for scalable quantum computing, it comes with a significant overhead. First of all, QECC requires logical qubits to be encoded into multiple physical qubits, leading to an constant factor increase in the number of physical qubits needed. This overhead is also in the number of required gate operations and measurement rounds, all of which must be performed with high fidelity, meaning the error rates of physical gates and measurements must be below a certain threshold to be effective. This threshold exists because QEC itself introduces additional operations, which can themselves introduce further errors. If the physical error rate is too high, the process of error correction may introduce more errors than it corrects.

4.4.3 Simulating errors in the DQ framework

When working with the DQ quantum circuit simulator, we can model the effect of noise by explicitly introducing error operations into the circuit. This could be done in several ways, detailed below.

After certain gates or at fixed intervals in the circuit, we could insert a Pauli X gate on a qubit with some probability p to simulate a bit-flip error on that qubit. We could e.g. generate a random number, between 0 and 1, for each gate application and, if its below p, we apply an X gate to that qubit in the circuit. Similarly, we can insert Pauli-Z gates with some probability on qubits to simulate phase flips. Both bit-flip and phase-flip error channels can be applied to any qubit at any point. In practice, we probably want to apply these probabilistic errors right after each logical gate in the circuit, to reflect the idea that each gate has some chance to introduce error. We could also simulate a more general depolarizing channel of strength p, where we apply no error with probability $(1-p)$ or one of X, Y or Z each with probability $\frac{p}{3}$, as mentioned earlier. This would cause the qubit's state to drift toward the maximally mixed state over many runs. We can also post-process measurement outcomes to simulate readout errors. After the measurement, we can flip each output bit to the opposite value with some error probability p, corresponding to known readout error rates.

By implementing these randomized error gates and outcome flips, the DQ simulator can produce outcomes that resemble what real quantum computer might output. These simulated error provide a testbed, where we can apply errors randomly and average over many runs with different error seeds, thereby estimating the overall error rates and algorithm performance under noise.

4.4.4 Simulating error corrections

In the DQ simulator, we can also simulate QECC, to show how an encoded logical qubit can be protected by detecting and correcting errors. A basic implementation of QECC in DQ could be as follows.

First we would add extra qubits to encode a logical qubit. For example, to implement the 3-qubit bit-flip code, we start with three qubits all initialized to $|0\rangle$. We then assume some arbitrary logical state is applied to the data qubit, which is the first qubit, and we then apply two CNOT gates to entangle this qubit with the two auxiliary qubits. The result is an encoded state, which represents the single logical qubit in a distributed form [1]. In DQ, assuming q_0 is the data qubit, this means applying CNOT($q_0 \rightarrow q_1$) and CNOT($q_0 \rightarrow q_2$). This encoding spreads the quantum information across three physical qubits, so that an error on any one of them can later be detected and corrected by examining the joint state.

We then assume an error has been introduced, as explained in the section above. If for instance, an X error is inserted on qubit 0, the state in our example would flip from $\alpha|000\rangle + \beta|111\rangle$ to $\alpha|100\rangle + \beta|011\rangle$, and we have a state where qubit 0 is now out of sync with the others.

After the error has occurred, the next step is to measure the syndrome bits. This is done by adding two ancillary qubits to detect errors in the three data qubits. The first ancilla qubit measures the the parity z_0z_1 via CNOT with qubit 0 and qubit 1, where as the second ancilla qubit measures the parity z_0z_2.

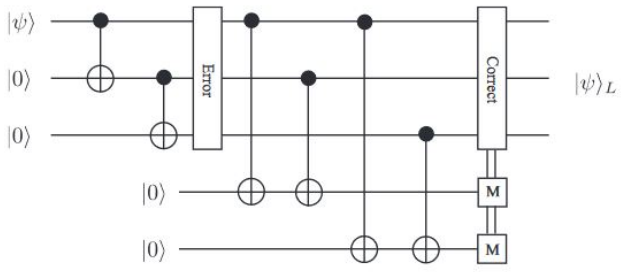

Figure 1: Circuit for the 3-qubit code [3]

These parity check measurements constitute the syndrome. This is implemented simply, by adding

two ancilla qubits initialized to $|0\rangle$, then applying CNOT gates from each data qubit into the ancilla, and finally measuring the ancilla qubits. The outcome yields two classical bits, that indicate which qubit, if any, is erroneous. For example, a measurement result of 01 suggest that the third data qubit has experience a bit-flip error, as shown in figure 2. The encoded logical information is not directly measured and thus is preserved, only the parities are revealed by the ancillas.

Ancilla measurement	Collapsed state	Consequence
00	$\alpha\|000\rangle + \beta\|111\rangle$	No error
01	$\alpha\|001\rangle + \beta\|110\rangle$	X on qubit 3
10	$\alpha\|010\rangle + \beta\|101\rangle$	X on qubit 2
11	$\alpha\|100\rangle + \beta\|011\rangle$	X on qubit 1

Figure 2: Measurements and their corresponding error [3]

Based on the syndrome measurement outcomes, a conditional correction is applied to the affected qubit, to restore the logical state. In the DQ framework, we can implement this classically by branching on the measured syndrome bits and modifying the circuit. Using the example from before, where we measure 01, we would apply an X on the third qubit to flip it back. Assuming no errors has happened on the ancilla qubits, this would recover the logical qubit.

4.4.5 Measurements for QEC

Measurements are important to QEC and QECC because they enable the detection of errors without collapsing the encoded quantum information. QEC relies on syndrome measurements, which extract information about whether and where an error has occurred by measuring stabilizers. Syndrome measurements are implemented via ancilla qubits entangled with the data qubits and then measured in the computational basis. The outcome determines the recovery operations needed to restore the correct encoded state. Without the ability to perform such measurements, QECC cannot operate, as no error information could be extracted. Thus, measurements in simulation frameworks like DQ are needed for implementing and evaluating QEC schemes.

5 Future Work

Although this work adds core measurement primitives and QFT support to DQ, several important extensions remain. In the following, we outline key directions, ranging from noise modeling to full algorithm implementations and type safe quantum data handling, that would broaden DQ's abilities.

5.1 Multi-qubit measurements

Extending the measurement API to handle simultaneous collapse of multiple qubits will allow us to model entangled syndrome extraction and parity checks more directly. This requires computing joint outcome probabilities, sampling from a larger outcome space, and collapsing the state vector across multiple indices in one operation

5.2 Implementing Shor's algorithm

With measurements and QFT in place, the next step is to implement the full Shor subroutine. This includes register allocation, modular exponentiation, inverse QFT and the continued fractions algorithm. Although we have discussed these steps theoretically, a concrete implementation in DQ will validate our measurement support further.

5.3 Noise models

Add built-in support for common noise channels, like depolarizing and dephasing, or a density matrix simulator to model decoherence and gate errors natively. This will let us study algorithm and error correction performance under realistic hardware conditions.

5.4 Implementing the 3-qubit QECC

A minimal quantum error correction code will test both multi-qubit measurements and conditional operations. By encoding and injecting errors, measuring syndromes on ancilla registers, and applying recovery gates, we can evaluate the interplay of measurement collapse and classical feedback.

5.5 Creating a DSL for classical control flow

Currently, classical decisions after measurements are written as SML conditional statements. Embedding these in a domain specific language, will let us capture complete quantum classical programs as data. This is a prerequisite for program analysis, optimizations and Futhark code generation that respects dynamic circuits.

5.6 Making a monadic structure for the state vector

Wrapping the state vector and measurement operations in a monad would enforce linear usage and sequencing, reduce boilerplate and enable chaining quantum operations with classical effects in a principled way. This abstraction can also track dependencies and prevent misuse of states after collapse or additional operations.

5.7 Full `Repeat` support

Although `Repeat` works in the pretty-printer and the main simulator, extending its LaTeXrenderer and implementing it in `execute` and `execute2` is still missing. This currently renders some functionality of the DQ framework unusable, while using `Repeat`.

5.8 Expanding the test suite

Beyond the two basic tests, we need end-to-end tests for larger circuits, statistical checks of outcome distributions, edge cases and integration tests for measurement driven program, such as Shorøs algorithm and QECC.

6 Conclusion

In this work, we have extended the DQ quantum simulation framework to support projective measurements in the computational basis. By adding functions for probability calculation, random sampling, state collapse and normalization, we enable users to perform single and repeated measurements on state vectors. Our implementation was validated through two targeted tests. Collapsing an entangled Bell state and verifying consistent outcomes under back-to-back measurement. These tests confirm that measurement outcomes in DQ match theoretical expectations and that collapsed states persist across measurements.

We also sketched how to integrate the measurement primitives into higher-level examples, such as Shor's period finding subroutine and quantum error correction codes, which demonstrates how measurement results could feed back into classical control flow. Although we have not fully implemented Shor's algorithm or quantum error correction codes, the existing measurement support lays the groundwork for these next steps. The QFT module, combined with projective measurement, show how quantum classical interactions can be modeled and tested within DQ.

Our findings support the central hypothesis: "Extending the DQ quantum simulation framework with projective measurement functionality will enable accurate and consistent simulation of quantum-classical interactions, such that measured outcomes can be used to influence classical control flow without deviating from the expected theoretical distribution and post-measurement state behavior." The agreement between single-shot measurement outcomes and the predicted collapse behavior, together with the stability of collapsed states under repeated measurement, shows that DQ now correctly captures both the inherent randomness and the irreversible state collapse of quantum measurement.

Despite these successes, there remain limitations. Current measurement routines operate only on single qubits at a time, and the framework does not yet support conditional branching inside circuit definitions. Future work could address multi-qubit measurements, integration with Shor's algorithm and QECC circuits, and richer classical control abstractions, like a monadic or DSL-based structure. Fully implementing `Repeat` functionality and expanding the test suite will further improve confidence in DQ's measurement implementation.

In summary, we have closed a gap in the DQ simulator by adding measurement and collapse mechanics, and demonstrated their correctness and utility. This positions DQ for more advanced experiments in quantum algorithms and error correction, brining us closer to a robust tool for exploring quantum-classical worksflows.

References

[1] Avimita Chatterjee, Koustubh Phalak, and Swaroop Ghosh. "Quantum Error Correction For Dummies". In: (). URL: https://arxiv.org/pdf/2304.08678.

[2] *Classical feedforward and control flow (dynamic circuits) — IBM Quantum Documentation.* URL: https://docs.quantum.ibm.com/guides/classical-feedforward-and-control-flow.

[3] Simon J Devitt, William J Munro, and Kae Nemoto. "Quantum Error Correction for Beginners". In: (). URL: https://arxiv.org/pdf/0905.2794.

[4] Martin Elsman. *diku-dk/dq: The DIKU Quantum Simulator Framework.* URL: https://github.com/diku-dk/dq.

[5] *Integer factorization - Wikipedia.* URL: https://en.wikipedia.org/wiki/Integer_factorization.

[6] *No-cloning theorem - Wikipedia.* URL: https://en.wikipedia.org/wiki/No-cloning_theorem.

[7] *Quantum depolarizing channel - Wikipedia.* URL: https://en.wikipedia.org/wiki/Quantum_depolarizing_channel.

[8] *Quantum error correction - Wikipedia.* URL: https://en.wikipedia.org/wiki/Quantum_error_correction.

[9] *Quantum Errors and Quantum Error Correction (QEC) Methods.* URL: https://postquantum.com/quantum-computing/quantum-error-correction/.

[10] *Quantum Fourier transform - Wikipedia.* URL: https://en.wikipedia.org/wiki/Quantum_Fourier_transform.

[11] *Quantum Fourier Transform (QFT).* URL: https://postquantum.com/quantum-computing/quantum-fourier-transform-qft/.

[12] *Quantum programming - Wikipedia.* URL: https://en.wikipedia.org/wiki/Quantum_programming.

[13] *Shor's algorithm - Wikipedia.* URL: https://en.wikipedia.org/wiki/Shor%27s_algorithm.

[14] *Why probability in quantum mechanics is given by the wave function squared.* URL: https://www.preposterousuniverse.com/blog/2014/07/24/why-probability-in-quantum-mechanics-is-given-by-the-wave-function-squared/.