SQUANCH Documentation

Release 1.1.0

Ben Bartlett

Aug 21, 2018

CONTENTS:

CHAPTER

ONE

OVERVIEW

1.1 A Simulator for Quantum Networks and Channels

SQUANCH (Simulator for QUAntum Networks and CHannels) is an open-source Python framework for creating performant and parallelized simulations of distributed quantum information processing. Although it can be used as a general-purpose quantum computing simulation library, SQUANCH is designed specifically for simulating quantum *networks*, acting as a sort of "quantum playground" to test ideas in quantum transmission and networking protocols. The package includes flexible modules that allow you to intuitively design and simulate a multi-party quantum network, extensible quantum and classical error models which introduce realism and the need for error correction in your simulations, and a multi-threaded framework for manipulating quantum information in a performant manner.

SQUANCH is developed as part of the Intelligent Quantum Networks and Technologies (INQNET) program, a collaboration between AT&T and the California Institute of Technology. The source is hosted on GitHub.

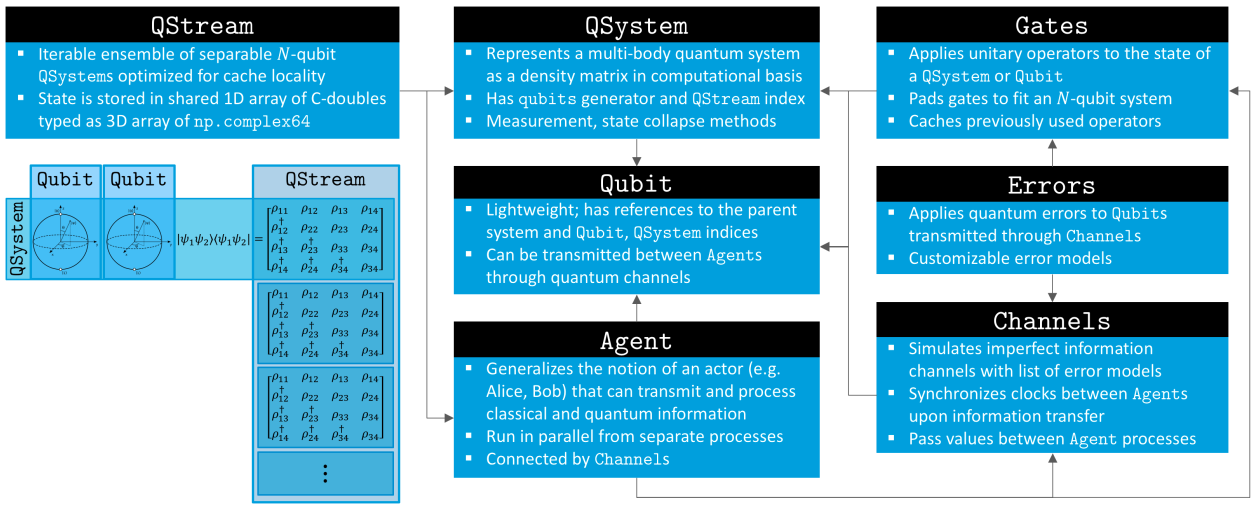
1.1.1 SQUANCH Whitepaper

We encourage interested users to read the whitepaper for the SQUANCH platform, "A distributed simulation framework for quantum networks and channels" (arXiv: 1808.07047), which provides an overview of the framework and a primer on quantum information.

1.1.2 SQUANCH Manual

This documentation is also available in PDF format here.

1.2 Design Overview



QStream

- Iterable ensemble of separable N-qubit `QSystems` optimized for cache locality
- State is stored in shared 1D array of C-doubles typed as 3D array of `np.complex64`

QSystem

- Represents a multi-body quantum system as a density matrix in computational basis
- Has `qubits` generator and `QStream` index
- Measurement, state collapse methods

Gates

- Applies unitary operators to the state of a `QSystem` or `Qubit`
- Pads gates to fit an N-qubit system
- Caches previously used operators

Qubit

- Lightweight; has references to the parent system and `Qubit`, `QSystem` indices
- Can be transmitted between `Agents` through quantum channels

Errors

- Applies quantum errors to `Qubits` transmitted through `Channels`
- Customizable error models

Agent

- Generalizes the notion of an actor (e.g. Alice, Bob) that can transmit and process classical and quantum information
- Run in parallel from separate processes
- Connected by `Channels`

Channels

- Simulates imperfect information channels with list of error models
- Synchronizes clocks between `Agents` upon information transfer
- Pass values between `Agent` processes

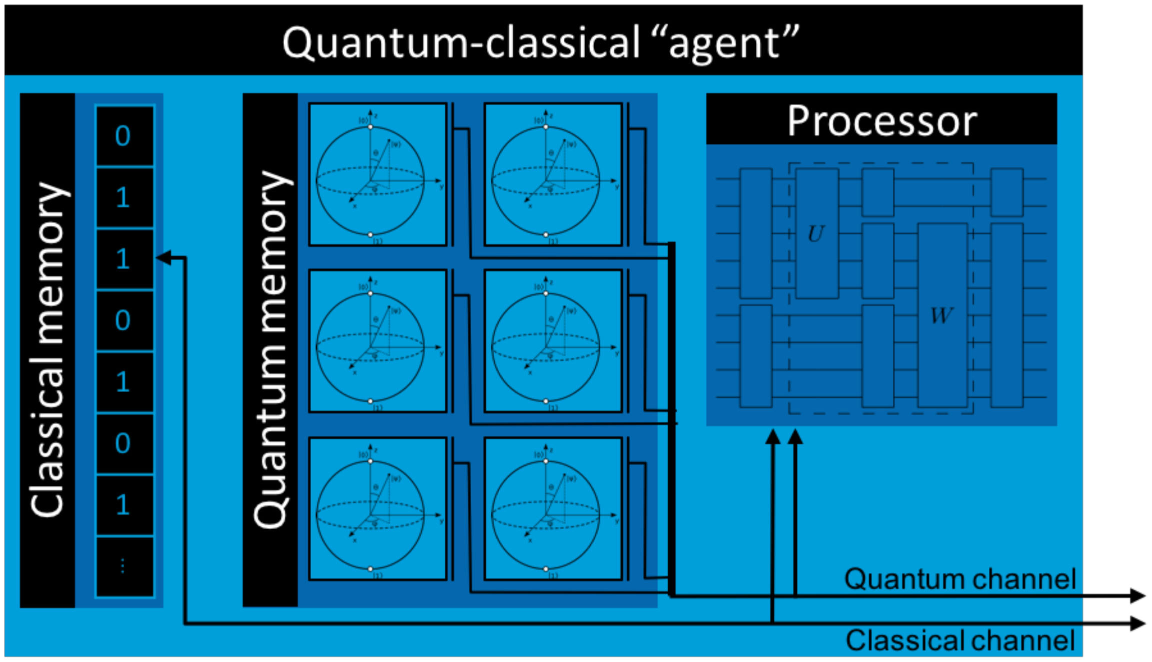
1.2.1 Information Representation and Processing

The fundamental unit of information in SQUANCH is the *QSystem*, which represents the quantum state of a multi-particle entangled system as a complex-valued density matrix. It contains references to the `Qubit` s that it comprises. A *QStream* represents a collection of disjoint (mutually unentangled) quantum systems, such as an ensemble of one million EPR pairs. `QSystem` s are lightweight, and can be instantiated by reference from a portion of an existing array (typically from a `QStream`), which vastly improves the cache locality and performance of operations on sequential quantum systems (such as encoding a stream of classical information on qubits using *superdense coding*).

SQUANCH users will interact most frequently with the lightweight wrapper *Qubit* class, which mirrors the methods of `QSystem` to more intuitively manipulate the states of quantum systems. `Qubits` have very little internal information, maintaining only a reference to their parent `QSystem` and a qubit index.

The *Gates* module provides a number of built-in quantum gates to manipulate qubits. Under the hood, it has a number of caching functions that remember previously-used operators to avoid repeating expensive tensor calculations, and it is easily extensible to define custom operators.

1.2.2 Agents and Channels



The top-level modules that provide the greatest abstraction are *Agents* and *Channels*, which implement the nodes and connections in a quantum network, repsectively. An `Agent` generalizes the notion of an actor (Alice, Bob, etc.) that can manipulate, send, receive, and store classical and quantum information. Agents have internal clocks, classical and quantum memories, classical and quantum incoming and outgoing channels that can connect to other agents, and a processor in the form of a `run()` function that implements runtime logic.

In simulations, agents run in parallel from separate processes, synchronizing clocks and passing information between each other through classical and quantum channels. Channels are effectively wrappers for multiprocessed queues that track transmission times and speed of light delays and simulate errors on transmitted qubits, which are passed by a serialized reference.

1.2.3 Memory Structure and Time Synchronization

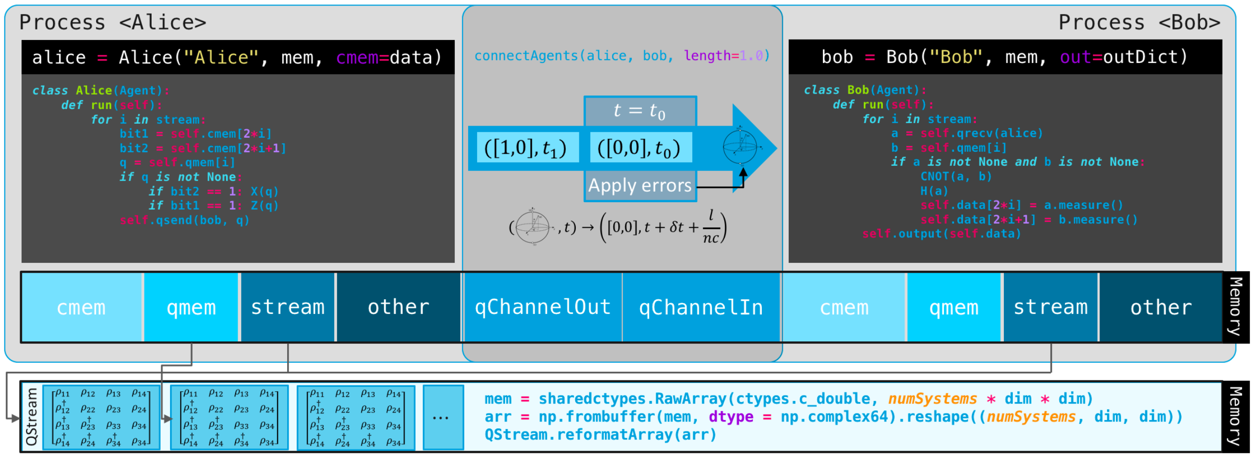
For optimal performance and for conceptual realism, agents (nodes in a network) run concurrently in separate processes that can only communicate by sending information through channels. Since separate processes normally have separate memory pools, this requires an interesting memory structure, since two agents running in separate processes must manipulate the same set of matrices in memory which represent the non-local combined quantum state shared between them. In other words, if Alice and Bob share an entangled pair, Alice's particle needs to be aware of the measurements performed on Bob's particle.

This is solved in SQUANCH by explicitly allocating an appropriately sized block of shared memory using the `sharedctypes` module, creating a 1D array of c-type doubles (which have the same size as the `numpy.complex64` values that are used to express density matrices in SQUANCH), which is casted and reshaped to a 3D complex-valued NumPy array. Agents can then instantiate separate `QStream` s that all point to the same physical memory location to represent their state. Since `Qubit` objects must be serialized to pass through Python's multiprocessing queues, channels serialized qubits to their (system, qubit) indices and reinstance the qubit for the receiving agent, insuring that they reference the correct location in memory.

SQUANCH includes rudimentary built-in timing features for agents to allow users to characterize the efficiency of protocols, taking specified values of photon pulse widths, signal travel speeds, length of channels, etc. into account.

Agents maintain separate clocks which are synchronized upon exchanging dependent information. For example, suppose Alice and Bob are sparated by 300m, and Alice transfers 10^5 qubits with a 10ps pulse width to Bob. Alice's clock at the beginning of the transmission is $1.5\mu s$, and Bob's clock is $2.0\mu s$. After the transmission, Alice's clock reads $1.5+10^5 \cdot 10^{-5} = 2.5\mu s$, and Bob's accounts for a speed of light delay to update to $2.0+10^5 \cdot 10^{-5} + \frac{300m}{c} = 4\mu s$.

A conceptual diagram of the memory structure and time synchronization protocol for two agents simulating information transfer via *superdense coding* is shown below.



```
alice = Alice("Alice", mem, cmem=data)
```

```
class Alice(Agent):
    def run(self):
        for i in stream:
            bit1 = self.cmem[2*i]
            bit2 = self.cmem[2*i+1]
            q = self.qmem[i]
            if q is not None:
                if bit2 == 1: X(q)
                if bit1 == 1: Z(q)
            self.qsend(bob, q)
```

```
connectAgents(alice, bob, length=1.0)
```

```
bob = Bob("Bob", mem, out=outDict)
```

```
class Bob(Agent):
    def run(self):
        for i in stream:
            a = self.qrecv(alice)
            b = self.qmem[i]
            if a is not None and b is not None:
                CNOT(a, b)
                H(a)
                self.data[2*i] = a.measure()
                self.data[2*i+1] = b.measure()
        self.output(self.data)
```

```
mem = sharedctypes.RawArray(ctypes.c_double, numSystems * dim * dim)
arr = np.frombuffer(mem, dtype = np.complex64).reshape((numSystems, dim, dim))
QStream.reformatArray(arr)
```

CHAPTER

TWO

GETTING STARTED

2.1 Requirements

SQUANCH is programmed in Python 3 and NumPy. You can obtain both of these, along with a host of other scientific computing tools, from the Anaconda package. Deprecated Python 2-compatible versions of SQUANCH are available in the Github repository commit history.

2.2 Installation

You can install SQUANCH directly using the Python package manager pip:

```
pip install squanch
```

If you don't have pip, you can get it using `easy_install pip`.

2.3 The basics of SQUANCH

Before we can run our first simulation, we'll need to introduce the notions of a `QSystem` and `Qubit`. A `QSystem` is the fundamental unit of quantum information in SQUANCH, and maintains the quantum state of a multi-particle, maximally-entangleable system. A `QSystem` also contains references to the `Qubit` s that comprise it, which allows you to work with them in a more intuitive manner. To manipulate qubits and quantum systems, we use quantum gates. Let's play around with these concepts for a moment.

```
from squanch import *

# Prepare a two-qubit system, which defaults to the |00> state
qsys = QSystem(2)
```

The state of a quantum system is tracked as a complex-valued density matrix in the computational basis:

```
qsys.state
```

```
array([[ 1.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
       [ 0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
       [ 0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
       [ 0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j]], dtype=complex64)
```

`QSystem` s also have a generator to yield their consistuent qubits. Note that this isn't the same as a list, as the qubits are instantiated only when they are asked for, not upon instantiation of the QSystem. (This saves on overhead, especially in cases when only one qubit in a system of many needs to be modified.)

```
qsys.qubits
```

```
<generator object <genexpr> at 0x107000460>
```

A possible point of confusion: since `qubits` is a generator object, you can only loop through the qubits of a given qsystem once (for a given agent – more on that below)! (If you need to access them multiple times, consider converting them to a list with `qubits = list(qsys.qubits)`.)

```
qsys2 = QSystem(2)
for i in range(3):
    print("Loop "+str(i))
    for qubit in qsys2.qubits:
        print(qubit)
```

```
Loop 0
<squanch.qubit.Qubit instance at 0x10d7d3828>
<squanch.qubit.Qubit instance at 0x10d7d3908>
Loop 1
Loop 2
```

You can access and work with the qubits of a system either by pattern matching them:

```
a, _ = qsys.qubits
print(a)
```

```
<squanch.qubit.Qubit instance at 0x10d540ea8>
```

or by requesting a specific qubit directly:

```
a2 = qsys.qubit(0)
print(a)
```

```
<squanch.qubit.Qubit instance at 0x10d533878>
```

Even though `a` and `a2` are separate objects in memory, they both represent the same qubit and will manipulate the same parent `QSystem`, which can be referenced using `a.qsystem`:

```
a.qsystem
<squanch.qubit.QSystem instance at 0x107cfc3b0>

a2.qsystem
<squanch.qubit.QSystem instance at 0x107cfc3b0>
```

For example, applying a Hadamard transformation to each of them yields the expected results:

```
H(a)
qsys.state
```

```
array([[ 0.5+0.j,  0.0+0.j,  0.5+0.j,  0.0+0.j],
       [ 0.0+0.j,  0.0+0.j,  0.0+0.j,  0.0+0.j],
       [ 0.5+0.j,  0.0+0.j,  0.5+0.j,  0.0+0.j],
       [ 0.0+0.j,  0.0+0.j,  0.0+0.j,  0.0+0.j]], dtype=complex64)
```

And applying the same (self-adjoint) transformation to `a2` gives the original $|00\rangle$ state (ignoring machine errors):

```
H(a2)
qsys.state
```

```
array([[  1.00000000e+00+0.j,   0.00000000e+00+0.j,   0.00000000e+00+0.j,   0.
↪00000000e+00+0.j],
       [  0.00000000e+00+0.j,   0.00000000e+00+0.j,   0.00000000e+00+0.j,   0.
↪00000000e+00+0.j],
       [ -2.23711427e-17+0.j,   0.00000000e+00+0.j,   0.00000000e+00+0.j,   0.
↪00000000e+00+0.j],
       [  0.00000000e+00+0.j,   0.00000000e+00+0.j,   0.00000000e+00+0.j,   0.
↪00000000e+00+0.j]], dtype=complex64)
```

2.4 Running your first simulation

Now that we've introduced the basics of working with quantum states in SQUANCH, let's start with a simple demonstration that can demonstrate some of the most basic capabilities of SQUANCH. We'll just prepare an ensemble of Bell pairs in the state $|q_1 q_2\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ and verify that they all collapse to the same states. For this example, all we'll need are the *qubit* and *gates* modules. We'll create a new two-particle quantum system in each iteration of the loop, and then apply H and CNOT operators to the system's qubits to make the Bell pair.

```
from squanch import *

results = [] # Where we'll put the measurement results

for _ in range(10):
    qsys = QSystem(2)
    a, b = qsys.qubits # enumerate the qubits of the system
    # Make a Bell pair
    H(a)
    CNOT(a, b)
    # Measure the pair and append to results
    results.append([a.measure(), b.measure()])

print(results)
```

Running the whole program, we obtain:

```
[[0, 0], [1, 1], [0, 0], [1, 1], [0, 0], [1, 1], [0, 0], [0, 0], [1, 1], [0, 0]]
```

2.5 Introduction to quantum streams

One of the more unique concepts to SQUANCH comapred to other quantum simulation frameworks is the notion of a "quantum stream", or *QStream*. This is the quantum analogue of a classical bitstream; a collection of disjoint (non-entangled) quantum systems. As before, let's play around with these.

```
from squanch import *

# Prepare a stream of 3 two-qubit systems
stream = QStream(2, 3)
```

The state of a `QStream` is just an array of density matrices, each element of which can be used to instantiate a `QSystem`:

```
stream.state
```

```
array([[[ 1.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
        [ 0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
        [ 0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
        [ 0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j]],

       [[ 1.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
        [ 0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
        [ 0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
        [ 0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j]],

       [[ 1.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
        [ 0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
        [ 0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
        [ 0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j]]], dtype=complex64)
```

You can pull specific systems from a stream an manipulate them. For example, let's apply H to the second qubit of the third system in the stream:

```
first_system = stream.system(2)
H(first_system.qubit(1))
```

```
array([[[ 1.0+0.j,  0.0+0.j,  0.0+0.j,  0.0+0.j],
        [ 0.0+0.j,  0.0+0.j,  0.0+0.j,  0.0+0.j],
        [ 0.0+0.j,  0.0+0.j,  0.0+0.j,  0.0+0.j],
        [ 0.0+0.j,  0.0+0.j,  0.0+0.j,  0.0+0.j]],

       [[ 1.0+0.j,  0.0+0.j,  0.0+0.j,  0.0+0.j],
        [ 0.0+0.j,  0.0+0.j,  0.0+0.j,  0.0+0.j],
        [ 0.0+0.j,  0.0+0.j,  0.0+0.j,  0.0+0.j],
        [ 0.0+0.j,  0.0+0.j,  0.0+0.j,  0.0+0.j]],

       [[ 0.5+0.j,  0.5+0.j,  0.0+0.j,  0.0+0.j],
        [ 0.5+0.j,  0.5+0.j,  0.0+0.j,  0.0+0.j],
        [ 0.0+0.j,  0.0+0.j,  0.0+0.j,  0.0+0.j],
        [ 0.0+0.j,  0.0+0.j,  0.0+0.j,  0.0+0.j]]], dtype=complex64)
```

You can also iterate over the systems in a stream:

```
for qsys in stream:
    a, b = qsys.qubits
    print([a.measure(), b.measure()])
```

```
[0, 0]
[0, 0]
[0, 1]
```

Using QStreams has a number of advantages: it reduces instantiation overhead, it allows *Agents* (which we'll talk about in a bit) to manipulate the same quantum states, and it can vastly increase performance by providing good cache locality. Typical sequential operations operating in a single thread will usually see a performance gain of about 2x, but for simulations involving a large number of Agents in separate processes working on qubits in varying positions in the stream, you may see much larger performance gains.

2.6 A simulation with QStreams

Here's a brief demonstration of how to use QStreams in your programs and an example of performance speedups.

```
from squanch import *
import time

num_systems = 100000

# Without streams: make a bunch of Bell pairs
start = time.time()
for _ in range(num_systems):
    a, b = QSystem(2).qubits
    H(a)
    CNOT(a, b)
print("Creating {} bell pairs without streams: {:.3f}s".format(num_systems, time.
↪time() - start))

# With a stream: make a bunch of Bell pairs
start = time.time()
stream = QStream(2, num_systems)
for qsys in stream:
    a, b = qsys.qubits
    H(a)
    CNOT(a, b)
print("Creating {} bell pairs with streams:    {:.3f}s".format(num_systems, time.
↪time() - start))
```

```
Creating 100000 bell pairs without streams: 5.564s
Creating 100000 bell pairs with streams:    2.355s
```

2.7 Using agents in your simulations

So far, we've touched on features that mostly have analogues in other quantum computing frameworks. However, SQUANCH is a quantum *networking* simulator, designed specifically for easily and concurrently simulating multiple agents which manipulate and transfer quantum inforamtion between each other.

An *Agent* generalizes the notion of a quantum-classical "actor". Agents are programmed by extending the base Agent class to contain the runtime logic in the `run()` function. In simulations, Agents run in separate processes, so it is necessary to explicitly pass in input and output structures, including the shared Hilbert space the Agents act on, and a multiprocessed return dictionary for outputting data from runtime. Both of these are included in the *Agents* module.

Here's a demonstration of a simple message tranmsision protocol using qubits as classical bits. There will be two agents, Alice and Bob; Alice will have a message encoded as a bitstream, which she will use to act on her qubits that she will send to Bob, who will reconstruct the original message. Let's start with the preliminary imports and string to bitstream conversion functions:

```
from squanch import *

def string_to_bits(msg):
    # Return a string of 0's and 1's from a message
    bits = ""
    for char in msg: bits += "{:08b}".format(ord(char))
    return bits
```

(continues on next page)

(continued from previous page)

```python

def bits_to_string(bits):
    # Return a message from a binary string
    msg = ""
    for i in range(0, len(bits), 8):
        digits = bits[i:i + 8]
        msg += chr(int(digits, 2))
    return msg

msg = "Hello, Bob!"
bits = string_to_bits(msg)
```

To program the agents themselves, we extend the Agent base class and overwrite the `run()` function:

```python
class Alice(Agent):
    def run(self):
        for qsys, bit in zip(self.qstream, self.data):
            q, = qsys.qubits
            if bit == "1": X(q)
            self.qsend(bob, q)


class Bob(Agent):
    def run(self):
        bits = ""
        for _ in self.qstream:
            q = self.qrecv(alice)
            bits += str(q.measure())
        self.output(bits)
```

To instantiate and run the agents, we need to provide them with a *QStream* to operate on, and if we want them to return values, we'll need to give them a shared output dictionary with *Agent.shared_output*. Explicitly passing output dictionaries to agents is necessary because each agent runs in its own separate process, which (generally) have separate memory pools. (See *Agent* API for more details.) We then connect the agents with a quantum channel:

```python
qstream = QStream(1, len(msgBits))
out = Agent.shared_output()

alice = Alice(qstream, data = bits)
bob = Bob(qstream, out = out)

alice.qconnect(bob)
```

Running the agents has the same syntax as running a *Process* in Python. *alice.start()* starts Alice's runtime logic, and *alice.join()* waits for all other agents to finish executing:

```python
alice.start()
bob.start()

alice.join()
bob.join()

received_msg = bits_to_string(out["Bob"])
print("Alice sent: '{}'. Bob received: '{}'.".format(msg, received_msg))
```

```
Alice sent: 'Hello, Bob!'. Bob received: 'Hello, Bob!'.
```

Alternately, SQUANCH also includes a *Simulation* module which can track the progress of each agent as they execute their code and display a progress bar in a terminal or Jupyter notebook:

```
Simulation(alice, bob).run()
received_msg = bits_to_string(out["Bob"])
print("Alice sent: '{}'. Bob received: '{}'.".format(msg, received_msg))
```

```
Alice sent: 'Hello, Bob!'. Bob received: 'Hello, Bob!'.
```

2.8 See also

This tutorial page only touches on some of the basic uses of SQUANCH. For demonstrations of more complex scenarios, see the *demonstrations section*, and for an overview of SQUANCH's core concepts and organization, see the *overview section*.

CHAPTER

THREE

DEMONSTRATIONS

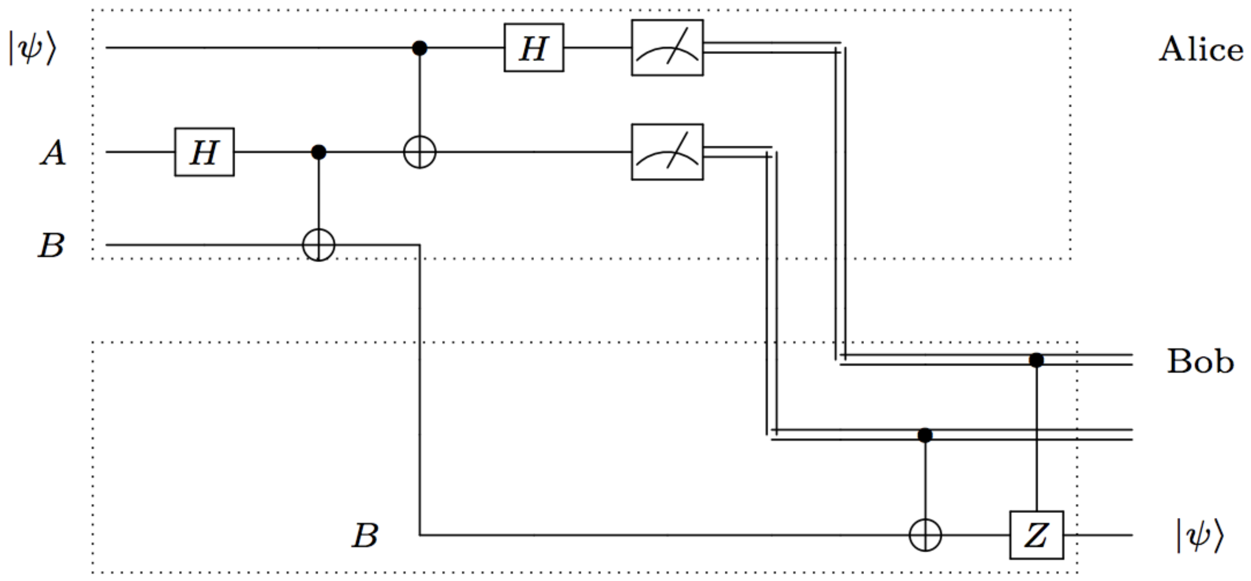
3.1 Quantum Teleportation

Quantum teleportation allows two parties that share an entangled pair to transfer an arbitrary quantum state using only classical communication. This process has tremendous applicability to quantum networks, transferring fragile quantum states between distant nodes. Conceptually, quantum teleportation is the inverse of *superdense coding*.

In general, all quantum teleportation experiments have the same underlying structure. Two distant parties, Alice and Bob, are connected via a classical information channel and share a maximally entangled state. Alice has an unknown state $|\psi\rangle$ which she wishes to send to Bob. She performs a joint projective measurement of her state and her half of the entangled state and communicates the outcomes to Bob, who operates on his half of the entangled state accordingly to reconstruct $|\psi\rangle$.

The source code for this demo is included in the *demos* directory of the SQUANCH repository.

3.1.1 Protocol



In this demo, we'll implement a simple two-party quantum teleportation protocol using the above circuit diagram.

1. Alice generates an entangled two-particle state $|AB\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, keeping half of the state and sending the other half to Bob.

2. Alice entangles her qubit $|\psi\rangle$ with her ancilla A by applying controlled-not and Hadamard operators.

3. Alice jointly measures $|\psi\rangle$ and A and communicates the outcomes to Bob through a classical channel. Bob's qubit is now in one of four possible Bell states, one of which is $|\psi\rangle$, and he will use Alice's two bits to recover $|\psi\rangle$
4. Bob applies a Pauli-X operator to his qubit if Alice's ancilla collapsed to $|A\rangle \mapsto |1\rangle$, and he applies a Pauli-Z operator to his qubit if her qubit collapsed to $|\psi\rangle \mapsto |1\rangle$. He has thus transformed $|B\rangle \mapsto |\psi\rangle$.

3.1.2 Implementation

Quantum teleportation is a simple protocol to implement in any quantum computing simulation framework, but SQUANCH's *Agent* and *Channel* modules provide an intuitive way to work with sending and receiving qubits, and the *QStream* module allows you to create performant simulations of teleporting a large number of states in succession.

First, let's import what we'll need.

```
import numpy as np
import matplotlib.pyplot as plt
from squanch import *
```

Now, we'll want to define the behavior of Alice and Bob. We'll extend the *Agent* class to create two child classes, and then we can change the *run()* method for each of them. For Alice, we'll want to include logic for creating an EPR pair and sending it to Bob, as well as the subsequent entanglement and measurement logic.

```
class Alice(Agent):
        '''Alice sends qubits to Bob using a shared Bell pair'''

        def distribute_bell_pair(self, a, b):
                # Create a Bell pair and send one particle to Bob
                H(a)
                CNOT(a, b)
                self.qsend(bob, b)

        def teleport(self, q, a):
                # Perform the teleportation
                CNOT(q, a)
                H(q)
                # Tell Bob whether to apply Pauli-X and -Z over classical channel
                bob_should_apply_x = a.measure() # if Bob should apply X
                bob_should_apply_z = q.measure() # if Bob should apply Z
                self.csend(bob, [bob_should_apply_x, bob_should_apply_z])

        def run(self):
                for qsystem in self.qstream:
                        q, a, b = qsystem.qubits # q is state to teleport, a and b␣
↪are Bell pair
                        self.distribute_bell_pair(a, b)
                        self.teleport(q, a)
```

Note that you can add arbitrary methods, such as *distribute_bellPair()* and *teleport()*, to agent child classes; just be careful not to overwrite any existing class methods other than *run()*.

For Bob, we'll want to include the logic to receive the particle from Alice and act on it according to Alice's measurement results.

```
class Bob(Agent):
        '''Bob receives qubits from Alice and measures the results'''

```

(continues on next page)

(continued from previous page)

```
        def run(self):
                measurement_results = []
                for _ in self.qstream:
                        # Bob receives a qubit from Alice
                        b = self.qrecv(alice)
                        # Bob receives classical instructions from alice
                        should_apply_x, should_apply_z = self.crecv(alice)
                        if should_apply_x: X(b)
                        if should_apply_z: Z(b)
                        # Measure the output state
                        measurement_results.append(b.measure())
                # Put results in output object
                self.output(measurement_results)
```

Now we want to prepare a set of states for Alice to teleport to Bob. Since each trial requires a set of three qubits, we'll allocate space for a 3×10 *QStream*. We'll encode the message as spin eigenstates in the *QStream*:

```
    # Prepare the initial states
qstream = QStream(3,10) # 3 qubits per trial, 10 trials
states_to_teleport = [1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
for state, qsystem in zip(states_to_teleport, qstream):
    q = qsystem.qubit(0)
    if state == 1: X(q) # flip the qubits corresponding to 1 states
```

Now let's make the agent instances. We create a shared output dictionary to allow agents to communicate between processes. Explicitly allocating and passing an output object to agents is necessary because each agent spawns and runs in a separate process, which (generally) have separate memory pools. (See *Agent* API for more details.) For agents to communicate with each other, they must be connected via quantum or classical channels. The *Agent.qconnect* and *Agent.cconnect* methods add a bidirectional quantum or classical channel, repsectively, to two agent instances and take a channel model and kwargs as optional arguments. In this example, we won't worry about a channel model and will just use the default QChannel and CChannel options. Let's create instances for Alice and Bob and connect them appropriately

```
# Make and connect the agents
out = Agent.shared_output()
alice = Alice(qstream, out)
bob = Bob(qstream, out)
alice.qconnect(bob) # add a quantum channel
alice.cconnect(bob) # add a classical channel
```

Finally, we call *agent.start()* for each agent to signal the process to start running, and *agent.join()* to wait for all agents to finish before proceeding in the program.

```
# Run everything
alice.start()
bob.start()
alice.join()
bob.join()

print("Teleported states {}".format(states_to_teleport))
print("Received states   {}".format(out["Bob"]))
```

Running what we have so far produces the following output:

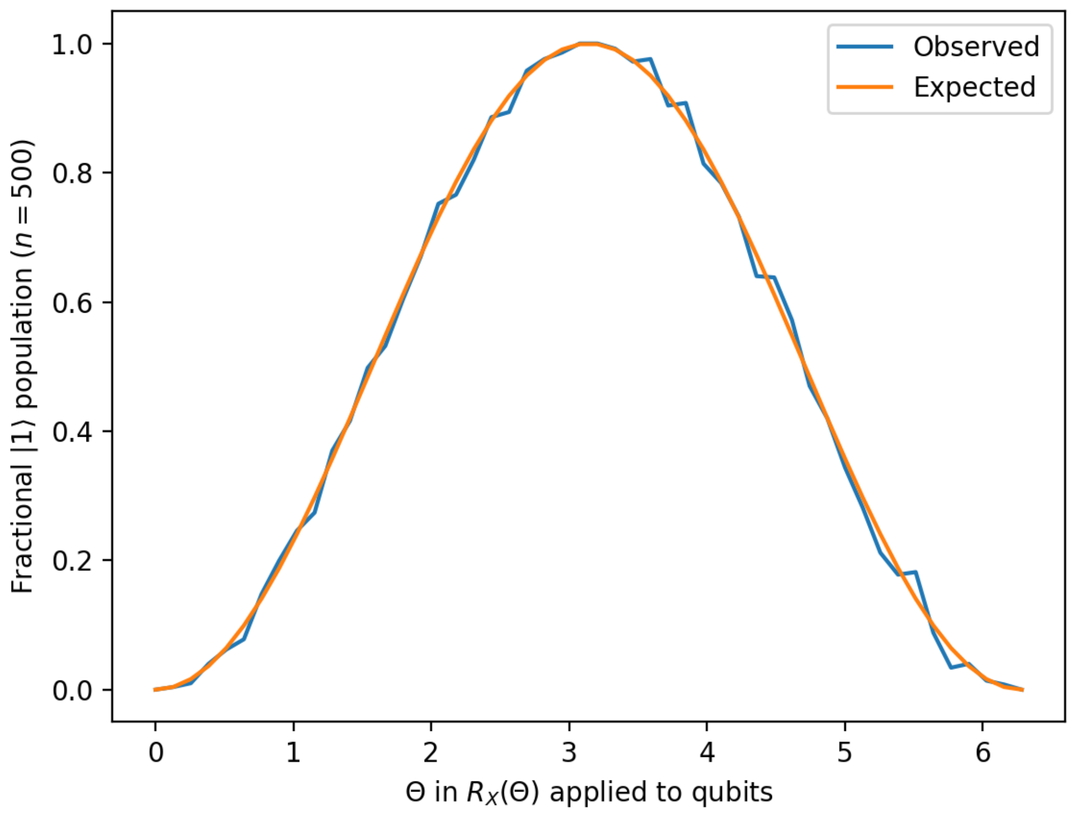
```
Teleported states [1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
Received states   [1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
```

So at least for the simple cases, our implementation seems to be working! Let's do a little more complex test case now.

We'll now try teleporting an ensemble of identical states $R_X(\theta)|0\rangle$ for several values of θ. We'll then measure each teleported state and see how it compares with the expected outcome.

```python
angles = np.linspace(0, 2 * np.pi, 50)  # RX angles to apply
num_trials = 250  # number of trials for each angle

# Prepare the initial states in the stream
qstream = QStream(3, len(angles) * num_trials)
for angle in angles:
    for _ in range(num_trials):
        q, _, _ = qstream.next().qubits
        RX(q, angle)

# Make the agents and connect with quantum and classical channels
out = Agent.shared_output()
alice = Alice(qstream, out = out)
bob = Bob(qstream, out = out)
alice.qconnect(bob)
alice.cconnect(bob)

# Run the simulation
Simulation(alice, bob).run()

# Plot the results
results = np.array(out["Bob"]).reshape((len(angles), num_trials))
observed = np.mean(results, axis = 1)
expected = np.sin(angles / 2) ** 2
plt.plot(angles, observed, label = 'Observed')
plt.plot(angles, expected, label = 'Expected')
plt.legend()
plt.xlabel("$\Theta$ in $R_X(\Theta)$ applied to qubits")
plt.ylabel("Fractional $\left | 1 \\right >$ population")
plt.show()
```

This gives us the following pretty plot.



3.1.3 Source code

The full source code for this demonstration is available in the demos directory of the SQUANCH repository.

3.2 Superdense Coding

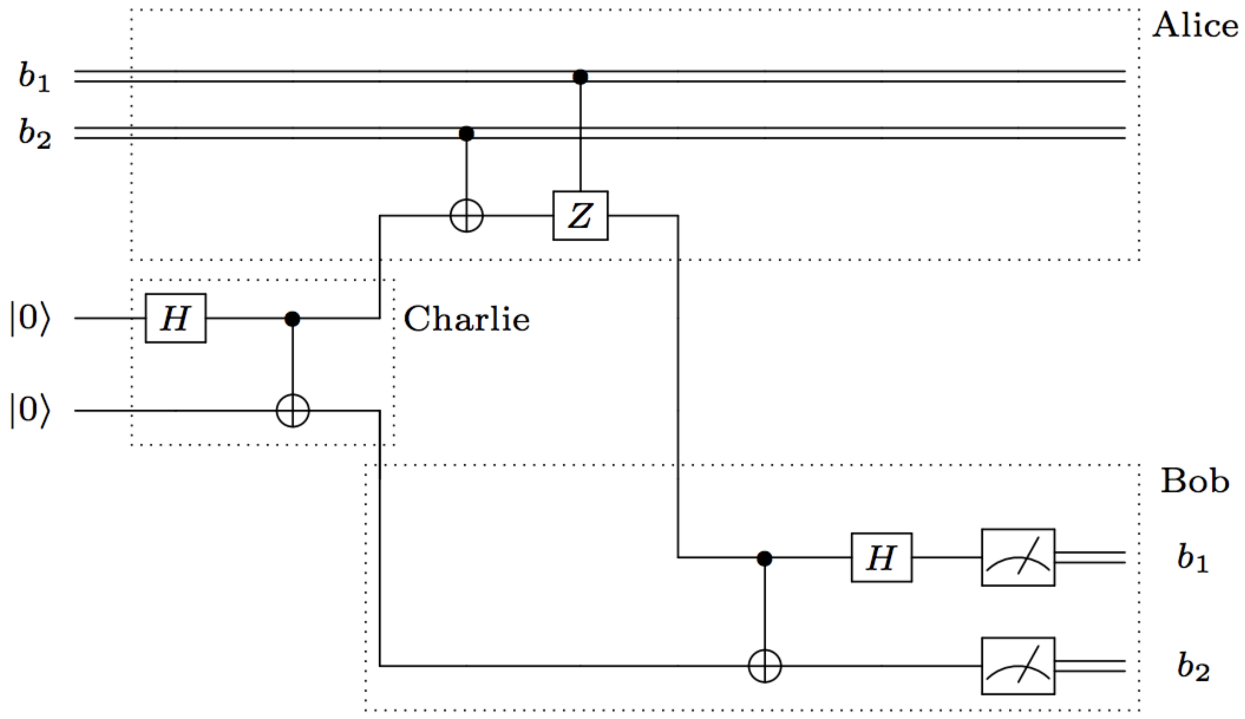
Superdense coding is a process whereby two parties connected via a quantum channel and sharing an entangled pair can send two classical bits of information using only a single qubit. Conceptually, superdense coding is the inverse of *quantum teleportation*.

In this demonstration, we'll implement the three-party superdense coding protocol depicted in the circuit diagram shown below. Charlie distributes entangled particles to Alice and Bob. Alice encodes her information in her particles and sends them to Bob, who decodes the information by matching Alice's qubits with his half of the shared state received from Charlie.

For this demonstration, Alice will send data to Bob in the form of a serialized bitstream representing an image. We'll use the built-in timing functionality to track the simulated time for each agent to complete their part of the protocol. Since superdense coding could be used as a networking protocol in the foreseeable future, even very rudimentary simulated timing data could be useful to quantify the performance of the algorithm, especially if data validation and error correction through multiple transmission attempts is simulated. We assume a photon pulse interval of 1ns and a spatial separation between Alice and Bob of 1km, with Charlie at the midpoint. All agents are connected with the

`FiberOpticQChannel` model, which simulates 0.16 dB/km attenuation errors by randomly changing transmitted qubits to `None`. Any dropped qubits lost to attenuation will have their bits replaced with 0.

3.2.1 Protocol



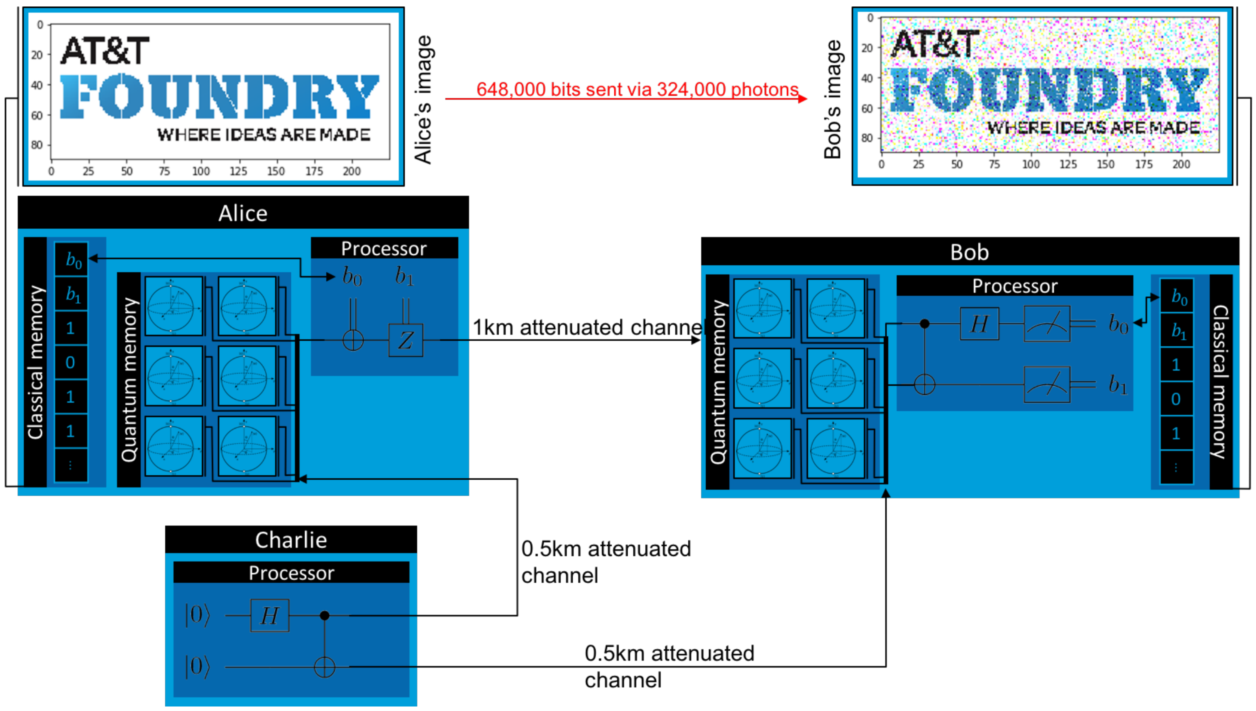
We'll be using the above circuit diagram to describe a three-party quantum superdense coding protocol. There are three agents: Charlie distributes entangled particles to Alice and Bob, Alice encodes her information in her particles and sends them to Bob, who decodes the information by matching Alice's qubits with his own qubits received from Charlie.

1. Charlie generates EPR pairs in the state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. He sends one particle to Alice and the other to Bob.
2. Alice encodes her two bits of classical inforation in the relative sign and phase of her qubit by acting with the Pauli-X and -Z gates. Formally, if she has two bits, b_1 and b_2, she applies X if $b_2 = 1$ and then applies Z if $b_1 = 1$. She then sends the modified qubit to Bob.
3. Bob disentangles the X and Z components of the qubit by applying CNOT and H to Alice's qubit and Charlie's qubit. He then measures each of Alice's and Charlie's qubits to obtain b_1 and b_2, respectively.

We'll also use SQUANCH's built-in timing functionality to track the simulated time for each agent to complete their part of the protocol assuming a photon pulse interval of 1ns.

3.2.2 Implementation

Because superdense coding transmits classical information through quantum channels, it makes for a good protocol to visually demonstrate both the tranmission of the information and some of SQUANCH's simulated errors. (This also makes it a good demonstration for implementing classical and quantum error corrections, although we won't do that in this demo.) The protocol we'll be implementing looks like this at a conceptual level:



First, let's import the modules we'll need.

```
import numpy as np
import time
import matplotlib.image as image
import matplotlib.pyplot as plt
from squanch import *
```

Now, as usual, we'll want to define child *Agent* classes that implement the behavior we want. For Charlie, we'll want to include the behavior to make an EPR pair and distribute it to Alice and Bob.

```
class Charlie(Agent):
        '''Charlie distributes Bell pairs between Alice and Bob.'''
        def run(self):
                for qsys in self.qstream:
                        a, b = qsys.qubits
                        H(a)
                        CNOT(a, b)
                        self.qsend(alice, a)
                        self.qsend(bob, b)
    self.output({"t": self.time})
```

For Alice, we'll want to include the transmission behavior. We'll pass in the data that she wants to transmit as a 1D array in an input argument when we instantiate her, and it will be stored in *self.data*.

```
class Alice(Agent):
        '''Alice sends information to Bob via superdense coding'''
        def run(self):
                for _ in self.qstream:
                        bit1 = self.data.pop(0)
                        bit2 = self.data.pop(0)
                        q = self.qrecv(charlie)
                        if q is not None: # qubit could be lost due to attenuation␣
↪errors
```

(continues on next page)

(continued from previous page)

```
                                if bit2 == 1: X(q)
                                if bit1 == 1: Z(q)
                        self.qsend(bob, q)
    self.output({"t": self.time})
```

Finally, for Bob, we'll want to include the disentangling and measurement behavior, and we'll want to output his measured data using *self.output*, which passes it to the parent process through the *shared_output* that is provided to agents on instantiation.

```
class Bob(Agent):
        '''Bob receives Alice's transmissions and reconstructs her information'''
        def run(self):
                bits = []
                for _ in self.qstream:
                        a = self.qrecv(alice)
                        c = self.qrecv(charlie)
                        if a is not None and c is not None:
                                CNOT(a, c)
                                H(a)
                                bits.extend([a.measure(), c.measure()])
                        else:
                                bits.extend([0,0])
    self.output({"t": self.time, "bits": bits})
```

Now, we want to instantiate Alice, Bob, and Charlie, and run the protocol. To do this, we'll need to pass in the data that Alice will send to Bob (which will be an image serialized to a 1D array of bits) using the *data* keyword argument, and we'll need to provide the agents with the *QStream* they will work on. If we want to get anything from the agents after their processes have finished, we'll also need to pass as an output structure to push their data to. (This is necessary because all agents run in separate processes, so explicitly shared output dictionaries must be passed to them.)

```
# Load an image and serialize it to a bitstream
img = image.imread("../docs/source/img/foundryLogo.bmp")
bitstream = list(np.unpackbits(img))

# Initialize the qstream
qstream = QStream(2, int(len(bitstream) / 2))

# Make agent instances
out = Agent.shared_output()
alice = Alice(qstream, out, data = bitstream)
bob = Bob(qstream, out)
charlie = Charlie(qstream, out)

# Set photon transmission rate
alice.pulse_length = 1e-9
bob.pulse_length = 1e-9
charlie.pulse_length = 1e-9
```

For agents to communicate with each other, they must be connected via quantum or classical channels. The *Agent.qconnect* and *Agent.cconnect* methods add a bidirectional quantum or classical channel, repsectively, to two agent instances and take as arguments a channel model and associated keyword arguments. SQUANCH includes several built-in rudimentary channel models, including a fiber optic cable model which simulates attenuation errors. (For more on channel and error models, see the *quantum error correction demo*.) Let's say that Alice and Bob are separated by a 1km fiber optic cable, and Charlie is at the midpoint, 0.5km away from each.

```
# Connect the agents with simulated fiber optic lines; see squanch.channels module
alice.qconnect(bob, FiberOpticQChannel, length=1.0)
charlie.qconnect(alice, FiberOpticQChannel, length=0.5)
charlie.qconnect(bob, FiberOpticQChannel, length=0.5)
```

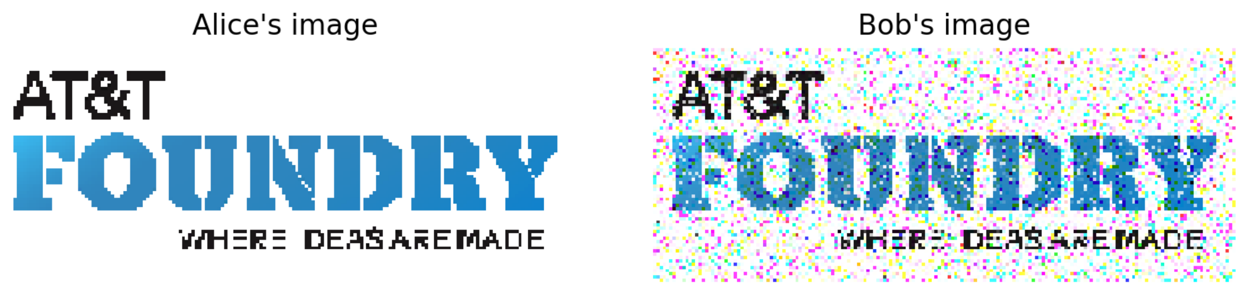
Once we've connected the agents, we just need to run all of the agent processes with *start()* and wait for them to finish with *join()*.

```
    # Run the agents
start = time.time()
Simulation(alice, bob, charlie).run()

print("Transmitted {} bits in {:.3f}s.".format(len(out["Bob"]), time.time() - start))
t_alice, t_bob, t_charlie = out["Alice"]["t"], out["Bob"]["t"], out["Charlie"]["t"]
print("Simulated time: Alice: {:.2e}s, Bob: {:.2e}s, Charlie: {:.2e}s"
      .format(t_alice, t_bob, t_charlie))
```

```
Transmitted 2 bits in 74.323s.
Simulated time: Alice: 4.16e-04s, Bob: 4.20e-04s, Charlie: 4.15e-04s
```

Finally, let's retrieve Bob's data and repackage it into an image array, then compare the results.

```
received = np.reshape(np.packbits(out["Bob"]["bits"]), img.shape)
f, ax = plt.subplots(1, 2, figsize = (8, 4))
ax[0].imshow(img)
ax[0].axis('off')
ax[0].title.set_text("Alice's image")
ax[1].imshow(received)
ax[1].axis('off')
ax[1].title.set_text("Bob's image")
plt.tight_layout()
plt.show()
```

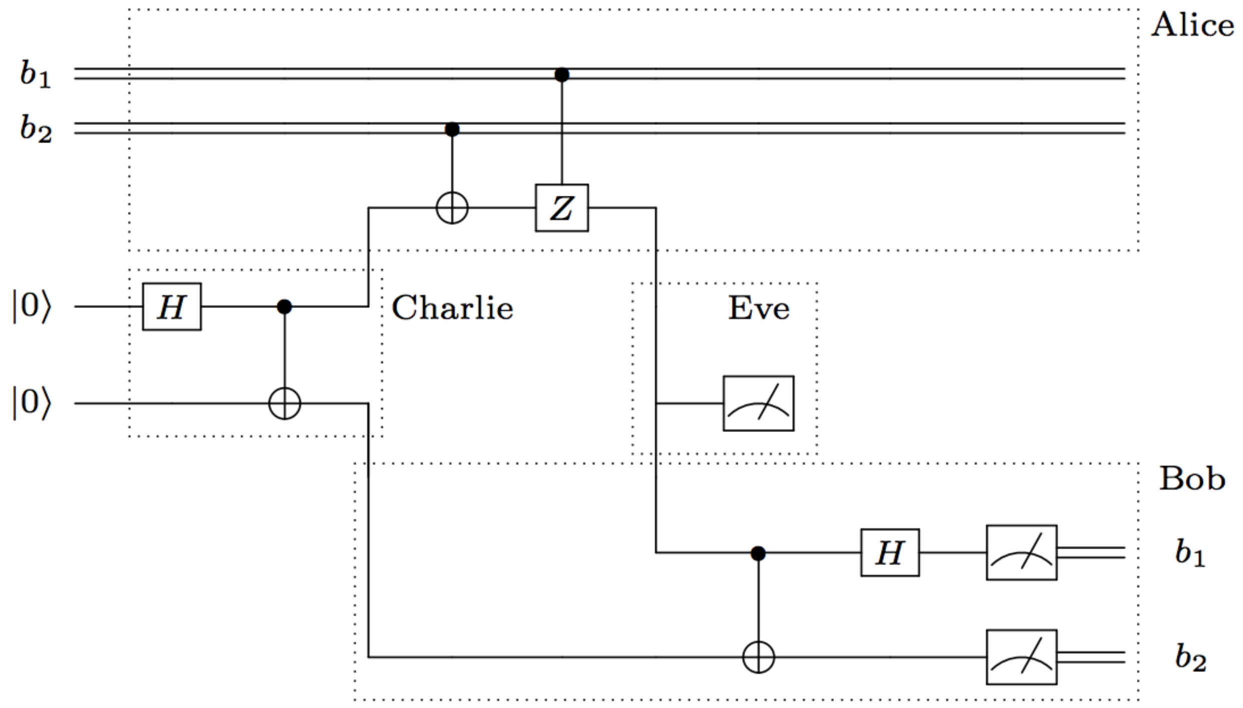


3.2.3 Source code

The full source code for this demonstration is available in the demos directory of the SQUANCH repository.

3.3 Man-In-The-Middle Attack

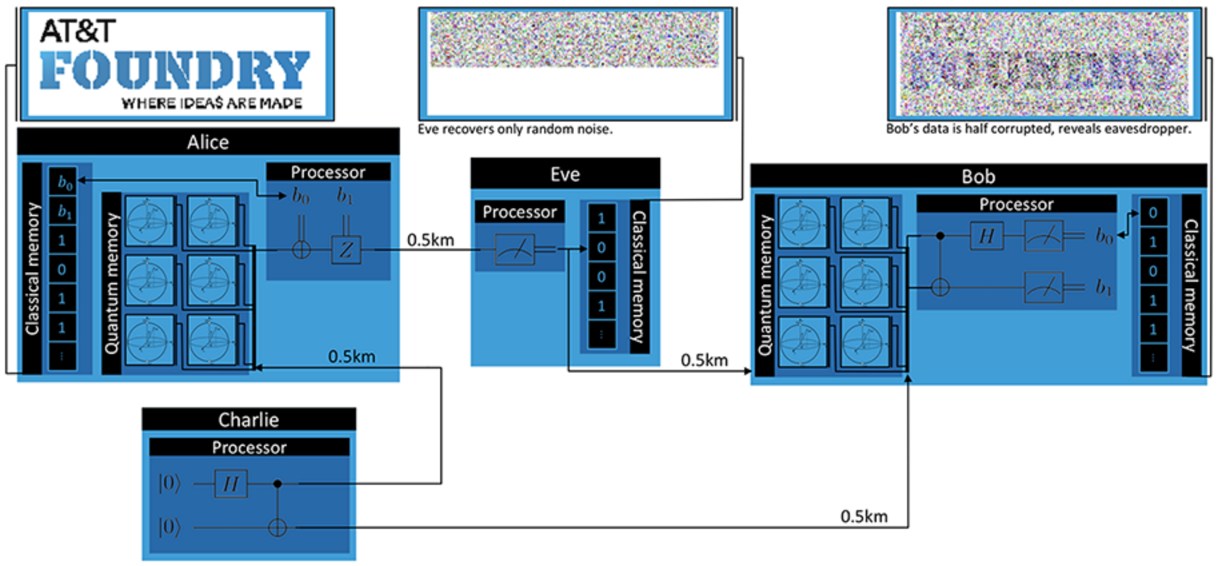
In this demo, we show how quantum networks can be resistant to interception ("man-in-the-middle") attacks by using a modified version of the *superdense coding*. As in the superdense coding demo, Charlie will distribute Bell pairs to Alice and Bob, and Alice will attempt to send a classical message to Bob. However, a fourth party, Eve, will try to naively intercept the message Alice sends to Bob. Eve will measure each qubit from Alice, record the result, and re-transmit the qubit to Bob. This scenario is illustrated in the circuit diagram shown below.



3.3.1 Protocol

As in the superdense coding demo, Charlie will distribute Bell pairs to Alice and Bob, and Alice will attempt to send a classical message to Bob. However, a fourth party, Eve, will try to naiively intercept the message Alice sends to Bob. Eve will measure each qubit from Alice, record the result, and send the qubit to Bob. This scenario is illustrated below.

1. Charlie generates EPR pairs in the state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. He sends one particle to Alice and the other to Bob.
2. Alice encodes her two bits of classical inforation in the relative sign and phase of her qubit by acting with the Pauli-X and -Z gates. Formally, if she has two bits, $b_1 and b_2$, she applies X if $b_2 = 1$ and then applies Z if $b_1 = 1$. She then sends the modified qubit to Bob, but it is intercepted by Eve first.
3. Eve wants to know Alice's message, so she naiively measures each qubit she intercepts from Alice and records the result. She then sends the qubits to Bob, hoping he won't notice.
4. Bob receives the qubit from Eve (who he thinks is Alice). He disentangles the X and Z components of the qubit by applying CNOT and H to Alice's qubit and Charlie's qubit. He then measures each of Alice's and Charlie's qubits to obtain b_1 and b_2, respectively.



3.3.2 Implementation

First, let's import the modules we'll need.

```
from squanch import *
import numpy as np
import matplotlib.image as image
import matplotlib.pyplot as plt
```

As before, we'll define the behavior of Alice, Bob, and Charlie. The only difference in their logic will be that Alice sends her qubit to Eve instead of Bob, and Bob receives his qubits from Eve instead of Alice.

```
class Alice(Agent):
        '''Alice sends information to Bob via superdense coding'''
        def run(self):
                for _ in self.qstream:
                        bit1 = self.data.pop(0)
                        bit2 = self.data.pop(0)
                        q = self.qrecv(charlie)
                        if q is not None:
                                if bit2 == 1: X(q)
                                if bit1 == 1: Z(q)
                        self.qsend(eve, q) # Alice unknowingly sends the qubit to Eve

class Bob(Agent):
        '''Bob receives Alice's transmissions and reconstructs her information'''
        def run(self):
                bits = []
                for _ in self.qstream:
                        a = self.qrecv(eve) # Bob receives his qubit from Eve
                        c = self.qrecv(charlie)
                        if a is not None and c is not None:
                                CNOT(a, c)
                                H(a)
                                bits.extend([a.measure(), c.measure()])
                        else:
```

(continues on next page)

(continued from previous page)

```
                                bits.extend([0,0])
                self.output(bits)

class Charlie(Agent):
        '''Charlie distributes Bell pairs between Alice and Bob.'''
        def run(self):
                for qsys in self.qstream:
                        a, b = qsys.qubits
                        H(a)
                        CNOT(a, b)
                        self.qsend(alice, a)
                        self.qsend(bob, b)
```

Now we'll add behavior for Eve to record and re-transmit Alice's qubits:

```
class Eve(Agent):
        '''Eve naively tries to intercept Alice's message'''
        def run(self):
                bits = []
                for _ in self.qstream:
                        a = self.qrecv(alice)
                        if a is not None:
                                bits.append(a.measure())
                        else:
                                bits.append(0)
                        self.qsend(bob, a)
                self.output(bits)
```

Next, we'll load an image and convert it to black and white and flatten it into a bitstream using some helper functions (see the corresponding Jupyter notebook in *demos* for details). We'll allocate an appropriately large *QStream* and make a shared output dictionary to allow the agents to return data.

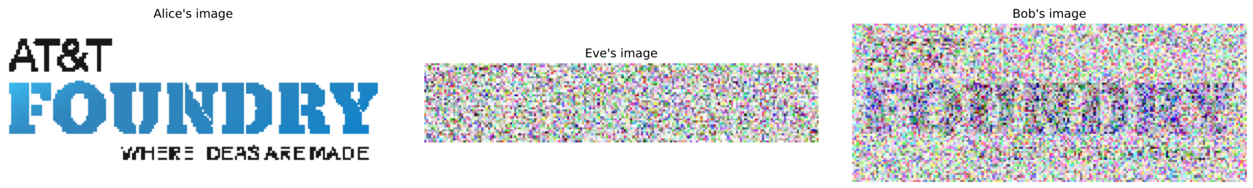
```
    # Load an image and serialize it to a bitstream
img = image_to_black_and_white("../docs/source/img/squanchLogo.jpg")
bitstream = list(img.flatten())

# Make the QStream for the agents to operate on
qstream = QStream(2, int(len(bitstream) / 2))

# Make agent instances
out = Agent.shared_output()

alice = Alice(qstream, out, data = bitstream)
bob = Bob(qstream, out)
charlie = Charlie(qstream, out)
eve = Eve(qstream, out)
```

Like in the superdense coding demonstration, we'll connect the agents with fiber optic lines to simulate attenuation errors.

```
# Connect the agents over simulated fiber optic lines
alice.qconnect(bob, FiberOpticQChannel, length=1.0)
alice.qconnect(eve, FiberOpticQChannel, length=0.5)
alice.qconnect(charlie, FiberOpticQChannel, length=0.5)
bob.qconnect(charlie, FiberOpticQChannel, length=0.5)
bob.qconnect(eve, FiberOpticQChannel, length=0.5)
```

Finally, we just need to make a simulation, run it, and plot what each agent receives.

```
Simulation(alice, eve, bob, charlie).run()
plot_alice_bob_eve_images(out["Eve"], out["Bob"])
```

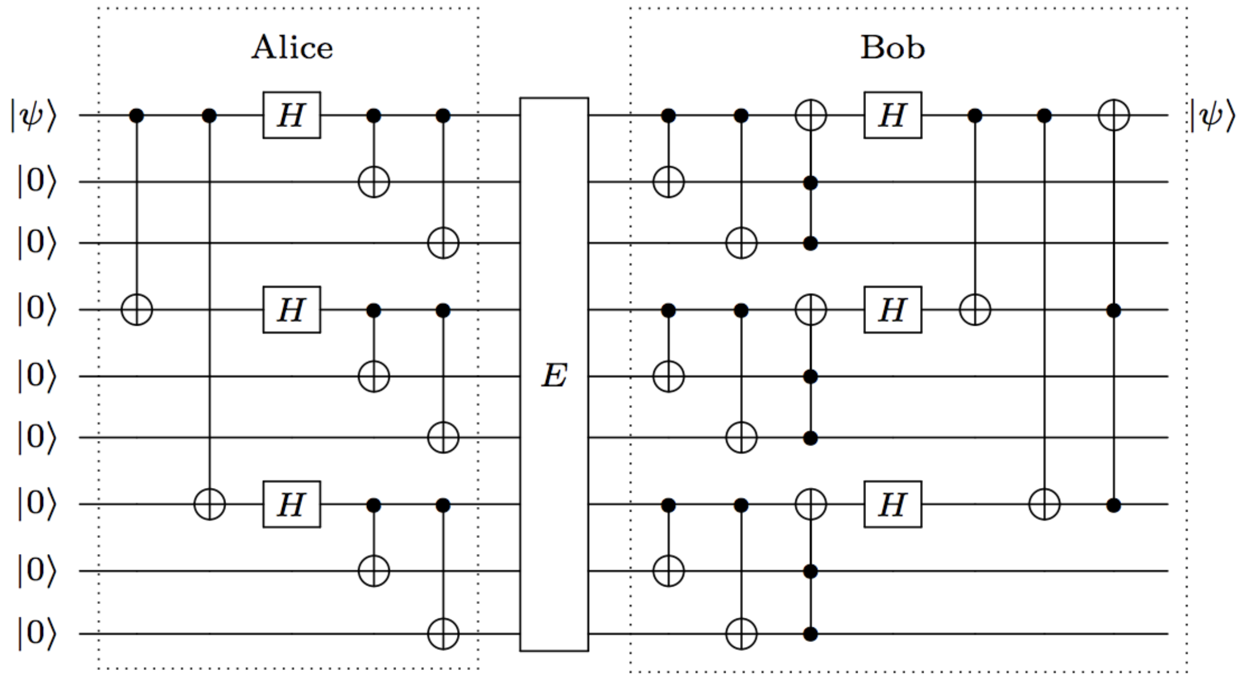


3.3.3 Source code

The full source code for this demonstration is available in the demos directory of the SQUANCH repository.

3.4 Quantum Error Correction

When qubits are transmitted over quantum channels, they are subject to a complex set of errors which can cause them to decohere, depolarize, or simply vanish completely. For quantum information transfer to be feasible, the information must be encoded in a error-resistant format using any of a variety of quantum error correction models. In this demonstration, we show how to use SQUANCH's channel and error modules to simulate quantum errors in a transmitted message, which we correct for using the Shor Code. This error correction model encodes a single logical qubit into the product of 9 physical qubits and is capable of correcting for arbitrary single-qubit errors. A circuit diagram for this protocol is shown below, where E represents a quantum channel which can arbitrarily corrupt a single qubit.



3.4.1 Protocol

In this demo, we have two pairs of agents: Alice and Bob will communicate a message which is error-protected using the Shor code, and DumbAlice an DumbBob will transmit the message without error correction. Formally, for each state $|\psi\rangle$ to be transmitted through the channel, the following procedure is simulated:

1. Alice has some state $|\psi\rangle = \alpha_0|0\rangle + \alpha_1|1\rangle$, which she wants to send to Bob through a noisy quantum channel. She encodes her state as $|\psi\rangle \to \alpha_0 \frac{1}{2\sqrt{2}}(|000\rangle + |111\rangle) \otimes (|000\rangle + |111\rangle) \otimes (|000\rangle + |111\rangle) + \alpha_1 \frac{1}{2\sqrt{2}}(|000\rangle - |111\rangle) \otimes (|000\rangle - |111\rangle) \otimes (|000\rangle - |111\rangle)$ using the circuit diagram above.
2. DumbAlice wants to do the same, but doesn't encode her state.
3. Alice and DumbAlice send their qubits through the quantum channel to Bob and DumbBob, respectively. The channel may apply an arbitrary unitary operation to a single physical qubit in each group of 9.
4. Bob and DumbBob receive their qubits. Bob decodes his using the Shor decoding circuit. DumbBob is dumb, and thus does nothing. For the purposes of this demonstration, the qubits will be measured and the results assembled to form a message.

Transmitting an image is unsuitable for this scenario due to the larger size of the Hilbert space involved compared to the previous two demonstrations. (Each `QSystem.state` for N=9 uses 2097264 bytes, compared to 240 bytes for N=2.) Instead, Alice and DumbAlice will transmit the bitwise representation of a short message encoded as σ_z-eigenstates, and Bob and DumbBob will attempt to re-assemble the message.

3.4.2 Implementation

```
from squanch import *
from scipy.stats import unitary_group
import copy
import numpy as np
import matplotlib.image as image
import matplotlib.pyplot as plt
```

First, let's define what each of the agents do. Let's start with Alice and DumbAlice.

```
class Alice(Agent):
        '''Alice sends an arbitrary Shor-encoded state to Bob'''
        def shor_encode(self, qsys):
                # psi is state to send, q1...q8 are ancillas from top to bottom in␣
↪diagram
                psi, q1, q2, q3, q4, q5, q6, q7, q8 = qsys.qubits
                # Gates are enumerated left to right, top to bottom from figure
                CNOT(psi, q3)
                CNOT(psi, q6)
                H(psi)
                H(q3)
                H(q6)
                CNOT(psi, q1)
                CNOT(psi, q2)
                CNOT(q3, q4)
                CNOT(q3, q5)
                CNOT(q6, q7)
                CNOT(q6, q8)
                return psi, q1, q2, q3, q4, q5, q6, q7, q8

        def run(self):
                for qsys in self.qstream:
```

(continues on next page)

(continued from previous page)

```
                # send the encoded qubits to Bob
                for qubit in self.shor_encode(qsys):
                        self.qsend(bob, qubit)
```

```
class DumbAlice(Agent):
        '''DumbAlice sends a state to Bob but forgets to error-correct!'''
        def run(self):
                for qsys in self.qstream:
                        for qubit in qsys.qubits:
                                self.qsend(dumb_bob, qubit)
```

Now let's define Bob's behavior:

```
class Bob(Agent):
        '''Bob receives Alice's qubits and applied error correction'''
        def shor_decode(self, psi, q1, q2, q3, q4, q5, q6, q7, q8):
                # same enumeration as Alice
                CNOT(psi, q1)
                CNOT(psi, q2)
                TOFFOLI(q2, q1, psi)
                CNOT(q3, q4)
                CNOT(q3, q5)
                TOFFOLI(q5, q4, q3)
                CNOT(q6, q7)
                CNOT(q6, q8)
                TOFFOLI(q7, q8, q6) # Toffoli control qubit order doesn't matter
                H(psi)
                H(q3)
                H(q6)
                CNOT(psi, q3)
                CNOT(psi, q6)
                TOFFOLI(q6, q3, psi)
                return psi # psi is now Alice's original state

        def run(self):
                measurement_results = []
                for _ in self.qstream:
                        # Bob receives 9 qubits representing Alice's encoded state
                        received = [self.qrecv(alice) for _ in range(9)]
                        # Decode and measure the original state
                        psi_true = self.shor_decode(*received)
                        measurement_results.append(psi_true.measure())
                self.output(measurement_results)
```

```
class DumbBob(Agent):
        '''DumbBob receives a state from Alice but does not error-correct'''
        def run(self):
                measurement_results = []
                for _ in self.qstream:
                        received = [self.qrecv(dumb_alice) for _ in range(9)]
                        psi_true = received[0]
                        measurement_results.append(psi_true.measure())
                self.output(measurement_results)
```

Now we need to make an error model to simulate the qubit corruption. SQUANCH includes base classes for defining error models and quantum/classical channels. In this demonstration, we'll only use a quantum error, from the base class `QError`, and a quantum channel model, from the base class `QChannel`. Let's start with the error model, which can apply a random unitary operation to a single qubit in each group of nine.

```
class ShorError(QError):

        def __init__(self, qchannel):
                '''
                Instatiate the error model from the parent class
                :param QChannel qchannel: parent quantum channel
                '''
                QError.__init__(self, qchannel)
                self.count = 0
                self.error_applied = False

        def apply(self, qubit):
                '''
                Apply a random unitary operation to one of the qubits in a set of 9
                :param Qubit qubit: qubit from quantum channel
                :return: either unchanged qubit or None
                '''
                # reset error for each group of 9 qubits
                if self.count == 0:
                        self.error_applied = False
                self.count = (self.count + 1) % 9
                # qubit could be None if combining with other error models, such as␣
↪attenuation
                if not self.error_applied and qubit is not None:
                        if np.random.rand() < 0.5: # apply the error
                                random_unitary = unitary_group.rvs(2) # pick a random␣
↪U(2) matrix
                                qubit.apply(random_unitary)
                                self.error_applied = True
                return qubit
```

Adding this error to a channel model is simple: simply call `__init__` of the parent channel class and add the error class to the `self.errors` list:

```
class ShorQChannel(QChannel):
        '''Represents a quantum channel with a Shor error applied'''

        def __init__(self, from_agent, to_agent):
                QChannel.__init__(self, from_agent, to_agent)
                # register the error model
                self.errors = [ShorError(self)]
```

Before we move on, let's make some helper functions:

```
def to_bits(string):
        '''Convert a string to a list of bits'''
        result = []
        for c in string:
                bits = bin(ord(c))[2:]
                bits = '00000000'[len(bits):] + bits
                result.extend([int(b) for b in bits])
        return result

def from_bits(bits):
        '''Convert a list of bits to a string'''
```

(continues on next page)

(continued from previous page)

```
        chars = []
        for b in range(int(len(bits) / 8)):
                byte = bits[b*8:(b+1)*8]
                chars.append(chr(int(''.join([str(bit) for bit in byte]), 2)))
        return ''.join(chars)
```

Now let's prepare a set of states for Alice to transmit to Bob. Since each qsystem has 9 qubits – much larger than in the other demonstrations – we don't want to make anything too large, so a small text message is suitable.

```
    # Prepare a message to send
    msg = "Peter Shor once lived in Ruddock 238! But who was Airman?"
    bits = to_bits(msg)

    # Encode the message as spin eigenstates
qstream = QStream(9, len(bits)) # 9 qubits per encoded state
for bit, qsystem in zip(bits, qstream):
    if bit == 1:
        X(qsystem.qubit(0))
```

Finally, we need to instantiate Alice, DumbAlice, Bob, and DumbBob. We'll make a copy of `mem` for DumbAlice and DumbBob to use since they can't be trusted with the real thing. (Otherwise, manipulations done by DumbAlice would affect Bob's memory/QStream/qubits.)

```
# Alice and Bob will use error correction
out = Agent.shared_output()
alice = Alice(qstream, out)
bob = Bob(qstream, out)
alice.qconnect(bob, ShorQChannel)

# Dumb agents won't use error correction
qstream2 = copy.deepcopy(qstream)
dumb_alice = DumbAlice(qstream2, out)
dumb_bob = DumbBob(qstream2, out)
dumb_alice.qconnect(dumb_bob, ShorQChannel)
```

Finally, let's run the simulation!

```
Simulation(dumb_alice, dumb_bob, alice, bob).run()

print("DumbAlice sent:   {}".format(msg))
print("DumbBob received: {}".format(from_bits(out["DumbBob"])))
print("Alice sent:       {}".format(msg))
print("Bob received:     {}".format(from_bits(out["Bob"])))
```

```
DumbAlice sent:   Peter Shor once lived in Ruddock 238! But who was Airman?
DumbBob received: (Unicode characters are incompatible with LaTeX; see documentation␣
↪site)
Alice sent:       Peter Shor once lived in Ruddock 238! But who was Airman?
Bob received:     Peter Shor once lived in Ruddock 238! But who was Airman?
```

3.4.3 Source code

The full source code for this demonstration is available in the demos directory of the SQUANCH repository.

CHAPTER

FOUR

SQUANCH API REFERENCE

4.1 `Agent` – Alice and Bob in code

class `squanch.agent.`**`Agent`**(*qstream*, *out=None*, *name=None*, *data=None*)

Bases: `multiprocessing.context.Process`

Represents an entity (Alice, Bob, etc.) that can send messages over classical and quantum communication channels. Agents have the following properties:

- Incoming and outgoing classical and quantum channels connecting them to other agents
- Classical memory, implemented simply as a Python dictionary
- Quantum memory, implemented as a Python dictionary of qubits stored in keys of agent names
- Runtime logic in the form of an Agent.run() method

`__eq__`(*other*)

Agents are compared for equality by their names.

`__hash__`()

Agents are hashed by their (unique) names

`__init__`(*qstream*, *out=None*, *name=None*, *data=None*)

Instantiate an Agent from a unique identifier and a shared memory pool

Parameters

- **`qstream`** (`QStream`) – the QStream object that the agent operates on
- **`out`** (*`dict`*) – shared output dictionary to pass to Agent processes to allow for "returns". Default: {}
- **`name`** (*`str`*) – the unique identifier for the Agent. Default: class name
- **`data`** (*any*) – data to pass to the Agent's process, stored in `self.data`. Default: None

`__ne__`(*other*)

Agents are compared for inequality by their names

`cconnect`(*other*, *channel=<class 'squanch.channels.CChannel'>*, ***kwargs*)

Connect Alice and Bob bidirectionally with a specified classical channel model

Parameters

- **`other`** (`Agent`) – the other agent to connect to
- **`channel`** (`CChannel`) – the classical channel model to use
- **`**kwargs`** – optional channel arguments

crecv(*origin*)
Receive a serializable object from another connected agent. `self.time` is updated upon calling this method.

Parameters **origin** (`Agent`) – The agent that previously sent the qubit

Returns the retrieved object, which is also stored in `self.cmem`

csend(*target*, *thing*)
Send a serializable object to another agent. The transmission time is updated by (number of bits) pulse lengths.

Parameters

- **target** (`Agent`) – the agent to send the transmission to
- **thing** (*any*) – the object to send

increment_progress()
Adds 1 to the current progress

output(*thing*)
Output something to `self.out[self.name]`

Parameters **thing** (*any*) – the thing to put in the dictionary

qconnect(*other*, *channel=<class 'squanch.channels.QChannel'>*, ***kwargs*)
Connect Alice and Bob bidirectionally with a specified quantum channel model

Parameters

- **other** (`Agent`) – the other agent to connect to
- **channel** (`QChannel`) – the quantum channel model to use
- ****kwargs** – optional channel arguments

qrecv(*origin*)
Receive a qubit from another connected agent. `self.time` is updated upon calling this method.

Parameters **origin** (`Agent`) – The agent that previously sent the qubit

Returns the retrieved qubit, which is also stored in `self.qmem`

qsend(*target*, *qubit*)
Send a qubit to another agent. The qubit is serialized and passed through a QChannel to the targeted agent, which can retrieve the qubit with Agent.qrecv(). `self.time` is updated upon calling this method.

Parameters

- **target** (`Agent`) – the agent to send the qubit to
- **qubit** (`Qubit`) – the qubit to send

qstore(*qubit*)
Store a qubit in quantum memory. Equivalent to `self.qmem[self].append(qubit).`

Parameters **qubit** (`Qubit`) – the qubit to store

run()
Runtime logic for the Agent; this method should be overridden in child classes.

static shared_output()
Generate a output dictionary stored in a shared memory pool to distribute among agents in separate processes

Returns an empty multiprocessed Manager.dict()

update_progress(*value*)
Update the progress of this agent in the shared output dictionary. Used in Simulation.progress_monitor().

Parameters **value** – the value to update the progress to (out of a max of len(self.qstream))

4.2 `Channels` – Simulating realistic quantum channels

class squanch.channels.**QChannel**(*from_agent*, *to_agent*, *length=0.0*, *errors=()*)
Bases: `object`

Base class for a quantum channel connecting two agents

__init__(*from_agent*, *to_agent*, *length=0.0*, *errors=()*)
Instantiate the quantum channel

Parameters

- **from_agent** (`Agent`) – sending agent
- **to_agent** (`Agent`) – receiving agent
- **length** (*float*) – length of quantum channel in km; default: 0.0km
- **errors** (`QError[]`) – list of error models to apply to qubits in this channel; default: [] (no errors)

__weakref__
list of weak references to the object (if defined)

get()
Retrieve a qubit by reference from the channel queue, applying errors upon retrieval

Returns tuple: (the qubit with errors applied (possibly `None`), receival time)

put(*qubit*)
Serialize and push qubit into the channel queue

Parameters **qubit** (`Qubit`) – the qubit to send

class squanch.channels.**CChannel**(*from_agent*, *to_agent*, *length=0.0*)
Bases: `object`

Base class for a classical channel connecting two agents

__init__(*from_agent*, *to_agent*, *length=0.0*)
Instantiate the quantum channel

Parameters

- **from_agent** (`Agent`) – sending agent
- **to_agent** (`Agent`) – receiving agent
- **length** (*float*) – length of fiber optic line in km; default: 0.0km

__weakref__
list of weak references to the object (if defined)

get()
Retrieve a classical object form the queue

Returns tuple: (the object, receival time)

put (*thing*)
Serialize and push a serializable object into the channel queue

Parameters **thing** (*any*) – the qubit to send

class squanch.channels.**FiberOpticQChannel** (*from_agent*, *to_agent*, *length=0.0*)
Bases: *squanch.channels.QChannel*

Represents a fiber optic line with attenuation errors

__init__ (*from_agent*, *to_agent*, *length=0.0*)
Instantiate the simulated fiber optic quantum channel

Parameters

- **from_agent** (Agent) – sending agent
- **to_agent** (Agent) – receiving agent
- **length** (*float*) – length of fiber optic channel in km; default: 0.0km

4.3 `Errors` – Quantum errors in channels

class squanch.errors.**QError** (*qchannel*)
Bases: object

A generalized quantum error model

__init__ (*qchannel*)
Base initialization class; extend in child methods by overwriting along with QError.__init__(self, qchannel)

Parameters **qchannel** – the quantum channel this error model is being used on

__weakref__
list of weak references to the object (if defined)

apply (*qubit*)
Applies the error to the transmitted qubit. Overwrite this method in child classes while maintaining the Qubit->(Qubit | None) signature

Parameters **qubit** (Qubit) – the qubit being withdrawn from the quantum channel with channel.get(); possibly None

Returns the modified qubit

class squanch.errors.**AttenuationError** (*qchannel*, *attenuation_coefficient=-0.16*)
Bases: *squanch.errors.QError*

Simulate the possible loss of a qubit in a fiber optic channel due to attenuation effects

__init__ (*qchannel*, *attenuation_coefficient=-0.16*)
Instatiate the error class

Parameters

- **qchannel** (QChannel) – parent quantum channel
- **attenuation_coefficient** (*float*) – attenuation of fiber in dB/km; default: -.16 dB/km, from Yin, et al

apply (*qubit*)
Simulates possible loss + measurement of qubit

Parameters **qubit** (Qubit) – qubit from quantum channel

Returns either unchanged qubit or None

class squanch.errors.**RandomUnitaryError**(*qchannel*, *variance*)
Bases: *squanch.errors.QError*

Simualates a random rotation along X and Z with a Gaussian distribution of rotation angles

__init__(*qchannel*, *variance*)
Instatiate the error class

Parameters

- **qchannel** (QChannel) – parent quantum channel
- **variance** (*float*) – variance to use in the Gaussian sampling of X and Z rotation angles

apply(*qubit*)
Simulates random rotations on X and Z of a qubit

Parameters **qubit** (Qubit) – qubit from quantum channel

Returns rotated qubit

class squanch.errors.**SystematicUnitaryError**(*qchannel*, *operator=None*, *variance=None*)
Bases: *squanch.errors.QError*

Simulates a random unitary error that is the same for each qubit

__init__(*qchannel*, *operator=None*, *variance=None*)
Instantiate the systematic unitary error class

Parameters

- **qchannel** (QChannel) – parent quantum channel
- **operator** (*np.array*) –
- **variance** (*float*) –

apply(*qubit*)
Simulates the application of the unitary error

Parameters **qubit** (Qubit) – qubit from quantum channel

Returns rotated qubit

4.4 `Gates` – Manipulating quantum states

squanch.gates.**H**(*qubit*)
Applies the Hadamard transform to the specified qubit, updating the qsystem state. `cache_id`: `H`

Parameters **qubit** (Qubit) – the qubit to apply the operator to

squanch.gates.**X**(*qubit*)
Applies the Pauli-X (NOT) operation to the specified qubit, updating the qsystem state. `cache_id`: `X`

Parameters **qubit** (Qubit) – the qubit to apply the operator to

squanch.gates.**Y**(*qubit*)
Applies the Pauli-Y operation to the specified qubit, updating the qsystem state. `cache_id`: `Y`

Parameters **qubit** (Qubit) – the qubit to apply the operator to

`squanch.gates.`**`Z`**`(`*qubit*`)`
Applies the Pauli-Z operation to the specified qubit, updating the qsystem state. `cache_id`: `Z`

Parameters **`qubit`** (`Qubit`) – the qubit to apply the operator to

`squanch.gates.`**`RX`**`(`*qubit*, *angle*`)`
Applies the single qubit X-rotation operator to the specified qubit, updating the qsystem state. `cache_id`: `Rx*`, where * is angle/pi

Parameters

- **`qubit`** (`Qubit`) – the qubit to apply the operator to
- **`angle`** (*`float`*) – the angle by which to rotate

`squanch.gates.`**`RY`**`(`*qubit*, *angle*`)`
Applies the single qubit Y-rotation operator to the specified qubit, updating the qsystem state. `cache_id`: `Ry*`, where * is angle/pi

Parameters

- **`qubit`** (`Qubit`) – the qubit to apply the operator to
- **`angle`** (*`float`*) – the angle by which to rotate

`squanch.gates.`**`RZ`**`(`*qubit*, *angle*`)`
Applies the single qubit Z-rotation operator to the specified qubit, updating the qsystem state. `cache_id`: `Rz*`, where * is angle/pi

Parameters

- **`qubit`** (`Qubit`) – the qubit to apply the operator to
- **`angle`** (*`float`*) – the angle by which to rotate

`squanch.gates.`**`PHASE`**`(`*qubit*, *angle*`)`
Applies the phase operation from control on target, mapping |1> to e^(i*angle)|1>. `cache_id`: `PHASE*`, where * is angle/pi

Parameters

- **`qubit`** (`Qubit`) – the qubit to apply the operator to
- **`angle`** (*`float`*) – the phase angle to apply

`squanch.gates.`**`CNOT`**`(`*control*, *target*`)`
Applies the controlled-NOT operation from control on target. This gate takes two qubit arguments to construct an arbitrary CNOT matrix. `cache_id`: `CNOTi,j,N`, where i and j are control and target indices and N is num_qubits

Parameters

- **`control`** (`Qubit`) – the control qubit
- **`target`** (`Qubit`) – the target qubit, with Pauli-X applied according to the control qubit

`squanch.gates.`**`TOFFOLI`**`(`*control1*, *control2*, *target*`)`
Applies the Toffoli (or controlled-controlled-NOT) operation from control on target. This gate takes three qubit arguments to construct an arbitrary CCNOT matrix. `cache_id`: `CCNOTi,j,k,N`, where i and j are control indices and k target indices and N is num_qubits

Parameters

- **`control1`** (`Qubit`) – the first control qubit
- **`control2`** (`Qubit`) – the second control qubit

- **`target`** (`Qubit`) – the target qubit, with Pauli-X applied according to the control qubit

`squanch.gates.`**`CU`**(*control*, *target*, *unitary*)

Applies the controlled-unitary operation from control on target. This gate takes control and target qubit arguments and a unitary operator to apply `cache_id`: `CUi,j,<str(unitary)>,N`, where i and j are control and target indices and N is num_qubits

Parameters

- **`control`** (`Qubit`) – the control qubit
- **`target`** (`Qubit`) – the target qubit
- **`unitary`** (*`np.array`*) – the unitary single-qubit gate to apply to the target qubit

`squanch.gates.`**`CPHASE`**(*control*, *target*, *angle*)

Applies the controlled-phase operation from control on target. This gate takes control and target qubit arguments and a rotation angle, and calls CU(control, target, np.array([[1, 0], [0, np.exp(1j * angle)]])).

Parameters

- **`control`** (`Qubit`) – the control qubit
- **`target`** (`Qubit`) – the target qubit
- **`angle`** (*`float`*) – the phase angle to apply

`squanch.gates.`**`SWAP`**(*q1*, *q2*)

Applies the SWAP operator to two qubits, switching the states. This gate is implemented by three CNOT operations and thus has no `cache_id`. :param q1: the first qubit :param q2: the second qubit

`squanch.gates.`**`expand`**(*operator*, *index*, *num_qubits*, *cache_id=None*)

Apply a k-qubit quantum gate to act on n-qubits by filling the rest of the spaces with identity operators

Parameters

- **`operator`** (*`np.array`*) – the single- or n-qubit operator to apply
- **`index`** (*`int`*) – if specified, the index of the qubit to perform the operation on
- **`num_qubits`** (*`int`*) – the number of qubits in the system
- **`cache_id`** (*`str`*) – a character identifier to cache gates and their expansions in memory

Returns the expanded n-qubit operator

4.5 `Linalg` – Useful linear algebra functions for QM

`squanch.linalg.`**`is_hermitian`**(*matrix*)

Checks if an operator is Hermitian

Parameters **`matrix`** (*`np.array`*) – the operator to check

Returns true or false

`squanch.linalg.`**`tensor_product`**(*state1*, *state2*)

Returns the Kronecker product of two states

Parameters

- **`state1`** (*`np.array`*) – the first state
- **`state2`** (*`np.array`*) – the second state

Returns the tensor product

`squanch.linalg.`**`tensors`**(*operator_list*)

Returns the iterated Kronecker product of a list of states

Parameters **`operator_list`** (*`[np.array]`*) – list of states to tensor-product

Returns the tensor product

`squanch.linalg.`**`tensor_fill_identity`**(*single_qubit_operator*, *n_qubits*, *qubit_index*)

Create the n-qubit operator I x I x ... Operator x I x I... with operator applied to a given qubit index

Parameters

- **`single_qubit_operator`** (*`np.array`*) – the operator in the computational basis (a 2x2 matrix)
- **`n_qubits`** (*`int`*) – the number of qubits in the system to fill
- **`qubit_index`** (*`int`*) – the zero-indexed qubit to apply this operator to

Returns the n-qubit operator

4.6 `Simulate` – Parallelized quantum network simulation

`class` `squanch.simulate.`**`Simulation`**(**args*)

Bases: `object`

Simulation class for easily creating and running agent-based simulations. Includes progress monitors for terminal and Jupyter notebooks.

`__init__`(**args*)

Initialize the simulation

Parameters **`args`** – unpacked list of agents, e.g. Simulation(alice, bob, charlie). All agents must share the same output dictionary using Agent.shared_output()

`__weakref__`

list of weak references to the object (if defined)

`progress_monitor`(*poison_pill*)

Display a tqdm-style progress bar in a Jupyter notebook

Parameters **`poison_pill`** (*`threading.Event`*) – a flag to kill the progressMonitor thread

`run`(*monitor_progress=True*)

Run the simulation

Parameters **`monitor_progress`** – whether to display a progress bar for each agent

4.7 `QStream` – Working with quantum datastreams

`class` `squanch.qstream.`**`QStream`**(*system_size*, *num_systems*, *array=None*, *agent=None*)

Bases: `object`

Efficiently represents many separable quantum subsystems in a contiguous block of shared memory. `` QSystem``s and ``Qubit``s can be instantiated from the ``state `` of this class.

`__init__`(*system_size*, *num_systems*, *array=None*, *agent=None*)

Instantiate the quantum datastream object

Parameters

- **system_size** (*int*) – number of entangled qubits in each quantum system; each system has dims 2^system_size
- **num_systems** (*int*) – number of small quantum systems in the data stream
- **array** (*np.array*) – pre-allocated array in memory for purposes of sharing QStreams in multiprocessing
- **agent** (*Agent*) – optional reference to the Agent owning the qstream; useful for progress monitoring across separate processes

__iter__()
Iterates over the ‘‘QSystem‘‘s in this class instance

Returns each system in the stream

__len__()
Custom length method for streams; equivalent to stream.num_systems

Returns stream.num_systems

__weakref__
list of weak references to the object (if defined)

classmethod from_array(*array*, *reformat=False*, *agent=None*)
Instantiates a quantum datastream object from an existing state array

Parameters

- **array** (*np.array*) – the pre-allocated np.complex64 array representing the shared Hilbert space
- **reformat** (*bool*) – if providing a pre-allocated array, whether to reformat it to the all-zero state

Returns the child QStream

next()
Access the next element in the quantum stream, returning it as a QSystem object, and increment the head by 1

Returns a QSystem for the "head" system

static reformat()
Reformats a Hilbert space array in-place to the all-zero state

Parameters array (*np.array*) – a num_systems x 2^system_size x 2^system_size array of np.complex64 values

static shared_hilbert_space(*num_systems*)
Allocate a portion of shareable c-type memory to create a numpy array that is sharable between processes

Parameters

- **system_size** (*int*) – number of entangled qubits in each quantum system; each has dimension 2^system_size
- **num_systems** (*int*) – number of small quantum systems in the data stream

Returns a blank, sharable, num_systems * 2^system_size * 2^system_size array of np.complex64 values

system(*index*)
Access the nth quantum system in the quantum datastream object

Parameters index (*int*) – zero-index of the quantum system to access

Returns the quantum system

4.8 `Qubit` – Qubits and quantum systems

class `squanch.qubit.`**`QSystem`**(*num_qubits*, *index=None*, *state=None*)

Bases: `object`

Represents a multi-body, maximally-entangleable quantum system. Contains references to constituent qubits and (if applicable) its parent `QStream`. Quantum state is represented as a density matrix in the computational basis.

`__init__`(*num_qubits*, *index=None*, *state=None*)

Instatiate the quantum state for an n-qubit system

Parameters

- **num_qubits** (*int*) – number of qubits in the system, treated as maximally entangled
- **index** (*int*) – index of the QSystem within the parent QStream
- **state** (*np.array*) – density matrix representing the quantum state. By default, **|000...0><0...000|** is used

`__weakref__`

list of weak references to the object (if defined)

`apply`(*operator*)

Apply an N-qubit unitary operator to this system's N-qubit quantum state

Parameters **operator** (*np.array*) – the unitary N-qubit operator to apply

Returns nothing, the qsystem state is mutated

classmethod from_stream(*qstream*, *index*)

Instantiate a QSystem from a given index in a parent QStream

Parameters

- **qstream** (`QStream`) – the parent stream
- **index** (*int*) – the index in the parent stream corresponding to this system

Returns the QSystem object

`measure_qubit`(*index*)

Measure the qubit at a given index, partially collapsing the state based on the observed qubit value. The state vector is modified in-place by this function.

Parameters **index** (*int*) – the qubit to measure

Returns the measured qubit value

`qubit`(*index*)

Access a qubit by index; self.qubits does not instantiate all qubits unless casted to a list. Use this function to access a single qubit of a given index.

Parameters **index** (*int*) – qubit index to generate a qubit instance for

Returns the qubit instance

class `squanch.qubit.`**`Qubit`**(*qsystem*, *index*)

Bases: `object`

A wrapper class representing a single qubit in an existing quantum system.

`__init__`(*qsystem*, *index*)

Instantiate the qubit from an existing QSystem and index

Parameters

- **`qsystem`** (`QSystem`) – n-qubit quantum system that this qubit points to
- **`index`** (*`int`*) – particle index in the quantum system, ranging from 0 to n-1

`__weakref__`

list of weak references to the object (if defined)

`apply`(*operator*, *id=None*)

Apply a single-qubit operator to this qubit, tensoring with I and passing to the qsystem.apply() method

Parameters

- **`operator`** (*`np.array`*) – a single qubit (2x2) complex-valued matrix
- **`cacheID`** (*`str`*) – a character or string to cache the expanded operator by (e.g. Hadamard qubit 2 -> "IHII…")

`classmethod from_stream`(*qstream*, *system_index*, *qubit_index*)

Instantiate a qubit from a parent stream (via a QSystem call)

Parameters

- **`qstream`** (`QStream`) – the parent stream
- **`system_index`** (*`int`*) – the index corresponding to the parent QSystem
- **`qubit_index`** (*`int`*) – the index of the qubit to be recalled

Returns the qubit

`measure`()

Measure a qubit, modifying the density matrix of its parent `QSystem` in-place

Returns the measured value

`serialize`()

Generate a reference to reconstruct this qubit from shared memory

Returns qubit reference as (systemIndex, qubitIndex)

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX