Documentation for the Simulator for Quantum Networks and Channels (SQUANCH)

Release 0.0.0

Ben Bartlett

CONTENTS:

CHAPTER ONE

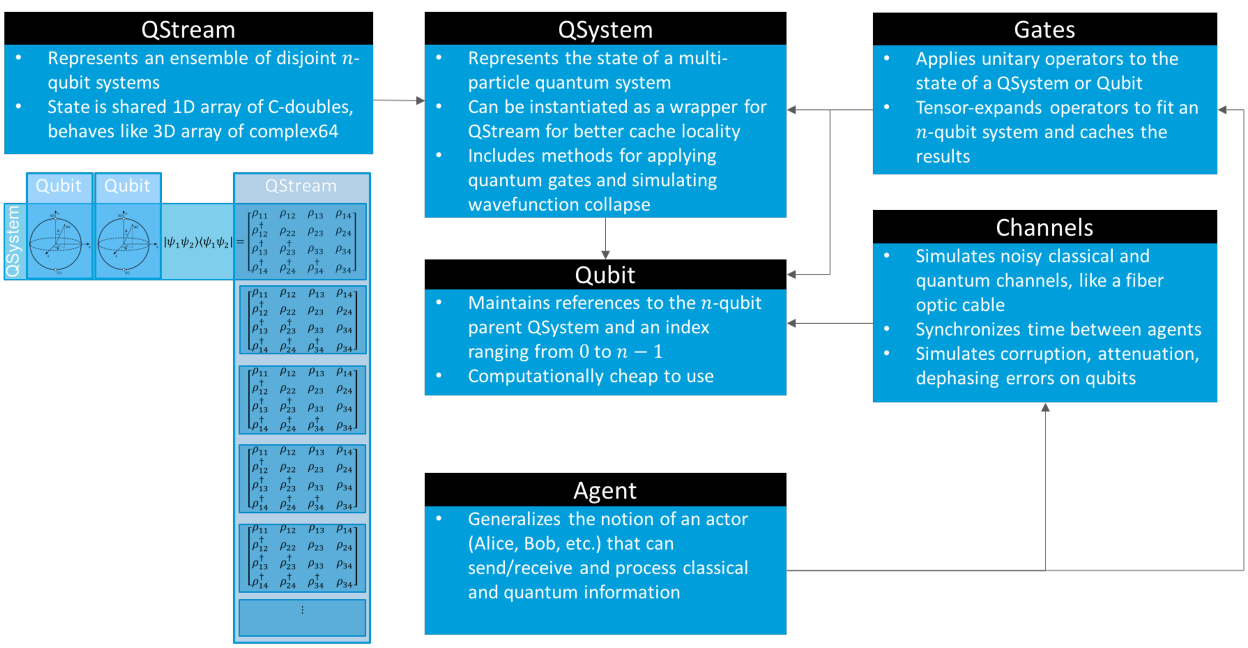
OVERVIEW

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 simulations of quantum information processing and transmission. Although it can be used as a general-purpose quantum computing simulation library, SQUANCH is designed for simulating quantum *networks*, acting as a sort of simulated quantum playground for you to test ideas for quantum transmission and networking protocols. For this purpose, it includes a number of extensible and flexible modules that allow you to intuitively design a quantum network, a range of built-in quantum error simulations to introduce realism and the need for error corrections in your simulations, and lightweight and easily parallelizable systems for manipulating quantum information that allow it to vastly outperform (by a factor of as much as 1000) other frameworks in certain tasks.

SQUANCH is developed as part of the Intelligent Quantum Networks and Technologies (INQNET) initiative, a collaboration between AT&T and the California Institute of Technology. The source is hosted on GitHub (INSERT LINK ONCE WE HAVE NEW REPO).

1.2 Design Overview



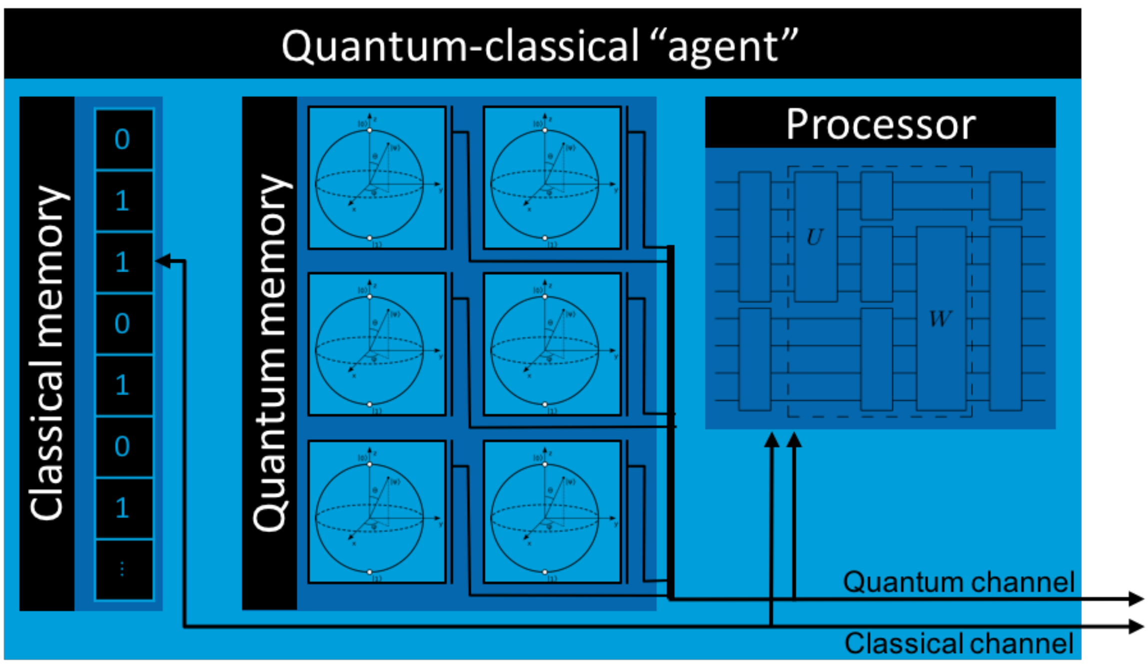
1.2.1 Information Representation and Processing

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 a collection of millions of 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 with 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.

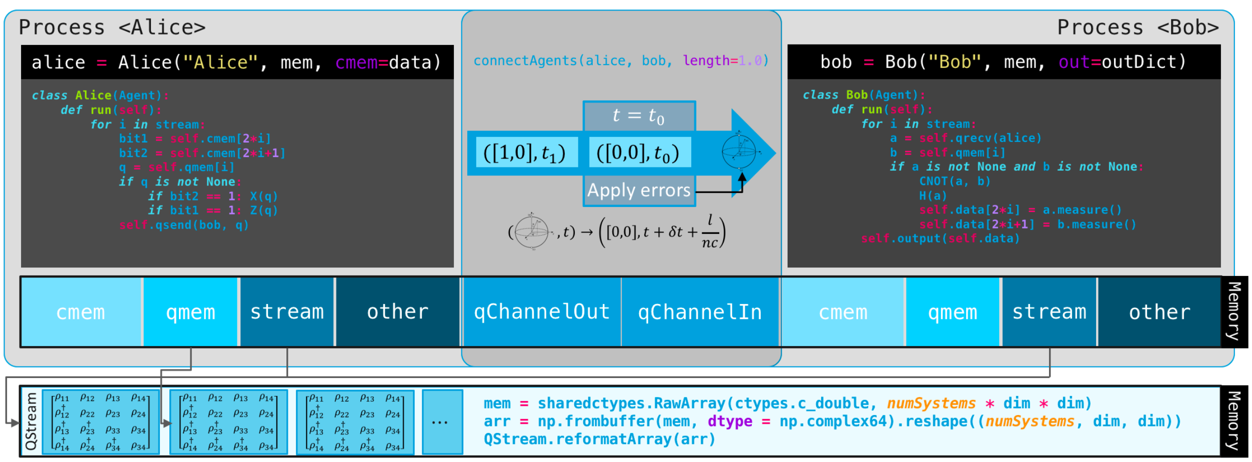
1.2.3 Memory Structure and Time Synchronization

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 that represent the non-local combined quantum state shared between agents. 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 `sharedHilbertSpace()` function in the `QStream` module. This creates 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 $1.5+10^5\cdot10^{-5} = 2.5\mu s$, and Bob's accounts for a speed of light delay to update to $2.0+10^5\cdot10^{-5}+\frac{300m}{c} = 4\mu s$.

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



CHAPTER

TWO

GETTING STARTED

2.1 Requirements

SQUANCH is programmed in Python 2.7 and NumPy. You can obtain both of these, along with a host of other scientific computing tools, from the Anaconda package.

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 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.qubit import *
from squanch.gates 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 on 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>
```

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.qubit import *
from squanch.gates 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.qstream import *
from squanch.gates 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:

```
firstSys = stream.system(2)
H(firstSys.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.

```
import time
from squanch.qstream import *
from squanch.qubit import *
from squanch.gates import *

numSystems = 100000
```

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

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

```
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, and its core feature set is the ability to easily simulate agents manipulating and transferring quantum inforamtion between each other concurrently.

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.agent import *
from squanch.gates import *

def stringToBits(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

def bitsToString(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
```

```
message = "Hello, Bob!"
msgBits = stringToBits(message)
```

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

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


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

Finally, to instantiate and run the agents, we need to name them (if no name is provided in the class call, it defaults to the name of the class, e.g. `Alice(...).name == "Alice"`) and we need to make an appropriately sized `sharedHilbertSpace` and a `sharedOutputDict` to pass to the agents. We then connect the agents (using a channel length of 0 to ignore speed-of-light delays and attenuation errors) and run their processes:

```
mem = sharedHilbertSpace(1, len(msgBits))
out = sharedOutputDict()

alice = Alice(mem, data = msgBits)
bob = Bob(mem, out = out)

connectAgents(alice, bob, length = 0.0)

alice.start(); bob.start()
alice.join(); bob.join()

receivedMessage = bitsToString(out["Bob"])
print "Alice sent: '{}'. Bob received: '{}'.".format(message, receivedMessage)
```

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

2.8 See also

This tutorial page only touches on some very 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 a quantum state using classical communication. This process has tremendous applicability to quantum networks, which will need to transfer fragile quantum states between distant nodes. Conecptually, quantum teleportation is the inverse of *superdense coding*.

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

3.1.1 Protocol

Below is a simple two-party quantum teleportation protocol. We'll be using the above circuit diagram.

1. Alice generates an EPR pair; for this protocol, we'll use the state $|q_1 q_2\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. She will keep one particle in the pair and send the other one to Bob.
2. Alice entangles her qubit q_0 with her ancilla q_1 by applying controlled-not and Hadamard operators.
3. Alice measures both of her qubits and communicates the results (two bits) to Bob through a classical channel. Bob's qubit is now in one of four possible states, one of which is $|q_0\rangle$. Bob will use Alice's two bits to determine what operations to apply to recover $|q_0\rangle$.
4. Bob applies a Pauli-X operator to his qubit if Alice's ancilla collapsed to $|q_1\rangle = |1\rangle$, and he applies a Pauli-Z operator to his qubit if her qubit collapsed to $|q_0\rangle = |1\rangle$.

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.agent import *
from squanch.gates import *
from squanch.qstream 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 teleport(self, qSystem):
        # Generate a Bell pair and send half of it to Bob
        q, a, b = qSystem.qubits
        H(a)
        CNOT(a, b)
        self.qsend(bob, b)
        # Perform the teleportation
        CNOT(q, a)
        H(q)
        bobZ = q.measure() # If Bob should apply Z
        bobX = a.measure() # If Bob should apply X
        self.csend(bob, [bobX, bobZ])

    def run(self):
        for qSys in self.stream:
            self.teleport(qSys)
```

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

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

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

    def run(self):
        measurementResults = []
        for _ in self.stream:
            b = self.qrecv(alice)
            doX, doZ = self.crecv(alice)
            if doX and b is not None: X(b)
            if doZ and b is not None: Z(b)
            measurementResults.append(b.measure())
        self.output(measurementResults)
```

This logic will allow Alice and Bob to act on a common quantum stream to teleport states to each other. Now we want to actually instantiate a quantum stream and manipulate the initial state of the first qubit (the one to be teleported) in each system of the stream so that we're not just teleporting the $|0\rangle$ state over and over.

```
# Allocate memory and output structures
mem = sharedHilbertSpace(3, 10)
out = sharedOutputDict()

# Prepare the initial states
stream = QStream.fromArray(mem)
statesList = [1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
for state, qSys in zip(statesList, stream):
    q = qSys.qubit(0)
    if state == 1: X(q)  # Flip the qubits corresponding to 1's
```

Finally, let's create Alice and Bob instances, plug in the Hilbert space and output structures, and run the program. Explicitly allocating and passing memory to agents is necessary because each agent spawns and runs in a separate process, which (in general) have separate memory pools. You'll also need to 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.

```
# Make the agents
alice = Alice(mem)
bob = Bob(mem, out = out)

# Connect the agents
connectAgents(alice, bob, length = 0.0)

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

print "Teleported states {} \n" \
      "Received states   {}".format(statesList, 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]
```

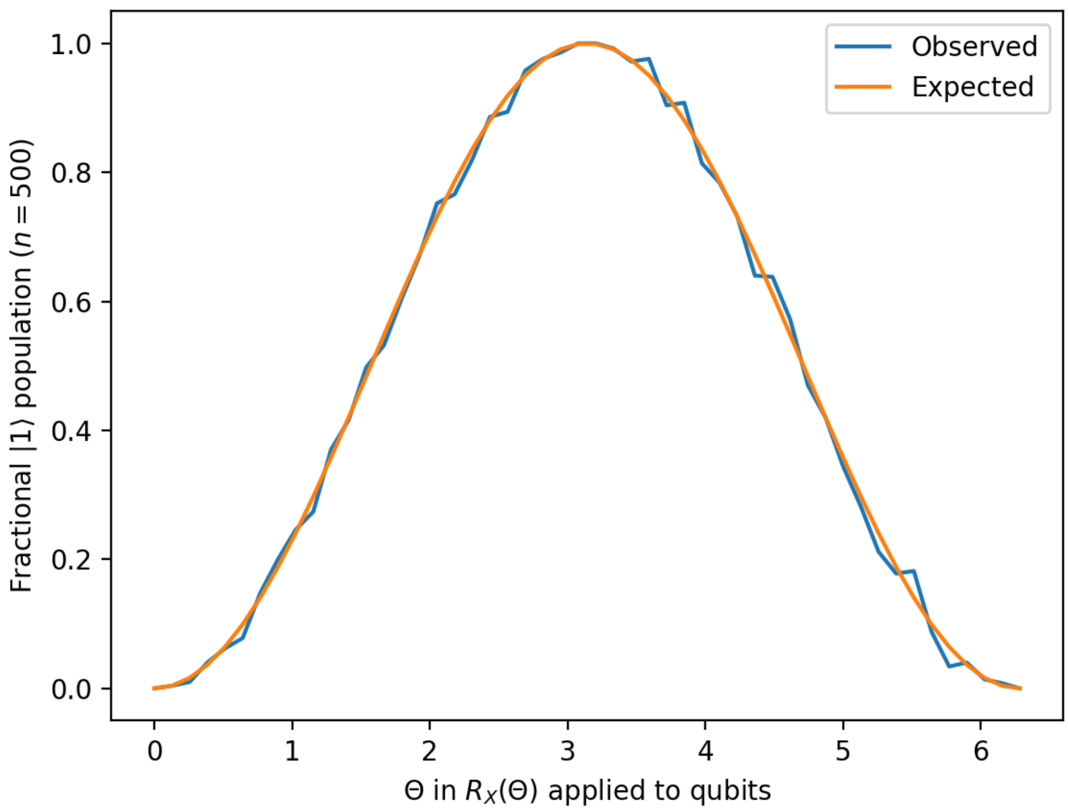
So at least for the simple cases, our implementation seems to be working! Let's do a little more complex test case now.

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.

```
angles = np.linspace(0, 2 * np.pi, 30)  # RX angles to apply
numTrials = 250  # number of trials for each angle

# Allocate memory and output structures
mem = sharedHilbertSpace(3, len(angles) * numTrials)
out = sharedOutputDict()

# Prepare the initial states in the stream
stream = QStream.fromArray(mem)
for angle in angles:
    for _ in range(numTrials):
        q = stream.head().qubit(0)
        RX(q, angle)
stream.index = 0  # reset the head counter

# Make the agents
alice = Alice(mem)
bob = Bob(mem, out = out)

# Connect the agents
connectAgents(alice, bob)

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

results = np.array(out["Bob"]).reshape((len(angles), numTrials))
meanResults = np.mean(results, axis = 1)
expectedResults = np.sin(angles / 2) ** 2
plt.plot(angles, meanResults, label = 'Observed')
plt.plot(angles, expectedResults, 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 sharing an entangled pair can send two classical bits with a single qubit. Conecptually, it is the inverse of *quantum teleportation*.

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.

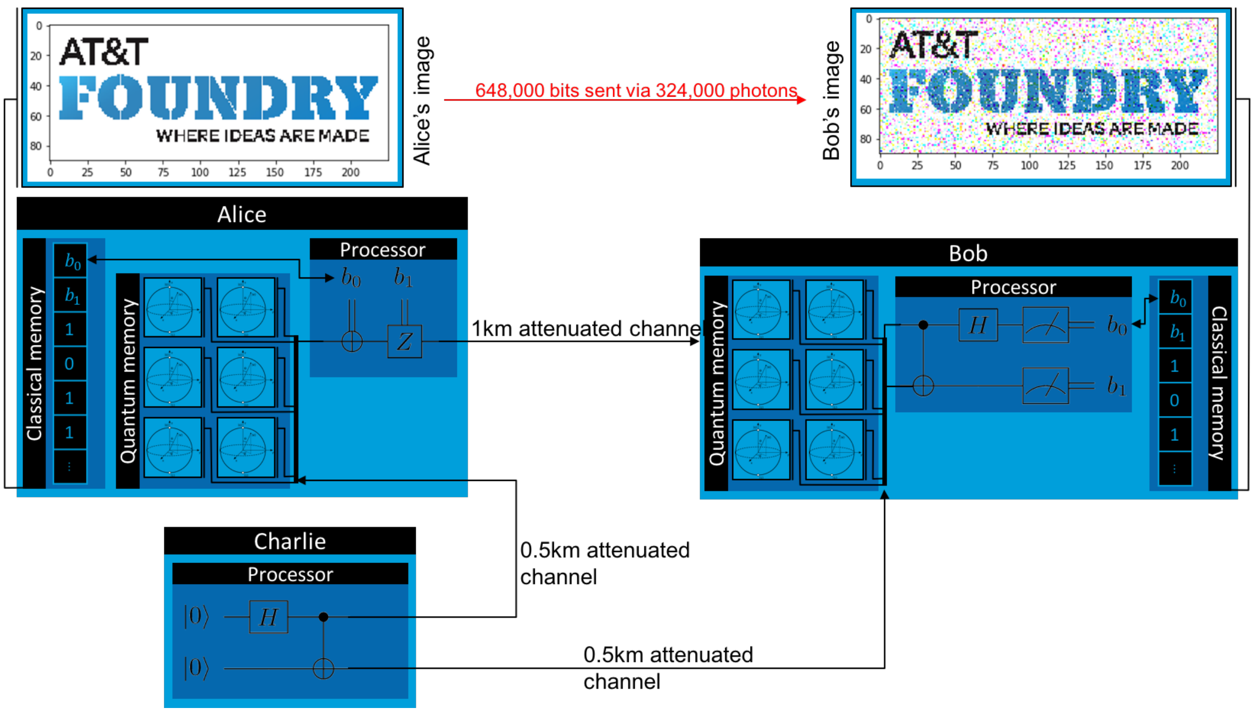
1. Charlie generates EPR pairs in the state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. He sends one particle to Alice and the other to Bob.

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.

3.2.2 Implementation

Because superdense coding transmits classical information, 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.agent import *
from squanch.gates import *
from squanch.qstream 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.stream:
            a, b = qSys.qubits
            H(a)
```

```
            CNOT(a, b)
            self.qsend(alice, a)
            self.qsend(bob, b)
```

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 i in range(len(self.stream)):
            bit1, bit2 = self.data[2 * i], self.data[2 * i + 1]
            q = self.qrecv(charlie)
            if q is not None:
                if bit2 == 1: X(q)
                if bit1 == 1: Z(q)
            self.qsend(bob, q)
```

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 *sharedOutputDict* that is provided to agents on instantiation.

```
class Bob(Agent):
    '''Bob receives Alice's transmissions and reconstructs her information'''
    def run(self):
        self.data = np.zeros(2 * len(self.stream), dtype = np.uint8)
        for i in range(len(self.stream)):
            a = self.qrecv(alice)
            c = self.qrecv(charlie)
            if a is not None and c is not None:
                CNOT(a, c)
                H(a)
                self.data[2 * i] = a.measure()
                self.data[2 * i + 1] = c.measure()
        self.output(self.data)
```

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), and we'll also need to provide the agents with appropriate arguments for the Hilbert space they will share as well as an output structure to push their data to. (This is necessary because all agents run in separate processes, so explicitly shared memory structures must be passed to them.)

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

# Allocate a shared Hilbert space and output object to pass to agents
mem = sharedHilbertSpace(2, len(imgBitstream) / 2)
out = sharedOutputDict()

# Make agent instances
alice = Alice(mem, data = imgBitstream)
bob = Bob(mem, out = out)
charlie = Charlie(mem)
```

Let's connect the agents with some simulated length parameter (for time simulation purposes and for application of errors). 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. 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()*.

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

# Run the agents
start = time.time()
agents = [alice, bob, charlie]
[agent.start() for agent in agents]
[agent.join() for agent in agents]
print "Transmitted {} bits in {:.3f}s.".format(len(out["Bob"]), time.time() - start)
```

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

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

CHAPTER

FOUR

SQUANCH API REFERENCE

4.1 `Agent` – Alice and Bob in code

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

Bases: `multiprocessing.process.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 communciation lines to other agents
- Incoming and outgoing quantum channels to other agents through which entangled pairs may be distributed
- Ideal classical memory
- Quantum memory with some characteristic corruption timescale

`__eq__`(*other*)

Agents are compared for equality by their names.

`__hash__`()

Agents are hashed by their names, which is why they must be unique.

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

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

Parameters

- **`hilbertSpace`** (*`np.array`*) – the shared memory pool representing the Hilbert space of the qstream
- **`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
- **`out`** (*`dict`*) – shared output dictionary to pass to Agent processes to allow for "returns". Default: None

`__module__` **= 'squanch.agent'**

`__ne__`(*other*)

Overridden inequality operator, for good practice.

`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

incrementProgress()
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

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()
This method should be overridden in extended class instances, and cannot take any arguments or return any values.

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

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

`squanch.agent.`**connectAgents**(*alice*, *bob*, *length=0.0*)
Connect Alice and Bob bidirectionally via a simulated fiber optic line

Parameters

- **alice** (`Agent`) – the first Agent
- **bob** (`Agent`) – the second Agent
- **length** (*float*) – the length of the simulated cable in km; default value: 0.0km

`squanch.agent.`**sharedHilbertSpace**(*systemSize*, *numSystems*)
Allocate a portion of shareable c-type memory to create a numpy array that is sharable between processes

Parameters

- **systemSize** (*int*) – number of entangled qubits in each quantum system; each system has dimension 2^systemSize

- **numSystems** (*int*) – number of small quantum systems in the data stream

Returns a blank, sharable, numSystems x 2^systemSize x 2^systemSize array of np.complex64 values

`squanch.agent.`**sharedOutputDict**()

Generate a shared output dictionary to distribute among agents in separate processes

Returns an empty multiprocessed Manager.dict()

4.2 `Channels` – Simulating realistic quantum channels

class `squanch.channels.`**CChannel**(*fromAgent*, *toAgent*, *length=0.0*)

Base class for a classical channel

__init__(*fromAgent*, *toAgent*, *length=0.0*)

Instantiate the quantum channel

Parameters

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

__module__ = 'squanch.channels'

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**(*fromAgent*, *toAgent*, *length=0.0*)

Bases: *squanch.channels.QChannel*

Represents a fiber optic line with attenuation errors

__init__(*fromAgent*, *toAgent*, *length=0.0*)

Instantiate the simulated fiber optic quantum channel

Parameters

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

__module__ = 'squanch.channels'

class `squanch.channels.`**QChannel**(*fromAgent*, *toAgent*, *length=0.0*, *errors=[]*)

Base class for a quantum channel

__init__(*fromAgent*, *toAgent*, *length=0.0*, *errors=[]*)

Instantiate the quantum channel

Parameters

- **fromAgent** (Agent) – sending agent

- **toAgent** (`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)

`__module__` = 'squanch.channels'

`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

4.3 `Errors` – Quantum errors in channels

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

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

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

Parameters

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

`__module__` = 'squanch.errors'

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

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

Returns either unchanged qubit or None

class `squanch.errors.`**`QError`**(*qchannel*)
A generalized 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

`__module__` = 'squanch.errors'

`apply`(*qubit*)
Generic apply method; overwrite in child methods 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.**RandomUnitaryError**(*qchannel*, *randomUnitarySigma*)

Bases: *squanch.errors.QError*

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

__init__(*qchannel*, *randomUnitarySigma*)

Instatiate the error class

Parameters

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

__module__ = 'squanch.errors'

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*, *unitaryOperation=None*, *randomUnitarySigma=None*)

Bases: *squanch.errors.QError*

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

__init__(*qchannel*, *unitaryOperation=None*, *randomUnitarySigma=None*)

Instantiate the systematic unitary error class

Parameters

- **qchannel** (QChannel) – parent quantum channel
- **unitaryOperation** (*np.array*) –
- **randomUnitarySigma** (*float*) –

__module__ = 'squanch.errors'

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.**CNOT**(*control*, *target*)

Applies the controlled-NOT operation from control on target. This gate takes two qubit arguments to construct an arbitrary CNOT matrix. cacheID: `CNOTij`, where i and j are control and target indices

Parameters

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

squanch.gates.**H**(*qubit*)

Applies the Hadamard transform to the specified qubit, updating the qSystem state. cacheID: `H`

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. cacheID: `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. cacheID: `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. cacheID: `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.`**`X`**(*qubit*)

Applies the Pauli-X (NOT) operation to the specified qubit, updating the qSystem state. cacheID: `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. cacheID: `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. cacheID: `Z`

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

`squanch.gates.`**`expandGate`**(*operator*, *index*, *nQubits*, *cacheID=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
- **nQubits** (*int*) – the number of qubits in the system
- **cacheID** (*str*) – a character identifier to cache common gates in memory to avoid having to call tensorFillIdentity

Returns the expanded n-qubit operator

4.5 `Linalg` – Useful linear algebra functions for QM

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

Checks if an operator is Hermitian

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

Returns true or false

`squanch.linalg.`**`tensorFillIdentity`**(*singleQubitOperator*, *nQubits*, *qubitIndex*)

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

Parameters

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

Returns the n-qubit operator

`squanch.linalg.`**`tensorProd`**(*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`**(*operatorList*)

Returns the iterated Kronecker product of a list of states

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

Returns the tensor product

4.6 `Simulate` – Plug-and-play agent simulation

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

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)

`__module__` **= 'squanch.simulate'**

`progressMonitor`(*poisonPill*)

Display a tqdm-style progress bar in a Jupyter notebook

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

`run`(*monitorProgress=True*)

Run the simulation

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

4.7 `QStream` – Working with quantum datastreams

class `squanch.qstream.`**`QStream`**(*systemSize*, *numSystems*, *array=None*, *reformatArray=False*)

Efficiently handle a large number of small entangled quantum systems to avoid having to perform many class instantiations when simulating transmission of data through quantum channels

`__init__`(*systemSize*, *numSystems*, *array=None*, *reformatArray=False*)

Instantiate the quantum datastream object

Parameters

- **`systemSize`** (*`int`*) – number of entangled qubits in each quantum system; each system has dimension 2^systemSize
- **`numSystems`** (*`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

`__iter__`()

Custom iterator method for streams

Returns each system in the stream

`__len__`()

Custom length method for streams; equivalent to stream.numSystems

Returns stream.numSystems

`__module__` **= 'squanch.qstream'**

classmethod **`fromArray`**(*array*, *reformatArray=False*)

Instantiates a quantum datastream object from a (typically shared) pre-allocated array

Parameters

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

Returns the child QStream

`head`()

Access the "head" of the quantum system "queue", returning it as a QSystem object, and increment the head by 1

Returns a QSystem for the "head" system

static **`reformatArray`**(*array*)

Reformats a Hilbert space array in-place to the all-zero state

Parameters **`array`** (*`np.array`*) – a numSystems x 2^systemSize x 2^systemSize 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

`squanch.qstream.`**`allZeroState`**(*systemSize*, *numSystems*)

Generate an array representing the numSystems Hilbert spaces in the state `|0>...|0><0|...<0|`

Parameters

- **`systemSize`** (*int*) – maximum size of entangled subsystems
- **`numSystems`** (*int*) – number of disjoint quantum subsystems to allocate

Returns the all-zero state array

4.8 `Qubit` – Qubits and quantum systems

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

Represents a multi-particle Hilbert space for several qubuts comprising a single system of a quantum datastream. Designed to have similar syntax to QubitSytem, but instantiation is much faster

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

Instatiate the quantum state for an n-qubit system

Parameters

- **`numQubits`** (*int*) – number of qubits in the system, treated as maximally entangled
- **`state`** (*np.array*) – density matrix representing the quantum state. By default, **|000...><...000|** is used

`__module__` = **'squanch.qubit'**

`apply`(*operator*)

Apply an n-qubit operator to this system's n-qubit quantum state: U|psi><psi|U^+ (U^+ = U for Hermitian)

Parameters **`operator`** (*np.array*) – the *Hermitian* n-qubit operator to apply

Returns nothing, the qSystem state is mutated

classmethod **`fromStream`**(*qStream*, *systemIndex*)

Instantiate a QSystem from a given point in a parent QStream

Parameters

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

Returns the QSystem object

`measureQubit`(*qubitIndex*)

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 **`qubitIndex`** (*int*) – the qubit to measure

Returns the measured qubit value

`qubit`(*qubitIndex*)

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

Returns the qubit instance

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

Represents a single physical qubit, which is a wrapper for part of a pre-allocated nonlocal QSystem

`__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

`__module__` = **'squanch.qubit'**

`apply`(*operator*, *cacheID=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 `fromStream`(*qStream*, *systemIndex*, *qubitIndex*)
Instantiate a qubit from a parent stream (via a QSystem call)

Parameters

- **qStream** (`QStream`) – the parent stream
- **systemIndex** (*int*) – the index corresponding to the parent QSystem
- **qubitIndex** (*int*) – the index of the qubit to be recalled

Returns the qubit

`getState`()
Traces over the remaining portions of the qSystem to return this qubit's state expressed as a density matrix.

Returns The (mixed) density matrix describing this qubit's state

`measure`()
Measure a qubit, mutating its state 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

S

INDEX