Python Implementation of Stabilizer Algorithms

Patrick Rall, Iskren Vankov - March 28, 2016

Progress over break

- Implement ExponentialSum
- More testing: **Code is definitely buggy!**

Questions

- What is wrong with the code below?
- **Unit tests: What tests can we perform to validate the implementations?**
- How to decompose $W(\mathcal{K}, q)$ into integers p, m, ϵ? Detailed in [15]?

Goals for next week

- *Patrick*: Implement the remaining routines: InnerProduct and MeasurePauli
- *Patrick*: Implement the main quantum circuit simulator
- *Patrick*: Debug code, implement unit tests
- *Iskren*: Identify C++ linalg libraries, review for threading support
- *Iskren*: Review, understand, and debug code below

Some random stabilizer states

```
States with n = 2

State 1:
00 (1+0j)
01 (1+0j)
10 (1+0j)
11 (1+0j)
Norm: 4.0

State 2:
00 (1+0j)
01 (1+0j)
10 (1+0j)
11 (1+0j)
Norm: 4.0

State 3:
00 (1+0j)
01 (1+0j)
10 (1+0j)
11 (1+0j)
Norm: 4.0

State 4:
00 (1+0j)
01 (1+0j)
10 (1+0j)
11 (1+0j)
Norm: 4.0

State 5:
00 (1+0j)
01 (1+0j)
10 (1+0j)
11 (1+0j)
Norm: 4.0
```

```
States with n = 3

State 1:
000 (1+0j)
001 (1+0j)
010 (1+0j)
011 (1+0j)
100 (1+0j)
101 (1+0j)
110 (1+0j)
111 (1+0j)
Norm: 8.0

State 2:
011 (0.5+0.5j)
111 (-0.5-0.5j)
Norm: 1.0

State 3:
010 (0.707+0j)
110 (-0-0.707j)
Norm: 1.0

State 4:
010 0.707j
110 (-0.707+0j)
Norm: 1.0

State 5:
010 (-0.5+0.5j)
110 (0.5+0.5j)
Norm: 1.0
```

```
States with n = 4

State 1:
0110 (-0.5+0.5j)
1110 (-0.5+0.5j)
Norm: 1.0

State 2:
0000 (0.354+0.354j)
0100 (-0.354+0.354j)
1000 (-0.354-0.354j)
1100 (0.354-0.354j)
Norm: 1.0

State 3:
0001 (0.354-0.354j)
0101 (0.354-0.354j)
1001 (0.354+0.354j)
1101 (-0.354-0.354j)
Norm: 1.0

State 4:
0000 (-0-0.5j)
0100 (-0-0.5j)
1000 (0.5+0j)
1100 (0.5+0j)
Norm: 1.0

State 5:
0001 (-0.354+0.354j)
0101 (0.354-0.354j)
1101 (0.354-0.354j)
Norm: 0.75
```

State Vector Extraction

```python
Evaluate q(x) for a string in K (page 10, equation 42)
def q(self, x):
    # If affine space has dimension zero then phase does not matter
    if (self.k == 0): return 0

    # x is a length n vector in basis of $\mathbb{F}_2^n$
    # vecx is a length k vector in basis of L(K)

    # B is n*k 'basis matrix' with each row a length n basis vector
    # Let vecx and x be row vectors. Then solve equation B vecx = x+h
    B = self.G[:self.k].T
    vecx = np.linalg.lstsq(B, x + self.h)[0].astype(int) % 2

    # check result: should succeed if x in K
    if not np.allclose(np.dot(B, vecx) % 2, (x + self.h) % 2):
        raise LookupError("Input vector is not the affine space.")

    # Evaluate equation 42
    qx = self.Q
    qx += np.inner(self.D, vecx)

    for a in range(self.k):
        for b in range(a):
            qx += self.J[a, b]*vecx[a]*vecx[b]

    return qx % 8
```

```python
# Coefficient for x in the superposition
def coeff(self, x):
    # compute coefficient according to page 10, equation 46
    try: return np.power(2, -0.5*self.k) * np.exp(self.q(x) * 1j * np.pi/4)
    except LookupError: return 0  # if vector is not in affine space
```

Helper Functions

```
# helper to update D, J using equations 48, 49 on page 10
def updateDJ(self, R):
    # equation 48
    self.D = np.dot(R, self.D)
    for b in range(self.k):
        for c in range(b):
            self.D += self.J[b, c]*R[b]*R[:, c]
    self.D = self.D % 8

    # equation 49
    self.J = np.dot(np.dot(R, self.J), R.T) % 8
```

```
# helper to update Q, D using equations 51, 52 on page 10
def updateQD(self, y):
    # equation 51
    self.Q += np.dot(self.D, y)
    for a in range(self.k):
        for b in range(a):
            self.Q += self.J[a, b]*y[a]*y[b]
    self.Q = self.Q % 8

    # equation 52
    self.D += np.dot(self.J, y)
    self.D = self.D % 8
```

ExponentialSum (Partial)

```python
def exponentialSum(self):
    S = [a for a in range(self.k) if self.D[a] in [2, 6]]
    if len(S) != 0:
        a = S[0]

        # Construct R as in comment on page 12
        R = np.identity(self.k)
        for b in S[1:]:
            R[b, a] += 1
        R = R % 2

        self.updateDJ(R)
        S = [a]
    # Now J[a, a] = 0 for all a not in S

    E = [k for k in range(self.k) if k not in S]
    M = []
    Dimers = []  # maintain list of dimers rather than r

    while len(E) > 0:
        a = E[0]
        K = [b for b in E[1:] if self.J[a, b] == 4]

        if len(K) == 0:  # found a new monomer {a}
            M.append(a)
            E = E[1:]
        else:
            b = K[0]

            # Construct R for basis change
            R = np.identity(self.k)
            for c in [x for x in E if x != a and x != b]:
                if self.J[a, c] == 4: R[c, a] += 1
                if self.J[b, c] == 4: R[c, b] += 1
            R = R % 2

            self.updateDJ(R)

            # {a, b} form a new dimer
            Dimers.append([a, b])
            E = [x for x in E if x != a and x != b]

    if len(S) != 0:
        # Compute W(K,q) from Eq. 63
        raise NotImplementedError  # Where exactly in reference 15?
    else:
        # Compute W_0, W_1 from Eq. 68
        raise NotImplementedError

# evaluates the expression in the comment on page 12
def W(p, m, eps):
    return eps * 2**(p/2) * np.exp(1j*np.pi*m/4)
```

Shrink

```python
# attempt to shrink the stabilizer state by eliminating a part
# of the basis that has inner product α with vector ξ
def shrink(self, xi, alpha, lazy=False):
    # S <- { a ∈ [k] : (ξ, g) = 1 }
    # Note that a is zero-indexed.
    S = [a for a in range(self.k) if np.inner(self.G[a], xi) % 2 == alpha]

    beta = (alpha + np.inner(xi, self.h)) % 2
    if len(S) == 0 and beta == 1: return "EMPTY"
    if len(S) == 0 and beta == 0: return "SAME"

    i = S[0]  # pick any i ∈ S
    S.remove(i)

    for a in S:
        # g^a ← g^a ⊕ g^i
        # compute shift matrix for G
        shift = np.concatenate((np.zeros((a, self.n)), [self.G[i]],
                                np.zeros((self.n - a - 1, self.n))))
        self.G = (self.G + shift) % 2

        # update D, J using equations 48, 49 on page 10
        # compute k*k basis change matrix R (equation 47)
        if not lazy:
            R = np.identity(self.k)
            R[a, i] = 1
            self.updateDJ(R)

        # ḡ^i ← ḡ^i + Σ_a ḡ^a
        self.Gbar[i] += self.Gbar[a]
    self.Gbar = self.Gbar % 2

    # swap g^i and g^k, ḡ^i and ḡ^k
    # remember elements are zero-indexed, so we use k-1
    self.G[[i, self.k-1]] = self.G[[self.k-1, i]]
    self.Gbar[[i, self.k-1]] = self.Gbar[[self.k-1, i]]

    # update D, J using equations 48, 49 on page 10
    if not lazy:
        R = np.identity(self.k)
        R[[i, self.k-1]] = R[[self.k-1, i]]
        self.updateDJ(R)

    # h ← h ⊕ β · g^k
    self.h = (self.h + beta*self.G[self.k-1]) % 2

    if not lazy:
        # update Q, D using equations 51, 52 on page 10
        y = np.zeros(self.k)
        y[self.k-1] = beta
        self.updateQD(y)

        self.J = self.J[1:, 1:]  # remove last row and column from J
        self.D = self.D[1:]      # remove last element from D

    self.k -= 1

    return "SUCCESS"
```

__init__ and RandomStabilizerState

```python
# Create an empty stabilizer state as used by
# the RandomStabilizerState function. It has
# K =\mathbb{F}^n_2 and has q(x) = 0 for all x.
def __init__(self, n, k):
    # define K from RandomStabilizerState algorithm (page 16)
    self.n = n
    self.k = k
    self.h = np.zeros(n)        # in $\mathbb{F}_2^n$
    self.G = np.identity(n)     # in $\mathbb{F}_2^{n\times n}$
    self.Gbar = np.identity(n)  # = $(G^{-1})^T$

    # define q to be zero for all x
    self.Q = 0                  # in $\mathbb{Z}_8$
    self.D = np.zeros(k)        # in $\{0,2,4,6\}^k$
    self.J = np.zeros((k, k))   # in $\{0,4\}^{k\times k}$, symmetric
```

```python
# cache probability distributions for stabilizer state dimension k
# in a dictionary, with a key for each n
dDists = {}

@classmethod
def randomStabilizerState(cls, n, provide_d=False):
    # ensure probability distribution is available for this n
    if n not in cls.dDists:
        # compute distribution given by equation 79 on page 15
        def eta(d):
            if d == 0: return 0

            product = 1
            for a in range(1, d+1):
                product *= (1 - 2**(d - n - a))
                product /= (1 - 2**(-a))
            return 2**(-d*(d+1)/2) * product

        # collect numerators
        dist = np.array([])
        for d in range(n):
            dist = np.append(dist, [eta(d)], 0)

        # normalize
        norm = sum(dist)
        dist /= norm

        # cache result
        cls.dDists[n] = dist

    # sample d from distribution
    sample = 1-np.random.random()  # sample from (0.0, 1.0]
    d = 0
    cumulative = 0
    while cumulative < sample:
        cumulative += cls.dDists[n][d]
        d += 1
    k = n - d
```

```python

    # pick random X in F_2^{d,n} with rank d
    while True:
        X = np.random.random_integers(0, 1, (d, n))

        if np.linalg.matrix_rank(X) == d: break

    # create the state object. __init__ gives the correct properties
    state = StabilizerState(n, k)

    for a in range(d):
        # lazy shrink with a'th row of X
        state.shrink(X[a], 0, lazy=True)

        # reset state's k after shrinking
        state.k = k

    # now K = ker(X) and is in standard form

    state.h = np.random.random_integers(0, 1, n)
    state.Q = np.random.random_integers(0, 7)
    state.D = 2*np.random.random_integers(0, 3, state.k)

    state.J = np.zeros((state.k, state.k))
    for a in range(state.k):
        state.J[a, a] = 2*state.D[a] % 8
        for b in range(a):
            state.J[a, b] = 4*np.random.random_integers(0, 1)
            state.J[b, a] = state.J[b, a]

    if not provide_d: return state
    else: return state, d
```