2.9.1 Addendum: Pauli Representation of Operators

In the section on Pauli operators we stated that Pauli matrices form the basis for 2×2 matrices and that Hermitian density operators can be written as the following, which is also called the *Pauli representation* of an operator:

$$\rho = \frac{I + xX + yY + zZ}{2}. \tag{2.1}$$

Let's see how to derive this result. First, let's note that since we claim that the Pauli matrices form an orthonormal basis for *any* 2×2 matrix, we should be able to write any such matrix as:

$$A = cI + xX + yY + zZ. \tag{2.2}$$

By simply adding up the four matrix terms we get:

$$A = \begin{bmatrix} c+z & x-iy \\ x+iy & c-z \end{bmatrix}.$$

Comparing Equation 2.1 and Equation 2.2 should immediately lead to three questions:

1. Why is there no factor c in front of the I in Equation 2.1?
2. Where does the factor $1/2$ come from?
3. Given a density matrix, what are its factors x, y, and z, and maybe c?

To answer the second and third questions first. For a given state $|\psi\rangle$ and it's density matrix $\rho = |\psi\rangle\langle\psi|$, we extract the individual factors by multiplying the density matrix with the corresponding Pauli matrix and taking the trace. Let's see how this works. To extract the factor x:

$$\begin{aligned} X|\psi\rangle\langle\psi| &= X \begin{bmatrix} c+z & x-iy \\ x+iy & c-z \end{bmatrix} \\ &= \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} c+z & x-iy \\ x+iy & c-z \end{bmatrix} \\ &= \begin{bmatrix} x+iy & c-z \\ c+z & x-iy \end{bmatrix}. \end{aligned}$$

Now, taking the trace of this matrix results in:

$$\mathrm{tr}(X|\psi\rangle\langle\psi|) = x + iy + x - iy = 2x.$$

We are able to extract the factor x, but with a factor of 2. This is the reason why in Equation 2.1 we compensate with a factor of $1/2$. Let's derive this for the other factors, starting with Y:

$$Y|\psi\rangle\langle\psi| = Y \begin{bmatrix} c+z & x-iy \\ x+iy & c-z \end{bmatrix}$$

$$
\begin{aligned}
&= \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \begin{bmatrix} c+z & x-iy \\ x+iy & c-z \end{bmatrix} \\
&= \begin{bmatrix} -i(x+iy) & -i(c-z) \\ i(c+z) & i(x-iy) \end{bmatrix} \\
&= \begin{bmatrix} -ix+y & -i(c-z) \\ i(c+z) & ix+y \end{bmatrix}.
\end{aligned}
$$

and

$$\mathrm{tr}(Y \left|\psi\right\rangle \left\langle\psi\right|) = -ix + y + ix + y = 2y.$$

And for Z:

$$
\begin{aligned}
Z \left|\psi\right\rangle \left\langle\psi\right| &= Z \begin{bmatrix} c+z & x-iy \\ x+iy & c-z \end{bmatrix} \\
&= \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} c+z & x-iy \\ x+iy & c-z \end{bmatrix} \\
&= \begin{bmatrix} c+z & x-iy \\ x+iy & z-c \end{bmatrix}.
\end{aligned}
$$

and

$$\mathrm{tr}(Z \left|\psi\right\rangle \left\langle\psi\right|) = c + z + z - c = 2z.$$

Finally, for the identity I, the right side remains unchanged:

$$
\begin{aligned}
I \left|\psi\right\rangle \left\langle\psi\right| &= I \begin{bmatrix} c+z & x-iy \\ x+iy & c-z \end{bmatrix} \\
&= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} c+z & x-iy \\ x+iy & c-z \end{bmatrix} \\
&= \begin{bmatrix} c+z & x-iy \\ x+iy & c-z \end{bmatrix}.
\end{aligned}
$$

Taking the trace results in

$$\mathrm{tr}(I \left|\psi\right\rangle \left\langle\psi\right|) = c + z + c - z = 2c.$$

Now we make use of the fact that the trace of the density matrix of a pure state must be 1. Since we already applied a factor $1/2$ in Equation 2.1 it follows that the factor c must be 1. This is why were able to omit this factor to I in Equation 2.1.

This is easy to verify in code. The full implementation is in the open-source repository in file `src/pauli_rep.py`. We construct a random qubit and extract the factors as described above:

```
qc = circuit.qc('random qubit')
qc.qubit(random.random())
qc.rx(0, math.pi * random.random())
```

```
qc.ry(0, math.pi * random.random())
qc.rz(0, math.pi * random.random())

[...]

rho = qc.psi.density()
i = np.trace(ops.Identity() @ rho)  # not strictly needed.
x = np.trace(ops.PauliX() @ rho)
y = np.trace(ops.PauliY() @ rho)
z = np.trace(ops.PauliZ() @ rho)
```

With these factors, we can verify that we indeed computed the correct results by simply plugging them into Equation 2.1:

```
new_rho = 0.5 * (i * ops.Identity() + x * ops.PauliX() +
                 y * ops.PauliY() + z * ops.PauliZ())
if not np.allclose(rho, new_rho):
  raise AssertionError('Invalid Pauli Representation')
```

Two (or more) qubits

We essentially computed the following for a single qubit, where σ_i are the Pauli matrices:

$$\rho = \frac{1}{2}\sum_{i=0}^{3} c_i \sigma_i.$$

This technique can be easily extended to two (or more qubits) by applying the same principles and multiplying the density matrix with *all* tensor products of two (or more) Pauli matrices. Similar to Equation 2.1, this assumes the density matrix can be constructed from this two-qubit basis in the following way:

$$\rho = \frac{1}{4}\sum_{i,j=0}^{3} c_{i,j} \left(\sigma_i \otimes \sigma_j\right).$$

Note that the factor is now $1/4$, or $1/2^n$ for n qubits. Also note that this form is related to the Schmidt decomposition, which we will discuss elsewhere.

To see how this work, let's write it in code. We first create a state and density matrix of two (or more) potentially entangled qubits:

```
qc = circuit.qc('random qubit')
qc.qubit(random.random())
qc.qubit(random.random())

# Potentially entangle them.
qc.h(0)
qc.cx(0, 1)
```

```
# Additionally rotate around randomly.
for i in range(2):
  qc.rx(i, math.pi * random.random())
  qc.ry(i, math.pi * random.random())
  qc.rz(i, math.pi * random.random())

# Compute density matrix.
rho = qc.psi.density()
```

Now multiply in all the Pauli matrix tensor products and compute the factors from the trace. Since two qubits are involved, instead of a vector of factors (c_i) we will obtain a matrix of factors $(c_{i,j})$, or higher-dimensional tensors, if more than two qubits are involved:

```
paulis = [ops.Identity(), ops.PauliX(), ops.PauliY(), ops.PauliZ()]
c = np.zeros((4, 4), dtype=np.complex64)
for i in range(4):
  for j in range(4):
    tprod = paulis[i] * paulis[j]
    c[i][j] = np.trace(rho @ tprod)
```

Similar to above, we can now construct a new state and verify that the computed factors were indeed the correct ones:

```
new_rho = np.zeros((4, 4), dtype=np.complex64)
for i in range(4):
  for j in range(4):
    tprod = paulis[i] * paulis[j]
    new_rho = new_rho + c[i][j] * tprod

if not np.allclose(rho, new_rho / 4, atol=1e-5):
  raise AssertionError('Invalid Pauli Representation')
```