COMS 4281 - Intro to Quantum Computing

Problem Set 5: Noise and Codes

Due: December 11, 11:59pm.

Collaboration is allowed and encouraged (teams of at most 3). Please read the syllabus carefully for the guidlines regarding collaboration. In particular, everyone must write their own solutions in their own words.

Name: Chandan Suri, UNI: CS4090

Write your collaborators here:

AS6413, ILD2105

Final 2 Late Days Consumed!

Problem 1: Mixed states

Problem 1.1

Compute the single- and two-qubit reduced density matrices of the 3-qubit GHZ state

$$\frac{|000\rangle + |111\rangle}{\sqrt{2}}.$$

Solution

Density matrix for the GHZ state can be written as:

$$|GHZ\rangle \langle GHZ| = \left[\frac{1}{\sqrt{2}}(|000\rangle + |111\rangle)\right]\left[\frac{1}{\sqrt{2}}(\langle 000| + \langle 111|)\right]$$

This can also be written as:

$$|GHZ\rangle \langle GHZ| = \frac{1}{2}(|000\rangle \langle 000| + |111\rangle \langle 000| + |000\rangle \langle 111| + |111\rangle \langle 111|)$$

As there are 3 qubits here, to find the single and two-qubit reduced density matrices, I will have to find the trace here. Treating first qubit from different system and 2nd and 3rd qubits of a different one, we can find the traces as follows:

Trace w.r.t the 1st qubit (giving two-qubit reduced density matrix) can be written as:

$$\rho_{23} = Tr_1(|GHZ\rangle \langle GHZ|) = |0\rangle_1 |GHZ\rangle \langle GHZ| \langle 0|_1 + |1\rangle_1 |GHZ\rangle \langle GHZ| \langle 1|_1$$

This can further be written down as:

$$\rho_{23} = Tr_1(|GHZ\rangle \langle GHZ|)$$

$$= \frac{1}{2}\Big[(|\langle 0|0\rangle|^2 |00\rangle \langle 00| + \langle 0|1\rangle\langle 0|0\rangle |11\rangle \langle 00| + \langle 0|0\rangle\langle 1|0\rangle |00\rangle \langle 11| + \langle 0|1\rangle\langle 1|0\rangle |11\rangle \langle 11|) + (\langle 1|0\rangle\langle 0|1\rangle |00\rangle \langle 00| + \langle 1|1\rangle\langle 0|1\rangle |11\rangle \langle 00| + \langle 1|0\rangle\langle 1|1\rangle$$

The values for the parts in the above equation where one part of ket-bra is 0 and another is 1, this gives us 0 which when multiplied to corresponding terms of a term will give out zero thus, making most of the terms in the above equation to be zero as follows:

$$\rho_{23} = Tr_1(|GHZ\rangle \langle GHZ|) = \frac{1}{2}\Big[(|\langle 0|0\rangle|^2 |00\rangle \langle 00| + 0 + 0 + 0) + (0 + 0 + 0 + |\langle 1|1\rangle|^2 |11\rangle \langle 11|)\Big]$$

This gives us the following:

$$\rho_{23} = Tr_1(|GHZ\rangle \langle GHZ|) = \frac{1}{2}(|\langle 0|0\rangle|^2 |00\rangle \langle 00| + |\langle 1|1\rangle|^2 |11\rangle \langle 11|)$$

The values like $|\langle 0|0\rangle|^2$ and $|\langle 1|1\rangle|^2$ will become 1, thus, giving us the following:

$$\rho_{23} = Tr_1(|GHZ\rangle \langle GHZ|) = \frac{1}{2}(|00\rangle \langle 00| + |11\rangle \langle 11|)$$

Next, Trace w.r.t the 2nd and 3rd qubit (giving one-qubit reduced density matrix) can be written as:

$$\rho_1 = Tr_{23}(|GHZ\rangle \langle GHZ|) = |00\rangle_{23} |GHZ\rangle \langle GHZ| \langle 00|_{23} + |11\rangle_{23} |GHZ\rangle \langle GHZ| \langle 11|_{23}$$

This can be further written down as:

$$\rho_1 = Tr_{23}(|GHZ\rangle \langle GHZ|)$$

$$= \frac{1}{2}\Big[(|\langle 00|00\rangle|^2 |0\rangle \langle 0| + \langle 00|11\rangle\langle 00|00\rangle |1\rangle \langle 0| + \langle 00|00\rangle\langle 11|00\rangle |0\rangle \langle 1| + \langle 00|11\rangle\langle 11|00\rangle |1\rangle \langle 1|) + (\langle 11|00\rangle\langle 00|11\rangle |0\rangle \langle 1| + \langle 11|11\rangle\langle 00|11\rangle |1\rangle \langle 0| \cdot$$

In the above equation, the values like $\langle 00|11\rangle$ and $\langle 11|00\rangle$ are actually zeros thus making the corresponding terms zero as well:

$$\rho_1 = Tr_{23}(|GHZ\rangle \langle GHZ|) = \frac{1}{2}\Big[(|\langle 00|00\rangle|^2 |0\rangle \langle 0| + 0 + 0 + 0) + (0 + 0 + 0 + |\langle 11|11\rangle|^2 |1\rangle \langle 1|)\Big]$$

Also, the values like $|\langle 00|00\rangle|^2$ and $|\langle 11|11\rangle|^2$ will become 1, thus, giving us the following:

$$\rho_1 = Tr_{23}(|GHZ\rangle \langle GHZ|) = \frac{1}{2}[|0\rangle \langle 0| + |1\rangle \langle 1|]$$

So, now that we have both the traces, essentially, whole density matrix can be written as a combination of these 2 reduced density matrices as:

$$= \rho_1 \otimes \rho_{23}$$

This, in turn can be written as:

$$= \frac{1}{4}(|0\rangle \langle 0| + |1\rangle \langle 1|) \otimes (|00\rangle \langle 00| + |11\rangle \langle 11|)$$

Problem 1.2

Let $\{(p_1, |\psi_1\rangle), \ldots, (\psi_k, |\psi_k\rangle)\}$ denote a probabilistic mixture of states -- not necessarily orthogonal -- where p_i's are probabilities that sum to 1. Consider the density matrix $\sigma = p_1 |\psi_1\rangle \langle \psi_1| + \cdots + p_k |\psi_k\rangle \langle \psi_k|$ on a system A. Show that the following vector $|\theta\rangle_{AB}$ is a valid pure state (i.e. unit length vector), and it purifies σ:

$$|\theta\rangle = \sum_{i=1}^{k} \sqrt{p_i} |\psi_i\rangle_A \otimes |i\rangle_B$$

where B is a register of dimension at least k and $\{|i\rangle\}$ is an orthonormal basis for B.

Solution

We know that if the vector $|\theta\rangle_{AB}$ is a valid pure state, then the following should be proven true:

$$\sigma_A = Tr_B(|\theta\rangle_{AB} \langle \theta|_{AB})$$

Let's try and prove this then!

We put the formulation for $|\theta\rangle_{AB}$ in the above formulation as follows:

$$\sigma_A = Tr_B\left(\sum_{i=1}^{k} \sqrt{p_i} |\psi_i\rangle_A \otimes |i\rangle_B \cdot \sum_{i=1}^{k} \sqrt{p_i} \langle i|_B \otimes \langle \psi_i|_A\right)$$

As, $\{|i\rangle\}$ is an orthonormal basis for B, we know that the vectors in the basis when multiplied with any vector in the basis but itself results in a zero, so, we won't take those terms into consideration and thus, we can join the two summations as one as follows:

$$\sigma_A = Tr_B\left(\sum_{i=1}^{k} p_i (|\psi_i\rangle_A \otimes |i\rangle_B) \cdot (\langle i|_B \otimes \langle \psi_i|_A)\right)$$

As this is a trace for the system B, we can take out the terms associated with system A outside of the Trace operation giving us the following:

$$\sigma_A = \sum_{i=1}^{k} p_i |\psi_i\rangle_A \langle \psi_i|_A \left[Tr_B\left(\sum_{i=1}^{k} |i\rangle_B \langle i|_B\right)\right]$$

Since, trace is linear, we can move the trace operator inside the summation as follows:

$$\sigma_A = \sum_{i=1}^{k} p_i |\psi_i\rangle_A \langle \psi_i|_A \left[\sum_{i=1}^{k} Tr_B(|i\rangle_B \langle i|_B)\right]$$

Using the property of trace where $Tr(|\psi\rangle \langle \psi|) = \langle \psi|\psi\rangle$, we can write the above formula as:

$$\sigma_A = \sum_{i=1}^{k} p_i |\psi_i\rangle_A \langle \psi_i|_A \left[\sum_{i=1}^{k} \langle i|i\rangle\right]$$

Next, since $\{|i\rangle\}$ is an orthonormal basis for B, multiplying the vectors in the basis with itself and adding them up will give us 1 over all the dimensions. Thus, the formulation becomes as follows:

$$\sigma_A = \sum_{i=1}^{k} p_i |\psi_i\rangle_A \langle\psi_i|_A \, [1]$$

Thus, this becomes:

$$\sigma_A = \sum_{i=1}^{k} p_i |\psi_i\rangle_A \langle\psi_i|_A$$

This is basically the same as LHS as:

$$\sigma_A = p_1 |\psi_1\rangle_A \langle\psi_1|_A + p_2 |\psi_2\rangle_A \langle\psi_2|_A + \ldots + p_k |\psi_k\rangle_A \langle\psi_k|_A$$

Hence, we proved that the vector $|\theta\rangle_{AB}$ is a valid pure state (i.e. unit length vector), and it purifies σ.

Problem 1.3

Come up with a two-qubit pure state $|\psi\rangle_{AB}$ whose reduced density matrix on system A is

$$\mathrm{Tr}_B(|\psi\rangle\langle\psi|) = \frac{1}{2}\begin{pmatrix} 1+p & -\sqrt{(1-p)p} \\ -\sqrt{(1-p)p} & 1-p \end{pmatrix}$$

where $0 \leq p \leq 1$ is some parameter.

Hint: Try to reduce to part (2) above.

Solution

For the case of two qubits, following the proof of problem 1.2, we can write the above trace matrix (on system A) to be:

$$= p_1 |\psi_1\rangle_A \langle\psi_1|_A + p_2 |\psi_2\rangle_A \langle\psi_2|_A$$

Let's try and break the $\mathrm{Tr}_B(|\psi\rangle\langle\psi|)$ above as:

$$= \frac{1}{2}\left[\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} + \begin{pmatrix} p & -\sqrt{(1-p)p} \\ -\sqrt{(1-p)p} & 1-p \end{pmatrix}\right]$$

The first matrix above can be written as follows:

$$\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} 1 & 0 \end{pmatrix}$$

Furthermore, the second matrix can be written as follows:

$$\begin{pmatrix} p & -\sqrt{(1-p)p} \\ -\sqrt{(1-p)p} & 1-p \end{pmatrix} = \begin{pmatrix} -\sqrt{p} \\ \sqrt{1-p} \end{pmatrix} \begin{pmatrix} -\sqrt{p} & \sqrt{1-p} \end{pmatrix}$$

Therefore, the equation can be written as:

$$\mathrm{Tr}_B(|\psi\rangle\langle\psi|) = \frac{1}{2}\begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} 1 & 0 \end{pmatrix} + \frac{1}{2}\begin{pmatrix} -\sqrt{p} \\ \sqrt{1-p} \end{pmatrix} \begin{pmatrix} -\sqrt{p} & \sqrt{1-p} \end{pmatrix}$$

Thus, comparing it with the equation from problem 1.2 gives us:

$$p_1 = p_2 = \frac{1}{2}$$

and,

$$|\psi_1\rangle_A = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

and,

$$|\psi_2\rangle_A = \begin{pmatrix} -\sqrt{p} \\ \sqrt{1-p} \end{pmatrix}$$

Following from problem 1.2, we can compute the two-qubit pure state as:

$$|\psi\rangle_A B = \sum_{i=1}^{2} \sqrt{p_i} |\psi_i\rangle_A \otimes |i\rangle_B$$

Putting in values from above computed before:

$$|\psi\rangle_{AB} = \sqrt{p_1}|\psi_1\rangle_A \otimes |1\rangle_B + \sqrt{p_2}|\psi_2\rangle_A \otimes |2\rangle_B = \sqrt{\frac{1}{2}}\left(\begin{pmatrix}1\\0\end{pmatrix} \otimes |1\rangle_B + \begin{pmatrix}-\sqrt{p}\\\sqrt{1-p}\end{pmatrix} \otimes |2\rangle_B\right)$$

Putting two orthonormal vectors as basis vectors:

$$|\psi\rangle_{AB} = \sqrt{\frac{1}{2}}\left(\begin{pmatrix}1\\0\end{pmatrix} \otimes \begin{pmatrix}1\\0\end{pmatrix} + \begin{pmatrix}-\sqrt{p}\\\sqrt{1-p}\end{pmatrix} \otimes \begin{pmatrix}0\\1\end{pmatrix}\right)$$

Solving the above equation gives us:

$$|\psi\rangle_{AB} = \sqrt{\frac{1}{2}}\left(\begin{pmatrix}1&0\\0&0\end{pmatrix} + \begin{pmatrix}0&-\sqrt{p}\\0&\sqrt{1-p}\end{pmatrix}\right) = \sqrt{\frac{1}{2}}\begin{pmatrix}1&-\sqrt{p}\\0&\sqrt{1-p}\end{pmatrix}$$

This is the two-qubit pure state $|\psi\rangle_{AB} = \sqrt{\frac{1}{2}}\begin{pmatrix}1&-\sqrt{p}\\0&\sqrt{1-p}\end{pmatrix}$ as computed above.

The above pure state can also be written as:

$$|\psi\rangle_{AB} == \sqrt{\frac{1}{2}}\Big(|0\rangle|0\rangle + \Big(-\sqrt{p}|0\rangle + \sqrt{1-p}|1\rangle\Big)|1\rangle\Big)$$

Problem 1.4

Let $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ denote an unknown qubit, and suppose it was just teleported from Alice to Bob. But imagine that, before Alice is able to tell Bob the outcomes of her measurements, her internet cuts out. So Bob's qubit is one of the states

$$|\psi\rangle, X|\psi\rangle, Z|\psi\rangle, XZ|\psi\rangle$$

with equal probability. What is the density matrix that describes the mixed state of Bob's qubit?

Solution

As $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, therefore:

$$X|\psi\rangle = \beta|0\rangle + \alpha|1\rangle,$$
$$Z|\psi\rangle = \alpha|0\rangle - \beta|1\rangle,$$
$$XZ|\psi\rangle = -\beta|0\rangle + \alpha|1\rangle$$

The density matrix for these can be written as (considering that the probabilities for each of these is equal and equal to $\frac{1}{4}$):

$$\rho = \frac{1}{4}((\alpha|0\rangle + \beta|1\rangle)(\alpha\langle 0| + \beta\langle 1|) + (\beta|0\rangle + \alpha|1\rangle)(\beta\langle 0| + \alpha\langle 1|) + (\alpha|0\rangle - \beta|1\rangle)(\alpha\langle 0| - \beta\langle 1|) + (-\beta|0\rangle + \alpha|1\rangle)(-\beta\langle 0| + \alpha\langle 1|))$$

When opened up as matrices, this looks like:

$$\rho = \frac{1}{4}\left(\begin{pmatrix}\alpha\\\beta\end{pmatrix}\begin{pmatrix}\alpha & \beta\end{pmatrix} + \begin{pmatrix}\beta\\\alpha\end{pmatrix}\begin{pmatrix}\beta & \alpha\end{pmatrix} + \begin{pmatrix}\alpha\\-\beta\end{pmatrix}\begin{pmatrix}\alpha & -\beta\end{pmatrix} + \begin{pmatrix}-\beta\\\alpha\end{pmatrix}\begin{pmatrix}-\beta & \alpha\end{pmatrix}\right)$$

Solving it gives the following:

$$\rho = \frac{1}{4}\left(\begin{pmatrix}\alpha^2 & \alpha\beta\\\alpha\beta & \beta^2\end{pmatrix} + \begin{pmatrix}\beta^2 & \alpha\beta\\\alpha\beta & \alpha^2\end{pmatrix} + \begin{pmatrix}\alpha^2 & -\alpha\beta\\-\alpha\beta & \beta^2\end{pmatrix} + \begin{pmatrix}\beta^2 & -\alpha\beta\\-\alpha\beta & \alpha^2\end{pmatrix}\right)$$

Solving it further gives us:

$$\rho = \frac{1}{4}\begin{pmatrix}2(\alpha^2+\beta^2) & 0\\0 & 2(\alpha^2+\beta^2)\end{pmatrix} = \frac{1}{2}\begin{pmatrix}\alpha^2+\beta^2 & 0\\0 & \alpha^2+\beta^2\end{pmatrix}$$

This is the final density matrix above.

Problem 2: Limitations of Shor's code

Show that there are distinct two-qubit errors $E_1 \neq E_2$ (unitaries acting on two qubits) such that

$$E_1\left|\bar{0}\right\rangle = E_2\left|\bar{1}\right\rangle$$

where $\left|\bar{0}\right\rangle, \left|\bar{1}\right\rangle$ represent the logical $|0\rangle, |1\rangle$ state using the 9-qubit Shor code. This shows that the Shor code is not capable of correcting more than 1 error.

Solution

Initially the Shor's code for $\left|\bar{0}\right\rangle$ and $\left|\bar{1}\right\rangle$ can be written as:

$$\left|\bar{0}\right\rangle = \frac{1}{\sqrt{8}}\Big(|000000000\rangle + |000111000\rangle + |111000000\rangle + |111111000\rangle + |000000111\rangle + |000111111\rangle + |111000111\rangle + |111111111\rangle \Big)$$
$$\left|\bar{1}\right\rangle = \frac{1}{\sqrt{8}}\Big(|000000000\rangle - |000111000\rangle - |111000000\rangle + |111111000\rangle - |000000111\rangle + |000111111\rangle + |111000111\rangle - |111111111\rangle \Big)$$

Let's take the following distinct two-qubit errors:

$$E_1 = X_1 Z_7$$
$$E_2 = X_1 Z_5 Z_1$$

First, let's compute $E_1 \left|\bar{0}\right\rangle = X_1 Z_7 \left|\bar{0}\right\rangle$:

$$E_1 \left|\bar{0}\right\rangle = X_1 Z_7 \left|\bar{0}\right\rangle = X_1\Big(\frac{1}{\sqrt{8}}\Big(|000000000\rangle + |000111000\rangle + |111000000\rangle + |111111000\rangle - |000000111\rangle - |000111111\rangle - |111000111\rangle - |1111$$

Solving further:

$$E_1 \left|\bar{0}\right\rangle = X_1 Z_7 \left|\bar{0}\right\rangle = \frac{1}{\sqrt{8}}\Big(|100000000\rangle + |100111000\rangle + |011000000\rangle + |011111000\rangle - |100000111\rangle - |100111111\rangle - |011000111\rangle - |01111111$$

Second, let's compute $E_2 \left|\bar{1}\right\rangle = X_1 Z_5 Z_1 \left|\bar{1}\right\rangle$:

$$E_2 \left|\bar{1}\right\rangle = X_1 Z_5 Z_1 \left|\bar{1}\right\rangle = X_1 Z_5\Big(\frac{1}{\sqrt{8}}\Big(|000000000\rangle - |000111000\rangle + |111000000\rangle - |111111000\rangle - |000000111\rangle + |000111111\rangle - |111000111\rangle + |$$

Solving further:

$$E_2 \left|\bar{1}\right\rangle = X_1 Z_5 Z_1 \left|\bar{1}\right\rangle = X_1\Big(\frac{1}{\sqrt{8}}\Big(|000000000\rangle + |000111000\rangle + |111000000\rangle + |111111000\rangle - |000000111\rangle - |000111111\rangle - |111000111\rangle - |11$$

Solving even further:

$$E_2 \left|\bar{1}\right\rangle = X_1 Z_5 Z_1 \left|\bar{1}\right\rangle = \frac{1}{\sqrt{8}}\Big(|100000000\rangle + |100111000\rangle + |011000000\rangle + |011111000\rangle - |100000111\rangle - |100111111\rangle - |011000111\rangle - |011111$$

As we can see above, using two distinct two-qubit errors gives us the same result such that:

$$E_1 \left|\bar{0}\right\rangle = E_2 \left|\bar{1}\right\rangle$$

As such a pair exists, thus, this shows that the Shor code is not capable of correcting more than 1 error.

Problem 3: An error *detection* code

Shor's 9-qubit code can **correct** any single-qubit error on one of its 9 qubits. Below we give a 4-qubit code which can \emph{detect} any single-qubit error on its qubits. By this we mean that there is a detection circuit that determines if a single-qubit error has occurred, but it can't necessarily identify which one has occurred.

The encoding map is as follows:

$$\left|\bar{0}\right\rangle = \frac{1}{2}\Big(|00\rangle + |11\rangle \Big) \otimes \Big(|00\rangle + |11\rangle \Big)$$
$$\left|\bar{1}\right\rangle = \frac{1}{2}\Big(|00\rangle - |11\rangle \Big) \otimes \Big(|00\rangle - |11\rangle \Big) .$$

Problem 3.1

Give an example of two distinct single-qubit unitaries $E_1 \neq E_2$ (which may act on different qubits) such that

$$E_1 \left|\bar{0}\right\rangle = E_2 \left|\bar{1}\right\rangle \ .$$

This shows that this code cannot uniquely identify single-qubit errors, because given $E_1 \left|\bar{0}\right\rangle$, one can't be sure whether the original state was $\left|\bar{0}\right\rangle$ or $\left|\bar{1}\right\rangle$. However, in the next few parts you will show that the code can still detect that **some** error has occurred.

Solution

The equations are specified as:

$$\left|\bar{0}\right\rangle = \frac{1}{2}\Big(\left|00\right\rangle + \left|11\right\rangle\Big) \otimes \Big(\left|00\right\rangle + \left|11\right\rangle\Big)$$
$$\left|\bar{1}\right\rangle = \frac{1}{2}\Big(\left|00\right\rangle - \left|11\right\rangle\Big) \otimes \Big(\left|00\right\rangle - \left|11\right\rangle\Big).$$

If we take $E_1 = Z_1$ and $E_2 = Z_3$, that means that E_1 is the single qubit unitary for applying Z gate on 1^{st} qubit and E_2 is the single qubit unitary for applying Z gate on 3^{rd} qubit as follows:

$$E_1\left|\bar{0}\right\rangle = Z_1\left|\bar{0}\right\rangle = \frac{1}{2}\Big(\left|00\right\rangle - \left|11\right\rangle\Big) \otimes \Big(\left|00\right\rangle + \left|11\right\rangle\Big)$$
$$E_2\left|\bar{1}\right\rangle = Z_3\left|\bar{1}\right\rangle = \frac{1}{2}\Big(\left|00\right\rangle - \left|11\right\rangle\Big) \otimes \Big(\left|00\right\rangle + \left|11\right\rangle\Big).$$

As we can see above, applying $E_1 = Z_1$ and $E_2 = Z_3$ above gives us $E_1\left|\bar{0}\right\rangle = E_2\left|\bar{1}\right\rangle$.

As those were two different single qubit unitaries acting on different qubits resulting in the same state, these two distinct single qubit unitaries show that this code cannot uniquely identify single-qubit errors, because given $E_1\left|\bar{0}\right\rangle$, one can't be sure whether the original state was $\left|\bar{0}\right\rangle$ or $\left|\bar{1}\right\rangle$.

Problem 3.2

Give a procedure (either as a quantum circuit or sufficiently detailed pseudocode) that detects a \emph{bitflip} error on a single qubit of an encoded state $\alpha\left|\bar{0}\right\rangle + \beta\left|\bar{1}\right\rangle$. In other words the procedure, if it's given a valid encoded state $\left|\bar{\psi}\right\rangle$, returns $\left|\bar{\psi}\right\rangle$ and outputs "No error", whereas if it's given $X_i\left|\bar{\psi}\right\rangle$ where X_i is the bitflip operation on some qubit i, then the circuit outputs "X error somewhere".

Solution

The encoded state can be written as follows:

$$\left|\bar{\psi}\right\rangle = \alpha\left|\bar{0}\right\rangle + \beta\left|\bar{1}\right\rangle = \frac{\alpha}{2}\Big(\left|00\right\rangle + \left|11\right\rangle\Big) \otimes \Big(\left|00\right\rangle + \left|11\right\rangle\Big) + \frac{\beta}{2}\Big(\left|00\right\rangle - \left|11\right\rangle\Big) \otimes \Big(\left|00\right\rangle - \left|11\right\rangle\Big)$$

Solving it further gives us the following:

$$\left|\bar{\psi}\right\rangle = \alpha\left|\bar{0}\right\rangle + \beta\left|\bar{1}\right\rangle = \frac{\alpha}{2}\Big(\left|0000\right\rangle + \left|1100\right\rangle + \left|0011\right\rangle + \left|1111\right\rangle\Big) + \frac{\beta}{2}\Big(\left|0000\right\rangle - \left|0011\right\rangle - \left|1100\right\rangle + \left|1111\right\rangle\Big)$$

Now, let's say there was a bit flip error on the 2nd qubit $X_2\left|\bar{\psi}\right\rangle$:

$$X_2\left|\bar{\psi}\right\rangle = \frac{\alpha}{2}\Big(\left|0100\right\rangle + \left|1000\right\rangle + \left|0111\right\rangle + \left|1011\right\rangle\Big) + \frac{\beta}{2}\Big(\left|0100\right\rangle - \left|0111\right\rangle - \left|1000\right\rangle + \left|1011\right\rangle\Big)$$

If we use one syndrome qubits (as explained below), the complete state of the system can be defined as:

$$\frac{\alpha}{2}\Big(\left|0100\right\rangle + \left|1000\right\rangle + \left|0111\right\rangle + \left|1011\right\rangle\Big)\left|0\right\rangle + \frac{\beta}{2}\Big(\left|0100\right\rangle - \left|0111\right\rangle - \left|1000\right\rangle + \left|1011\right\rangle\Big)\left|0\right\rangle$$

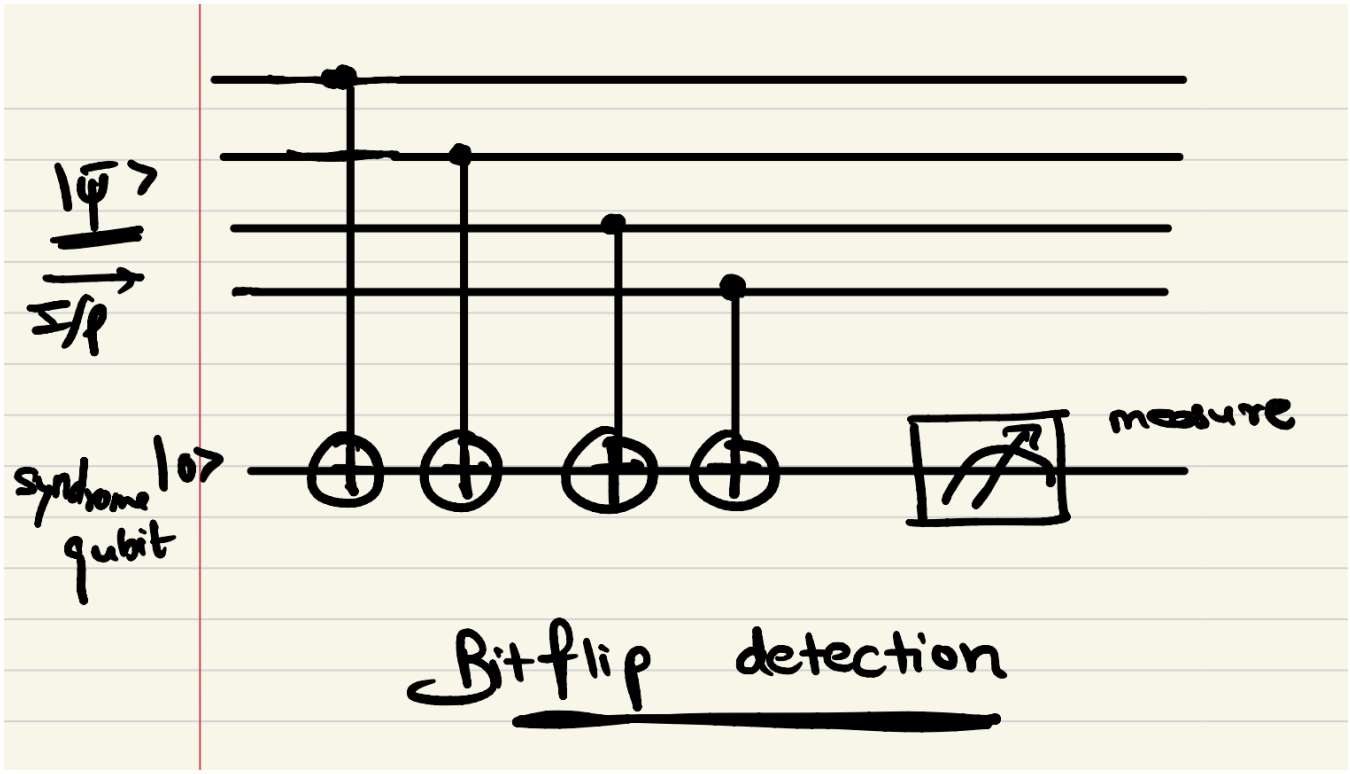
Coming to the procedure we could use, we can do the following:

1. We will take 1 syndrome qubit here that we will be measuring at the end to see if any error might have occured.
2. We will put a CNOT gate between the 1st qubit and the syndrome qubit such that the control qubit is the 1st qubit of the encoded state and target qubit is the syndrome qubit. Also, we put another CNOT after that between the 2nd qubit of the encoded state and the syndrome qubit such that the control qubit is from the encoded state and the target qubit is the syndrome qubit.
3. In a similar fashion as above, we put another CNOT gate between the 3rd qubit of the encoded state and the syndrome qubit, and between the 4th qubit of the encoded state and the syndrome qubit (the ones from the encoded state are control qubits and the syndrome qubit is the target qubit).
4. Then, at the end we perform the measurements on the syndrome qubit. If, the syndrome qubit is in $\left|0\right\rangle$ state then we can say with certainity that there is no bit flip error. But if the syndrome qubit is in $\left|1\right\rangle$ state, then we can say that there is some error although it doesn't certainly tell us the which qubit there is an error in. As, we are creating a bit flip error detection circuit, using this procedure will work.

The circuit for the procedure is shown below:

In [1]:
```
from IPython.display import Image
Image("BitFlipErrorDetectionCircuit.PNG")
```

Out[1]:



Let's prove that it works!

If we had performed X_2 operation, then after 1^{st} CNOT, the complete state of the system will be:

$$\frac{\alpha}{2}\Big(|0100\rangle|0\rangle + |1000\rangle|1\rangle + |0111\rangle|0\rangle + |1011\rangle|1\rangle\Big) + \frac{\beta}{2}\Big(|0100\rangle|0\rangle - |0111\rangle|0\rangle - |1000\rangle|1\rangle + |1011\rangle|1\rangle\Big)$$

Next, after 2^{nd} CNOT, the state will be:

$$\frac{\alpha}{2}\Big(|0100\rangle + |1000\rangle + |0111\rangle + |1011\rangle\Big)|1\rangle + \frac{\beta}{2}\Big(|0100\rangle - |0111\rangle - |1000\rangle + |1011\rangle\Big)|1\rangle$$

Next, after 3^{rd} CNOT, the state will be:

$$\frac{\alpha}{2}\Big(|0100\rangle|1\rangle + |1000\rangle|1\rangle + |0111\rangle|0\rangle + |1011\rangle|0\rangle\Big) + \frac{\beta}{2}\Big(|0100\rangle|1\rangle - |0111\rangle|0\rangle - |1000\rangle|1\rangle + |1011\rangle|0\rangle\Big)$$

Next, after 4^{th} CNOT, the state will be:

$$\frac{\alpha}{2}\Big(|0100\rangle + |1000\rangle + |0111\rangle + |1011\rangle\Big)|1\rangle + \frac{\beta}{2}\Big(|0100\rangle - |0111\rangle - |1000\rangle + |1011\rangle\Big)|1\rangle$$

As we can see above, we are getting $|1\rangle$ state at the end because there was a bit flip error. If we look very closely, we can see that if there is any single bit flip error, then there will be unequal number of 0's and 1's because of which the syndrome qubit will end up as $|1\rangle$. Precisely, if there is a bitflip error on any one of the qubits, then the $\left|\bar{\psi}\right\rangle$ will contains odd number of qubit in state 1, thus, applying CNOT odd number of times on the syndrome qubit will result in state $|1\rangle$. Thus, this procedure when gives $|0\rangle$ as an output after measurement then that means that "No error". Otherwise, we can say that "X error somewhere".

Problem 3.3

Give a procedure (either as a quantum circuit or sufficiently detailed pseudocode) that detects a **phaseflip** error on a single qubit of an encoded state $\alpha\left|\bar{0}\right\rangle + \beta\left|\bar{1}\right\rangle$.

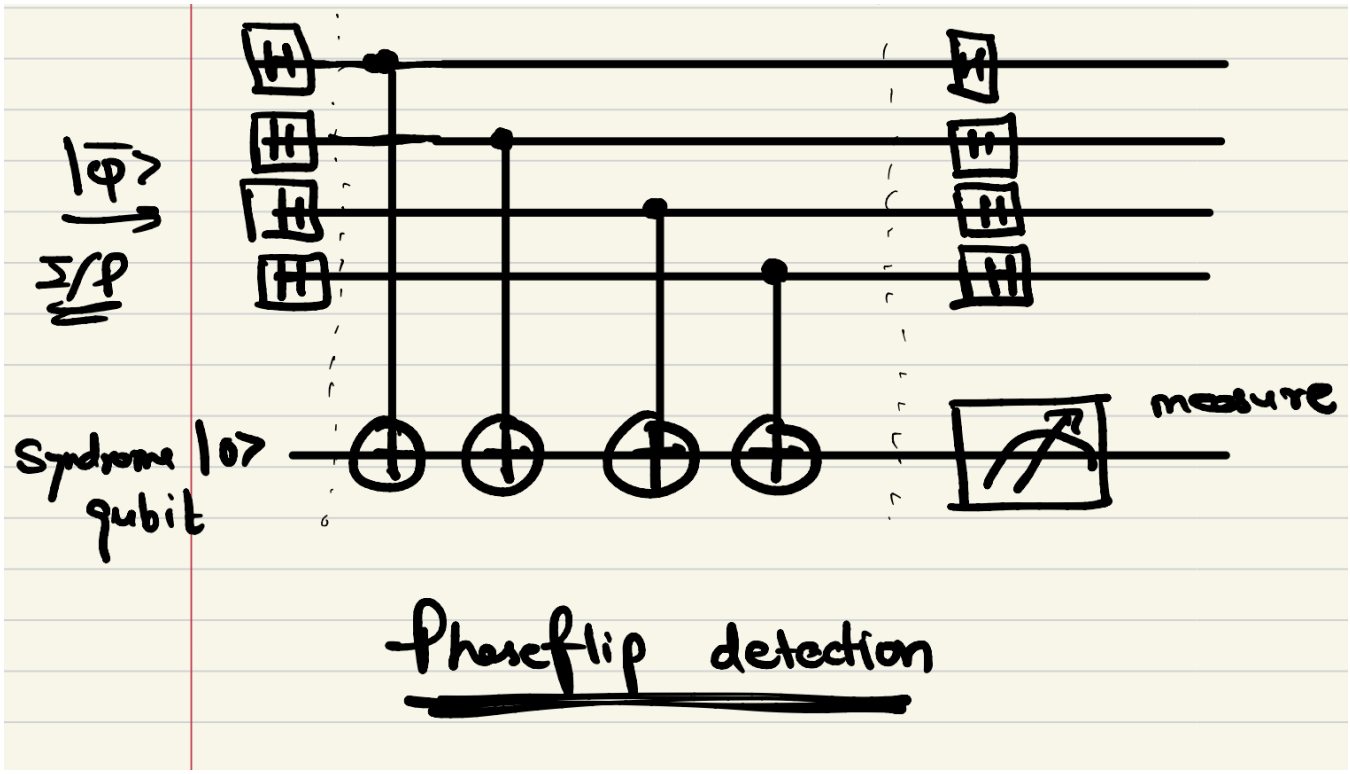
Solution

As we studied in class, a phaseflip error is basically a bit flip error such that the H gate (Hadamard) has been applied to all the qubits before passing in into the bit flip detection circuit. As we proved above that the procedure explained works for bit flip errors, we only need to do one additional thing here.

We would just need to add Hadamard gates on all the qubits as shown in the diagram below and add the bit flip detection circuit for detecting a phaseflip. This works as the H gates convert all the qubits to $|+\rangle$ or $|-\rangle$ states. When there is a phaseflip on any of the qubits, then that would change the state from $|+\rangle$ to $|-\rangle$ and vice-versa. Owing to this fact, the same bit flip detection circuit can be used where the syndrome qubit will only give the output as $|1\rangle$ state when there is a single qubit phaseflip error.

In [2]:
```
from IPython.display import Image
Image("PhaseFlipErrorDetectionCircuit.PNG")
```

Out[2]:



Thus, when the syndrome qubit gives a $|0\rangle$ state then there was "no error". But, when it is measured as $|1\rangle$ state, there was some "Z error somewhere". Although this procedure won't tell us the exact phaseflip error location (which qubit), but will be able to tell us with certainity, whether there is a phaseflip error or not.

Problem 3.4

Explain how to detect **any** unitary 1-qubit error on one of the 4 qubits.

Solution

To detect any unitary 1-qubit error one any one of the 4 qubits, we know there can be only one of the four following cases (considering C as the circuit explained below, s_1 syndrome qubit needed for bitflip error and s_2 for phaseflip error):

1. Only Bitflip error: $CX_i \left|\overline{\psi}\right\rangle |00\rangle = X_i \left|\overline{\psi}\right\rangle |10\rangle$
2. Only Phaseflip error: $CZ_i \left|\overline{\psi}\right\rangle |00\rangle = Z_i \left|\overline{\psi}\right\rangle |01\rangle$
3. Both Bitflip and Phaseflip error: $CY_i \left|\overline{\psi}\right\rangle |00\rangle = Y_i \left|\overline{\psi}\right\rangle |11\rangle$
4. No error: $CI \left|\overline{\psi}\right\rangle |00\rangle = \left|\overline{\psi}\right\rangle |00\rangle$

For the 1st case above, we can use the procedure or circuit shown and explained in problem 3.2 solution.

For the 2nd case above, we can use the procedure or circuit shown and explained in problem 3.3 solution.

For the 3rd case above, we can combine both the procedures for bit flip detection and phaseflip detection one after the other and use separate syndrome qubits for each of them (1 for phaseflip detection and 1 for bitflip detection).

For the 4th case, whatever we do there is no error to detect!

Thus,combining both the procedures one after the other would detect any unitary 1-qubit error on any one of the 4 qubits successfully!

Moreover, any unitary M can be written as a linear combination of the Pauli matrices as $M = aI + bX + cZ + dY$. Then, applying the circuit C to $M \left|\overline{\psi}\right\rangle$, we would get the following:

$$CM \left|\overline{\psi}\right\rangle |00\rangle = a \times CI \left|\overline{\psi}\right\rangle |00\rangle + b \times CX_i \left|\overline{\psi}\right\rangle |00\rangle + c \times CZ_i \left|\overline{\psi}\right\rangle |00\rangle + d \times CY_i \left|\overline{\psi}\right\rangle |00\rangle$$

Using the formulation above, we can also write this as:

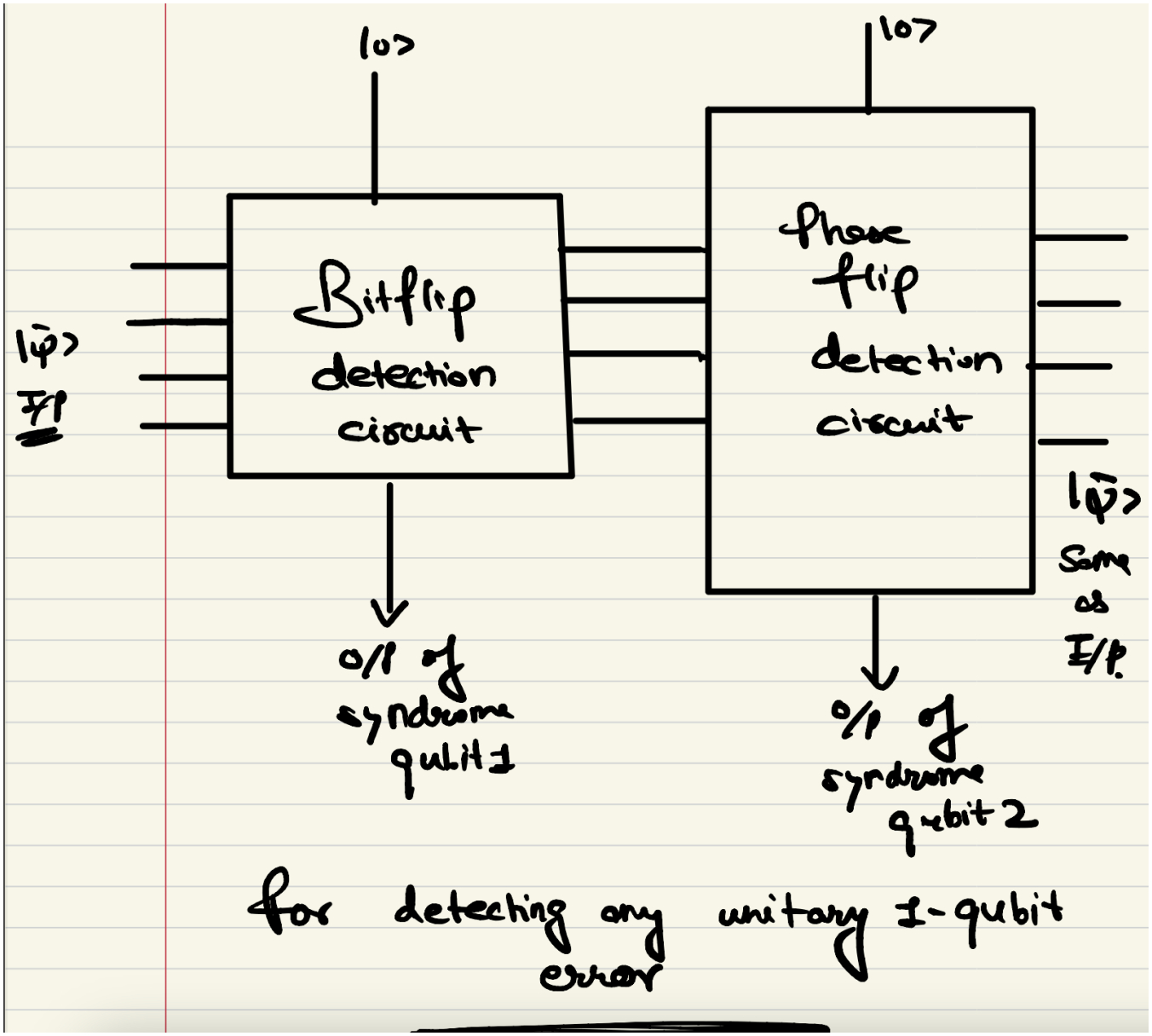
$$CM \left|\overline{\psi}\right\rangle |00\rangle = a \times \left|\overline{\psi}\right\rangle |00\rangle + b \times X_i \left|\overline{\psi}\right\rangle |10\rangle + c \times Z_i \left|\overline{\psi}\right\rangle |01\rangle + d \times Y_i \left|\overline{\psi}\right\rangle |11\rangle$$

Now, based on the measurement we make on the syndrome qubits, we would get one of the states out of the 4 possible cases. Thus, if we measure 00 on the syndrome qubits, that would mean that there was no error. Also, 10 would mean the presence of bit flip error, 01 would mean the presence of phase flip error, and 11 would mean the presence of both bit flip and phase flip error.

Please refer to the circuit diagram below:

In [3]:
```
from IPython.display import Image
Image("CombinedErrorDetectionCircuit.PNG")
```

Out[3]:



Problem 4: Circuits on noisy quantum computers

The IBM Quantum Lab gives public access to a number of their quantum computers. In this problem you'll get to play with them -- and see the effects of noise on your quantum circuits.

Problem 4.1 - Benchmarking individual qubits

In this subproblem, we will benchmark individual qubits on (a simulation of) the 5-qubit `ibmq_essex` device. A typical way to benchmark a qubit is to run a sequence of randomly chosen single-qubit gates $g_1, g_2, \ldots, g_k$, and then running the reverse sequence $g_k^{-1}, g_{k-1}^{-1}, \ldots, g_1^{-1}$ so that overall the effect should be the identity. Of course, each gate will incur some noise, so the state of the qubit will drift over time. One can measure the noise by measuring the qubit at the end of the sequence to see if it stayed in the $|0\rangle$ state.

You will write code to perform the following: for $k = 10, 20, 30, \ldots, 100$, for each qubit $q = 0, 1, 2, \ldots, 4$, pick a sequence of gates $g_1, \ldots, g_k$ where each g_i is chosen randomly from the gate set $\{X, Y, Z, H\}$. Then, on qubit q run the sequence (g_i) forward and then in reverse, and measure the qubit. Do this 1000 times and calculate the percentage $p_{q,k}$ of times that the qubit q ends back in the $|0\rangle$ state.

Below, we've provided two functions. The first function, `benchmark_qubit`, requires you to fill in some code, and it returns a `IBMQJob` object. The second function, `retrieve_job_results`, takes an `IBMQJob` object and gets the measurement counts. (The reason we're dividing this up into two steps is because later we'll run these circuits on a real device).

In [21]:

```
from qiskit import IBMQ, transpile
from qiskit import QuantumCircuit
from qiskit.providers.aer import AerSimulator
from qiskit.providers.fake_provider import FakeEssex
from qiskit.providers.jobstatus import JobStatus

import numpy as np
from matplotlib import pyplot as plt
import random

##########
# This function takes as input
#    - q : qubit number 0 thru 4
#    - k : length of random gate sequence
#    - shots: number of times to run and measure the sequence
#    - device: the device (either simulated or real) to run the circuit on
#
#  Returns:
#    - the IBMQJob object corresponding to the circuit (see https://qiskit.org/documentation/stubs/qiskit.providers.ibmq.job.IBM
##########
def benchmark_qubit(q, k, shots, device):

    circ = QuantumCircuit(5, 1)

    ### WRITE CODE TO GENERATE THE RANDOM SEQUENCE OF LENGTH k, and ITS REVERSAL, ON QUBIT q ####################
    gate_set = set(['X', 'Y', 'Z', 'H'])
    gates_selected = list()

    for curr_gate_idx in range(k):
        curr_gate = random.sample(gate_set, 1)[0]
        gates_selected.append(curr_gate)
        if curr_gate == 'X':
            circ.x(q)
        elif curr_gate == 'Y':
            circ.y(q)
        elif curr_gate == 'Z':
            circ.z(q)
        elif curr_gate == 'H':
            circ.h(q)
        else:
            raise Exception("The gate selected is not valid!")

    for curr_gate_idx in range(k-1, -1, -1):
        curr_gate = gates_selected[curr_gate_idx]
        if curr_gate == 'X':
            circ.x(q)
        elif curr_gate == 'Y':
            circ.y(q)
        elif curr_gate == 'Z':
            circ.z(q)
        elif curr_gate == 'H':
            circ.h(q)
        else:
            raise Exception("The gate selected is not valid!")

    ### END CODE BLOCK ###################

    # measure qubit q, and store it in classical register [0]
    circ.measure([q], [0])

    #this will break down the circuit into gates that can be implemented on the chip
    compiled_circuit = transpile(circ,device,optimization_level=0)

    job = device.run(compiled_circuit,shots=shots)
    return job

##########
# This function takes as input
#    - job: the IBMQJob object corresponding to a circuit being executed
#    - blocking: if True, then this will wait until the circuit results are done executing on the device
#                (either fake or real). If False, then this will first check the status of the job.
#
#  Returns:
#    - if the job is done, then it returns the counts as a dictionary (e.g., { '0000': 356, '0001': 288, ...})
#      otherwise if the job is still running or some other status, then it returns None
##########
def retrieve_job_results(job,blocking=True):

    #if it's blocking, then just go ahead and call result()
    if blocking:
        counts = job.result().get_counts(0)
        return counts
    else:
        #first, check the status
```

```
        job_status = job.status()

        if job_status is JobStatus.DONE:
            counts = job.result().get_counts(0)
            return counts
        else:
            print("The job ",job.job_id," has status: ",job_status)
            return None
```

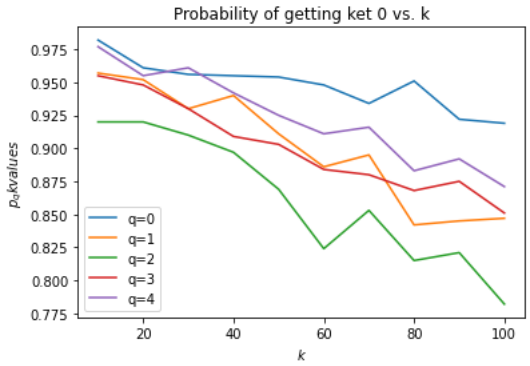
Below, write code using `benchmark_qubit` , `retrieve_job_results` to get the measurement counts and calculate $p_{q,k}$ for $q = 0, 1, 2, 3, 4$ and $k = 10, 20, 30, \ldots, 100$. Then, plot $p_{q,k}$ against k (you can consult https://www.geeksforgeeks.org/using-matplotlib-with-jupyter-notebook/ for an example on how to plot graphs). You should have 5 plots in one graph.

In [27]:

```
fake_device = AerSimulator.from_backend(FakeEssex())

#### WRITE YOUR CODE HERE #################
from collections import defaultdict

pqk_vals_all = defaultdict(dict)
for q in range(5):
    pqk_vals = dict()
    for k in range(10, 101, 10):
        job = benchmark_qubit(q,k,1000,fake_device)
        counts = retrieve_job_results(job)
        print(f'For q: {q} and k: {k}, the counts are as follows:{counts}')
        total_shots = sum(counts.values())
        pqk_vals[k] = float(counts['0'])/total_shots

    pqk_vals_all[q] = pqk_vals

    # Plotting the graph
    plt.plot(pqk_vals.keys(), pqk_vals.values(), label = f'q={q}')

plt.xlabel('$k$')
plt.ylabel('$p_qk values$')
plt.title('Probability of getting ket 0 vs. k')
plt.legend()

plt.show()

# #example code,
# job = benchmark_qubit(0,100,1000,fake_device)
# counts = retrieve_job_results(job)
# print(counts)
# #

# #an example plot, change to display your data instead
# for q in range(5):
#     ks = np.arange(100)
#     p_qks = [k*q + q*np.sin(k) for k in ks]
#     plt.plot(ks, p_qks, label=f'q={q}')
# plt.xlabel('$k$')
# plt.ylabel('$kq + q sin(k)$')
# plt.title('$kq + q sin(k)$ vs. k')
# plt.legend()

# plt.show()

#### END CODE BLOCK ###########################################################
```

```
For q: 0 and k: 10, the counts are as follows:{'1': 18, '0': 982}
For q: 0 and k: 20, the counts are as follows:{'1': 39, '0': 961}
For q: 0 and k: 30, the counts are as follows:{'1': 44, '0': 956}
For q: 0 and k: 40, the counts are as follows:{'1': 45, '0': 955}
For q: 0 and k: 50, the counts are as follows:{'1': 46, '0': 954}
For q: 0 and k: 60, the counts are as follows:{'1': 52, '0': 948}
For q: 0 and k: 70, the counts are as follows:{'1': 66, '0': 934}
For q: 0 and k: 80, the counts are as follows:{'1': 49, '0': 951}
For q: 0 and k: 90, the counts are as follows:{'1': 78, '0': 922}
For q: 0 and k: 100, the counts are as follows:{'1': 81, '0': 919}
For q: 1 and k: 10, the counts are as follows:{'1': 43, '0': 957}
For q: 1 and k: 20, the counts are as follows:{'1': 48, '0': 952}
For q: 1 and k: 30, the counts are as follows:{'1': 70, '0': 930}
For q: 1 and k: 40, the counts are as follows:{'0': 940, '1': 60}
For q: 1 and k: 50, the counts are as follows:{'1': 89, '0': 911}
For q: 1 and k: 60, the counts are as follows:{'1': 114, '0': 886}
For q: 1 and k: 70, the counts are as follows:{'1': 105, '0': 895}
For q: 1 and k: 80, the counts are as follows:{'1': 158, '0': 842}
For q: 1 and k: 90, the counts are as follows:{'1': 155, '0': 845}
For q: 1 and k: 100, the counts are as follows:{'0': 847, '1': 153}
For q: 2 and k: 10, the counts are as follows:{'1': 80, '0': 920}
For q: 2 and k: 20, the counts are as follows:{'1': 80, '0': 920}
For q: 2 and k: 30, the counts are as follows:{'1': 90, '0': 910}
For q: 2 and k: 40, the counts are as follows:{'1': 103, '0': 897}
For q: 2 and k: 50, the counts are as follows:{'0': 869, '1': 131}
For q: 2 and k: 60, the counts are as follows:{'1': 176, '0': 824}
For q: 2 and k: 70, the counts are as follows:{'1': 147, '0': 853}
```

```
For q: 2 and k: 80, the counts are as follows:{'1': 185, '0': 815}
For q: 2 and k: 90, the counts are as follows:{'1': 179, '0': 821}
For q: 2 and k: 100, the counts are as follows:{'1': 218, '0': 782}
For q: 3 and k: 10, the counts are as follows:{'1': 45, '0': 955}
For q: 3 and k: 20, the counts are as follows:{'1': 52, '0': 948}
For q: 3 and k: 30, the counts are as follows:{'1': 70, '0': 930}
For q: 3 and k: 40, the counts are as follows:{'1': 91, '0': 909}
For q: 3 and k: 50, the counts are as follows:{'1': 97, '0': 903}
For q: 3 and k: 60, the counts are as follows:{'1': 116, '0': 884}
For q: 3 and k: 70, the counts are as follows:{'1': 120, '0': 880}
For q: 3 and k: 80, the counts are as follows:{'1': 132, '0': 868}
For q: 3 and k: 90, the counts are as follows:{'1': 125, '0': 875}
For q: 3 and k: 100, the counts are as follows:{'1': 149, '0': 851}
For q: 4 and k: 10, the counts are as follows:{'1': 23, '0': 977}
For q: 4 and k: 20, the counts are as follows:{'1': 45, '0': 955}
For q: 4 and k: 30, the counts are as follows:{'1': 39, '0': 961}
For q: 4 and k: 40, the counts are as follows:{'1': 58, '0': 942}
For q: 4 and k: 50, the counts are as follows:{'1': 75, '0': 925}
For q: 4 and k: 60, the counts are as follows:{'1': 89, '0': 911}
For q: 4 and k: 70, the counts are as follows:{'1': 84, '0': 916}
For q: 4 and k: 80, the counts are as follows:{'1': 117, '0': 883}
For q: 4 and k: 90, the counts are as follows:{'1': 108, '0': 892}
For q: 4 and k: 100, the counts are as follows:{'1': 129, '0': 871}
```



Problem 4.2 - Estimating the single qubit gate noise

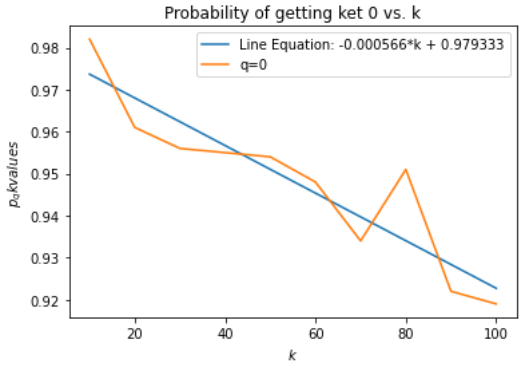
For each qubit q, find a best function $f(k)$ (linear, quadratic, exponential,...) that fits the plots. For example, if the plot of p_{qk} against k looks linear, then you should come up with parameters a, b such that p_{qk} is close to $ak + b$. If it looks like exponential decay, then you should find an approximate $f(k) = ae^{bk} + c$ for some parameters a, b, c.

This function can be used to give a simple model for how noise accumulates on each qubit from single-qubit gates.

In [59]:
```
from scipy.optimize import curve_fit
import numpy as np

def linear_func(k, a, b):
    return a*k + b

def exp_func(k, a, b, c):
    return a*np.exp(b*k) + c
```

In [42]:
```
# For qubit 0
q = 0
values_dict = pqk_vals_all[q]
args,_ = curve_fit(linear_func, list(values_dict.keys()), list(values_dict.values()))
a = args[0]
b = args[1]

print(f"Equation of the line is : {a}*k + {b}")

k_vals = np.linspace(10, 100, 91)
prob_vals = a*k_vals + b

plt.plot(k_vals, prob_vals, label = f'Line Equation: {round(a, 6)}*k + {round(b, 6)}')
plt.plot(values_dict.keys(), values_dict.values(), label = f'q={q}')
plt.xlabel('$k$')
plt.ylabel('$p_qk values$')
plt.title('Probability of getting ket 0 vs. k')
plt.legend()

plt.show()
```

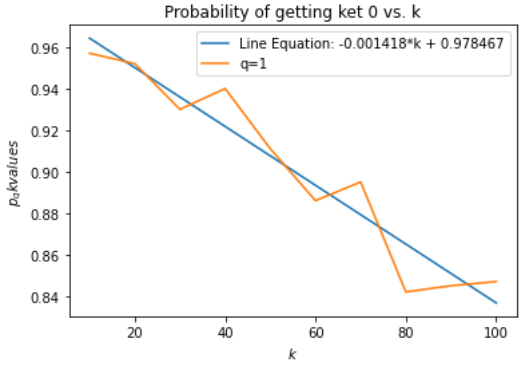
```
Equation of the line is : -0.0005660606082431574*k + 0.9793333333332853
```



In [43]:

```
# For qubit 1
q = 1
values_dict = pqk_vals_all[q]
args,_ = curve_fit(linear_func, list(values_dict.keys()), list(values_dict.values()))
a = args[0]
b = args[1]

print(f"Equation of the line is : {a}*k + {b}")

k_vals = np.linspace(10, 100, 91)
prob_vals = a*k_vals + b

plt.plot(k_vals, prob_vals, label = f'Line Equation: {round(a, 6)}*k + {round(b, 6)}')
plt.plot(values_dict.keys(), values_dict.values(), label = f'q={q}')
plt.xlabel('$k$')
plt.ylabel('$p_qk values$')
plt.title('Probability of getting ket 0 vs. k')
plt.legend()

plt.show()
```

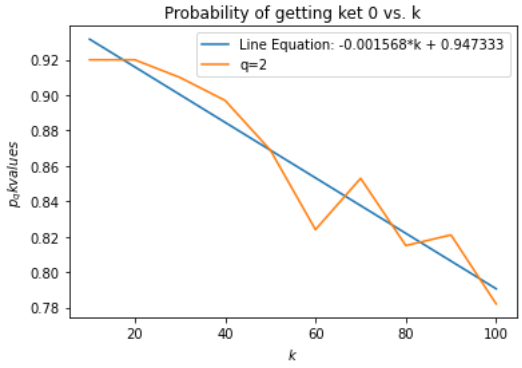
```
Equation of the line is : -0.0014175757575757575*k + 0.9784666666666666
```



In [44]:

```
# For qubit 2
q = 2
values_dict = pqk_vals_all[q]
args,_ = curve_fit(linear_func, list(values_dict.keys()), list(values_dict.values()))
a = args[0]
b = args[1]

print(f"Equation of the line is : {a}*k + {b}")

k_vals = np.linspace(10, 100, 91)
prob_vals = a*k_vals + b

plt.plot(k_vals, prob_vals, label = f'Line Equation: {round(a, 6)}*k + {round(b, 6)}')
plt.plot(values_dict.keys(), values_dict.values(), label = f'q={q}')
plt.xlabel('$k$')
plt.ylabel('$p_qk values$')
plt.title('Probability of getting ket 0 vs. k')
plt.legend()

plt.show()
```

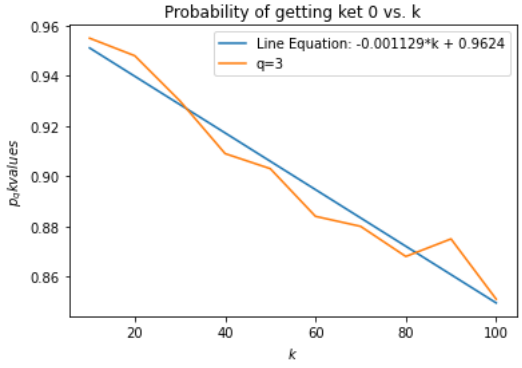
```
Equation of the line is : -0.0015678787859378378*k + 0.9473333332265811
```



In [57]:

```
# For qubit 3
q = 3
values_dict = pqk_vals_all[q]
args,_ = curve_fit(linear_func, list(values_dict.keys()), list(values_dict.values()))
a = args[0]
b = args[1]

print(f"Equation of the line is : {a}*k + {b}")

k_vals = np.linspace(10, 100, 91)
prob_vals = a*k_vals + b

plt.plot(k_vals, prob_vals, label = f'Line Equation: {round(a, 6)}*k + {round(b, 6)}')
plt.plot(values_dict.keys(), values_dict.values(), label = f'q={q}')
plt.xlabel('$k$')
plt.ylabel('$p_qk values$')
plt.title('Probability of getting ket 0 vs. k')
plt.legend()

plt.show()
```

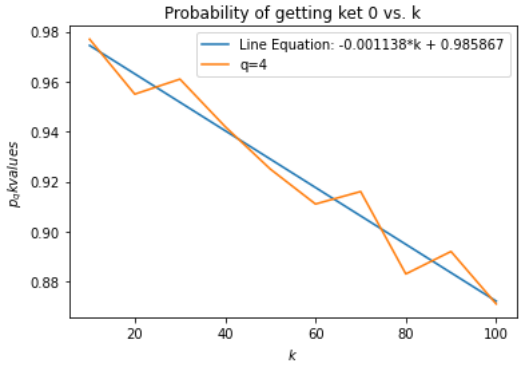
```
Equation of the line is : -0.001129090911274666*k + 0.9623999999999111
```



In [58]:

```
# For qubit 4
q = 4
values_dict = pqk_vals_all[q]
args,_ = curve_fit(linear_func, list(values_dict.keys()), list(values_dict.values()))
a = args[0]
b = args[1]

print(f"Equation of the line is : {a}*k + {b}")

k_vals = np.linspace(10, 100, 91)
prob_vals = a*k_vals + b

plt.plot(k_vals, prob_vals, label = f'Line Equation: {round(a, 6)}*k + {round(b, 6)}')
plt.plot(values_dict.keys(), values_dict.values(), label = f'q={q}')
plt.xlabel('$k$')
plt.ylabel('$p_qk values$')
plt.title('Probability of getting ket 0 vs. k')
plt.legend()

plt.show()
```

```
Equation of the line is : -0.0011375757597595904*k + 0.9858666666666378
```



Solution

As we can see above, the behavior of all the probabilities vs. k is actually linear. The equations for all the qubits are as follows:

For qubit 0: -0.0005660606082431574*k + 0.9793333333332853

For qubit 1: -0.0014175757575757575*k + 0.9784666666666666

For qubit 2: -0.0015678787859378378*k + 0.9473333332265811

For qubit 3: -0.001129090911274666*k + 0.9623999999999111

For qubit 4: -0.0011375757597595904*k + 0.9858666666666378

Problem 4.3 - Benchmarking entanglement generation

Now we benchmark the quantum computer on more complex circuits that involve entangling gates, which will generally be more noisy than single-qubit gates.

For each $r = 2, 3, 4, 5$, let $|\psi_r\rangle$ denote the r-qubit GHZ state

$$\frac{|0\rangle^{\otimes r} + |1\rangle^{\otimes r}}{\sqrt{2}}.$$

When $r = 2$, this is simply the EPR pair we know and love. Let C_r denote a circuit that starts with r zeroes and outputs $|\psi_r\rangle$.

Write code to do the same benchmarking as in Problem 4.1, except now we perform C_r, then C_r^{-1}, then C_r, and so on k times. Ideally, all of these circuits would cancel out so measuring all r qubits will yield zero. However, the gates will be noisy so error will accumulate as k grows larger.

Let p_{rk} denote the percentage of times (out of 1000 shots) that doing C_r and C_r^{-1} for k times yields all zeroes in the r qubits. Plot p_{rk} versus k for $r = 2, 3, 4, 5$ and for $k = 10, 20, 30, \ldots, 100$. There should be 4 plots on one graph.

Important: the `ibmq_essex` device has a very particular connectivity of qubits:

You can only apply 2-qubit gates between the connected nodes. For exampe, you can apply a CNOT between qubits 3 and 1, but not 3 and 2. Thus, when coding your circuit C_r, you may want to judiciously choose which r qubits you use to maximize the performance. The darker shading of a qubit means lower noise rate.

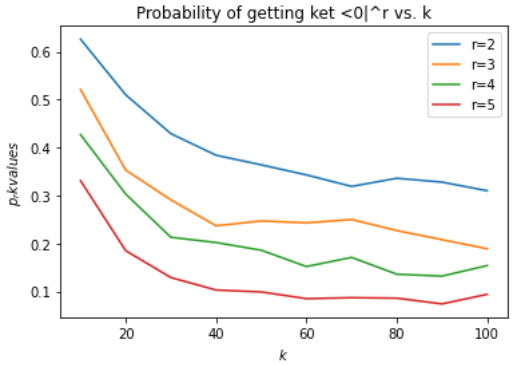
In [71]:

```
##########
# This function takes as input
#    - r : size of GHZ state
#    - k : length of random gate sequence
#    - shots: number of times to run and measure the sequence
#    - device: the device to run on, either simulated or real
#
#  Returns:
#    - the IBMQJob object corresponding to the circuit (see https://qiskit.org/documentation/stubs/qiskit.providers.ibmq.job.IBM
##########
def benchmark_circuit(r, k, shots, device):

    circ = QuantumCircuit(5, r)

    ### WRITE CODE TO GENERATE C_r and its reversal k times ###################

             ## You need to choose your subset of qubits carefully!
    if r == 2:
        qubits_to_use = [1, 2]
        for curr_k in range(k):
            circ.h(qubits_to_use[0])
            circ.cx(qubits_to_use[1], qubits_to_use[0])
            circ.cx(qubits_to_use[1], qubits_to_use[0])
```

```
            circ.h(qubits_to_use[0])
    elif r == 3:
        qubits_to_use = [2, 1, 3]
        for curr_k in range(k):
            circ.h(qubits_to_use[0])
            circ.cx(qubits_to_use[1], qubits_to_use[0])
            circ.cx(qubits_to_use[2], qubits_to_use[1])
            circ.cx(qubits_to_use[2], qubits_to_use[1])
            circ.cx(qubits_to_use[1], qubits_to_use[0])
            circ.h(qubits_to_use[0])
    elif r == 4:
        qubits_to_use = [2, 1, 3, 0]
        for curr_k in range(k):
            circ.h(qubits_to_use[0])
            circ.cx(qubits_to_use[1], qubits_to_use[0])
            circ.cx(qubits_to_use[2], qubits_to_use[1])
            circ.cx(qubits_to_use[3], qubits_to_use[1])
            circ.cx(qubits_to_use[3], qubits_to_use[1])
            circ.cx(qubits_to_use[2], qubits_to_use[1])
            circ.cx(qubits_to_use[1], qubits_to_use[0])
            circ.h(qubits_to_use[0])
    elif r == 5:
        qubits_to_use = [2, 1, 3, 0, 4]
        for curr_k in range(k):
            circ.h(qubits_to_use[0])
            circ.cx(qubits_to_use[1], qubits_to_use[0])
            circ.cx(qubits_to_use[2], qubits_to_use[1])
            circ.cx(qubits_to_use[3], qubits_to_use[1])
            circ.cx(qubits_to_use[4], qubits_to_use[2])
            circ.cx(qubits_to_use[4], qubits_to_use[2])
            circ.cx(qubits_to_use[3], qubits_to_use[1])
            circ.cx(qubits_to_use[2], qubits_to_use[1])
            circ.cx(qubits_to_use[1], qubits_to_use[0])
            circ.h(qubits_to_use[0])
    else:
        raise Exception("This value of r is not valid!")

    ### END CODE BLOCK ###################


    ### WRITE CODE TO MEASURE AND COMPUTE P_RK ####################

    # measure r of the qubits , and store it in the r classical registers
    #for example, if r = 3, circ.measure([1,3,4],[0,1,2]) would mean measuring qubits 1,3,4

    # <--- WRITE MEASUREMENT CODE HERE
    circ.measure(qubits_to_use, list(range(r)))

    #this will break down the circuit into gates that can be implemented on the chip
    compiled_circuit = transpile(circ,device,optimization_level=0)
    job = device.run(compiled_circuit,shots=shots)

    return job

#### WRITE CODE TO CALL benchmark_circuit, retrieve_job_results AND PLOT p_rk versus k #################

fake_device = AerSimulator.from_backend(FakeEssex())

#### WRITE YOUR CODE HERE #################
from collections import defaultdict

prk_vals_all = defaultdict(dict)
for r in range(2, 6):
    prk_vals = dict()
    for k in range(10, 101, 10):
        job = benchmark_circuit(r,k,1000,fake_device)
        counts = retrieve_job_results(job)
        print(f'For r: {r} and k: {k}, the counts are as follows:{counts}')
        total_shots = sum(counts.values())
        prk_vals[k] = float(counts['0'*r])/total_shots

    prk_vals_all[r] = prk_vals

    # Plotting the graph
    plt.plot(prk_vals.keys(), prk_vals.values(), label = f'r={r}')

plt.xlabel('$k$')
plt.ylabel('$p_rk values$')
plt.title('Probability of getting ket <0|^r vs. k')
plt.legend()

plt.show()

#### END CODE BLOCK ############################################################
```

```
For r: 2 and k: 10, the counts are as follows:{'11': 103, '10': 125, '00': 626, '01': 146}
For r: 2 and k: 20, the counts are as follows:{'01': 191, '11': 146, '00': 510, '10': 153}
For r: 2 and k: 30, the counts are as follows:{'11': 163, '01': 242, '10': 166, '00': 429}
For r: 2 and k: 40, the counts are as follows:{'11': 159, '01': 284, '00': 384, '10': 173}
For r: 2 and k: 50, the counts are as follows:{'11': 174, '10': 171, '00': 364, '01': 291}
For r: 2 and k: 60, the counts are as follows:{'10': 194, '00': 343, '11': 180, '01': 283}
```

For r: 2 and k: 70, the counts are as follows:{'00': 319, '10': 197, '11': 196, '01': 288}
For r: 2 and k: 80, the counts are as follows:{'01': 294, '11': 183, '10': 187, '00': 336}
For r: 2 and k: 90, the counts are as follows:{'01': 296, '11': 175, '00': 328, '10': 201}
For r: 2 and k: 100, the counts are as follows:{'00': 310, '10': 196, '11': 183, '01': 311}
For r: 3 and k: 10, the counts are as follows:{'101': 25, '110': 42, '111': 14, '011': 79, '001': 156, '100': 55, '010': 108, '000': 521}
For r: 3 and k: 20, the counts are as follows:{'110': 54, '001': 216, '101': 37, '100': 64, '111': 19, '011': 118, '000': 353, '010': 139}
For r: 3 and k: 30, the counts are as follows:{'110': 68, '111': 52, '011': 120, '101': 52, '001': 215, '100': 71, '000': 291, '010': 131}
For r: 3 and k: 40, the counts are as follows:{'011': 115, '111': 64, '101': 72, '110': 68, '001': 194, '100': 80, '010': 170, '000': 237}
For r: 3 and k: 50, the counts are as follows:{'100': 95, '101': 89, '001': 212, '110': 62, '111': 56, '011': 104, '000': 247, '010': 135}
For r: 3 and k: 60, the counts are as follows:{'101': 82, '110': 56, '001': 212, '011': 129, '111': 58, '100': 100, '010': 120, '000': 243}
For r: 3 and k: 70, the counts are as follows:{'101': 100, '110': 60, '100': 85, '111': 49, '011': 125, '010': 120, '000': 250, '001': 211}
For r: 3 and k: 80, the counts are as follows:{'101': 108, '111': 54, '011': 113, '100': 98, '001': 197, '010': 142, '000': 227, '110': 61}
For r: 3 and k: 90, the counts are as follows:{'110': 59, '111': 68, '011': 115, '100': 109, '001': 234, '010': 111, '000': 208, '101': 96}
For r: 3 and k: 100, the counts are as follows:{'110': 79, '001': 222, '100': 111, '000': 189, '010': 127, '101': 89, '111': 59, '011': 124}
For r: 4 and k: 10, the counts are as follows:{'1101': 1, '1111': 1, '0001': 136, '0100': 52, '1110': 3, '0000': 427, '0010': 120, '1100': 6, '0110': 31, '1000': 42, '0101': 20, '1001': 18, '0111': 16, '0011': 94, '1011': 7, '1010': 26}
For r: 4 and k: 20, the counts are as follows:{'1101': 6, '1110': 11, '1100': 13, '0001': 161, '1111': 8, '0011': 80, '0111': 28, '1011': 28, '0000': 303, '0010': 126, '0110': 33, '1000': 44, '1010': 31, '0100': 59, '1001': 23, '0101': 46}
For r: 4 and k: 30, the counts are as follows:{'1111': 16, '1100': 21, '0100': 53, '0001': 155, '0000': 213, '0010': 121, '0111': 40, '0011': 89, '1011': 33, '1101': 14, '1000': 54, '0110': 53, '1010': 26, '1110': 11, '1001': 43, '0101': 58}
For r: 4 and k: 40, the counts are as follows:{'1101': 13, '1110': 11, '1011': 31, '0010': 106, '0000': 202, '1100': 27, '0011': 70, '0111': 34, '0100': 52, '1000': 74, '0110': 45, '0001': 179, '1111': 14, '1001': 61, '0101': 37, '1010': 44}
For r: 4 and k: 50, the counts are as follows:{'1011': 38, '0101': 58, '1001': 53, '0000': 186, '0010': 93, '1100': 33, '1010': 33, '1000': 57, '0110': 44, '1101': 13, '1111': 16, '0001': 172, '0011': 93, '0111': 43, '1110': 14, '0100': 54}
For r: 4 and k: 60, the counts are as follows:{'1100': 21, '0001': 154, '1111': 21, '0000': 152, '0010': 78, '0110': 55, '1000': 69, '1001': 64, '0101': 56, '1110': 19, '0100': 55, '0111': 48, '0011': 97, '1011': 39, '1010': 44, '1101': 28}
For r: 4 and k: 70, the counts are as follows:{'1101': 21, '1110': 24, '0111': 50, '0011': 89, '1011': 48, '0000': 171, '0010': 77, '1000': 61, '0110': 51, '1010': 38, '0001': 138, '1111': 29, '1100': 26, '0101': 70, '1001': 60, '0100': 47}
For r: 4 and k: 80, the counts are as follows:{'1111': 18, '1010': 50, '1100': 32, '0011': 79, '0111': 37, '1011': 51, '0000': 136, '0010': 85, '0001': 148, '1101': 24, '0110': 50, '1000': 76, '1110': 22, '1001': 59, '0101': 78, '0100': 55}
For r: 4 and k: 90, the counts are as follows:{'1100': 32, '1011': 40, '0010': 75, '0000': 132, '0111': 46, '0011': 86, '0001': 134, '1111': 15, '0100': 73, '1000': 71, '0110': 48, '1101': 23, '1001': 80, '0101': 57, '1110': 26, '1010': 62}
For r: 4 and k: 100, the counts are as follows:{'1101': 39, '1110': 22, '0000': 154, '0010': 79, '1100': 31, '0110': 52, '1000': 64, '0111': 50, '0011': 74, '1011': 48, '1001': 72, '0101': 54, '0001': 150, '1111': 18, '0100': 53, '1010': 40}
For r: 5 and k: 10, the counts are as follows:{'01111': 1, '01101': 3, '01010': 20, '00110': 62, '00001': 98, '10010': 21, '10101': 8, '11100': 3, '00010': 83, '00100': 43, '01011': 5, '10111': 13, '10100': 22, '00011': 60, '00111': 22, '10110': 29, '10011': 7, '10000': 51, '10001': 14, '01001': 8, '00101': 18, '11010': 4, '11111': 2, '11000': 5, '11101': 1, '01110': 10, '00000': 331, '11001': 3, '11110': 4, '01000': 42, '01100': 7}
For r: 5 and k: 20, the counts are as follows:{'11101': 3, '11110': 4, '10010': 28, '11100': 9, '10101': 15, '10110': 39, '00001': 110, '00110': 62, '00111': 39, '00011': 56, '00010': 63, '11001': 6, '00000': 185, '00101': 24, '01001': 19, '10001': 28, '11010': 9, '11011': 4, '11111': 6, '11000': 8, '01101': 13, '10000': 52, '00100': 56, '01010': 20, '10111': 24, '10100': 26, '10011': 21, '01000': 28, '01100': 13, '01111': 10, '01011': 11, '01110': 9}
For r: 5 and k: 30, the counts are as follows:{'11110': 5, '01011': 12, '01100': 4, '01000': 31, '00101': 44, '10001': 43, '01001': 25, '11010': 10, '10000': 48, '00100': 38, '00011': 51, '00111': 46, '11101': 9, '10110': 34, '01111': 11, '00110': 64, '00001': 95, '10010': 31, '11100': 9, '10101': 25, '00010': 73, '10011': 30, '10111': 21, '10100': 23, '01101': 16, '11000': 12, '11011': 5, '11111': 5, '01010': 27, '00000': 129, '11001': 10, '01110': 14}
For r: 5 and k: 40, the counts are as follows:{'11101': 6, '01110': 15, '01001': 32, '11010': 12, '10001': 35, '00101': 41, '10010': 34, '10101': 23, '11100': 15, '11111': 14, '11011': 9, '10000': 44, '01010': 22, '10011': 23, '00011': 55, '00111': 34, '10111': 29, '10100': 28, '00010': 69, '10110': 34, '00001': 88, '00110': 57, '00100': 54, '01111': 9, '01000': 32, '01100': 16, '01011': 23, '11000': 17, '11110': 9, '00000': 103, '11001': 10, '01101': 8}
For r: 5 and k: 50, the counts are as follows:{'01101': 11, '01111': 21, '10010': 20, '01001': 37, '11010': 13, '00101': 48, '10001': 53, '11000': 18, '00111': 44, '00011': 52, '10101': 35, '10100': 32, '10111': 24, '11100': 19, '01010': 31, '10000': 48, '11111': 7, '11011': 14, '00100': 45, '00010': 59, '11001': 21, '00000': 99, '11101': 11, '01110': 16, '10110': 33, '10011': 14, '00001': 69, '00110': 27, '01000': 27, '01100': 19, '11110': 12, '01011': 21}
For r: 5 and k: 60, the counts are as follows:{'01111': 18, '01011': 28, '11000': 20, '00110': 39, '00001': 75, '10101': 33, '10010': 35, '11100': 17, '11111': 13, '11011': 14, '10000': 34, '00011': 48, '00111': 36, '00010': 53, '11001': 13, '00000': 85, '10001': 46, '00101': 34, '11010': 17, '01001': 22, '01110': 22, '01101': 22, '01010': 26, '01000': 42, '01100': 24, '00100': 46, '10110': 35, '11101': 10, '10011': 31, '11110': 9, '10111': 31, '10100': 22}
For r: 5 and k: 70, the counts are as follows:{'10101': 34, '01111': 11, '00010': 51, '11100': 12, '10010': 33, '11000': 17, '01100': 24, '01000': 36, '00001': 93, '00110': 47, '00101': 35, '11010': 15, '10001': 47, '01001': 29, '10000': 50, '11111': 9, '11011': 5, '00100': 45, '01011': 27, '00011': 51, '00111': 36, '11001': 20, '00000': 87, '01101': 20, '11110': 8, '10110': 26, '01110': 24, '01010': 22, '10111': 29, '10100': 22, '11101': 11, '10011': 24}
For r: 5 and k: 80, the counts are as follows:{'10101': 25, '01111': 22, '10001': 49, '01001': 35, '00101': 22, '11010': 14, '10011': 30, '01100': 21, '01000': 42, '00100': 52, '11100': 16, '00110': 38, '00001': 75, '10010': 26, '10000': 39, '00000': 86, '11001': 20, '00111': 40, '00011': 42, '00010': 55, '01110': 18, '01101': 12, '01010': 35, '11011': 18, '11111': 12, '11000': 20, '11101': 16, '10110': 26, '01011': 27, '10111': 20, '10100': 27, '11110': 20}
For r: 5 and k: 90, the counts are as follows:{'11100': 14, '11111': 16, '11011': 18, '01101': 23, '01010': 21, '11010': 21, '01001': 45, '00101': 36, '10001': 33, '00100': 38, '10011': 34, '10101': 27, '10010': 25, '11101': 15, '00111': 38, '00011': 33, '01100': 21, '01000': 46, '00001': 80, '00110': 37, '10000': 33, '10111': 29, '10100': 34, '01011': 32, '00000': 74, '11001': 20, '00010': 56, '01110': 16, '11000': 24, '01111': 21, '10110': 27, '11110': 13}
For r: 5 and k: 100, the counts are as follows:{'10101': 13, '11010': 17, '01001': 44, '00101': 32, '10001': 40, '10010': 32, '11100': 24, '10110': 27, '01110': 21, '01010': 34, '01100': 23, '01000': 40, '00001': 73, '00110': 38, '00111': 38, '00011': 54, '01011': 38, '11000': 21, '11011': 21, '11111': 9, '10000': 37, '00100': 27, '00010': 44, '10111': 28, '10100': 26, '10011': 23, '11110': 15, '00000': 94, '11001': 19, '01111': 16, '11101': 11, '01101': 21}



Problem 4.4 - Estimating noise of the GHZ circuit

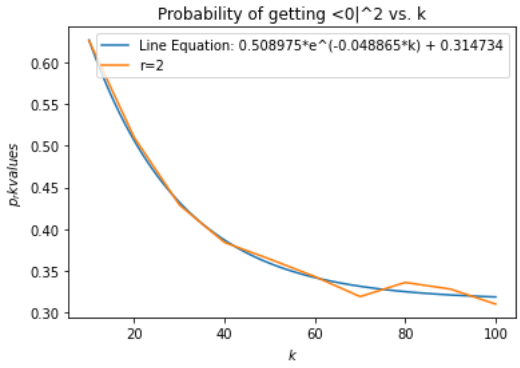
For each $r = 2, 3, 4, 5$ find a best function $f(k)$ (linear, quadratic, exponential,...) that fits the plots.

These functions can be used to give a simple model for how noise accumulates at a global level for the GHZ circuit. How much worse is this than the single-qubit gates situation?

In [93]:
```
from scipy.optimize import curve_fit
import numpy as np

def linear_func(k, a, b):
    return a*k + b

def exp_func(k, a, b, c):
    return a*np.exp(-b*k) + c
```

In [94]:
```
# For r = 2
r = 2
values_dict = prk_vals_all[r]
args,_ = curve_fit(exp_func, list(values_dict.keys()), list(values_dict.values()), method = 'dogbox')
a = args[0]
b = args[1]
c = args[2]

print(f"Equation of the Function Fitted is : {a}*e^(-{b}*k) + {c}")

k_vals = np.linspace(10, 100, 91)
prob_vals = a*np.exp(-b*k_vals) + c

plt.plot(k_vals, prob_vals, label = f'Line Equation: {round(a, 6)}*e^(-{round(b, 6)}*k) + {round(c, 6)}')
plt.plot(values_dict.keys(), values_dict.values(), label = f'r={r}')
plt.xlabel('$k$')
plt.ylabel('$p_rk values$')
plt.title(f'Probability of getting <0|^{r} vs. k')
plt.legend()

plt.show()
```

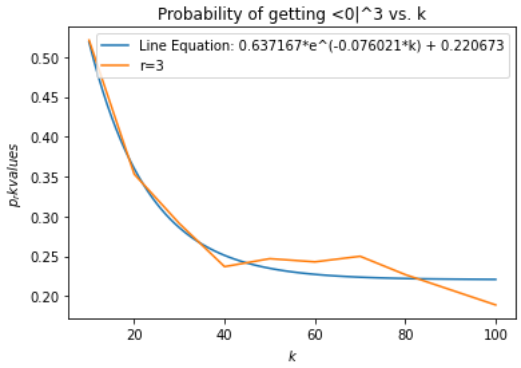
```
Equation of the Function Fitted is : 0.5089754373848295*e^(-0.04886494449080688*k) + 0.31473443695748876
```



In [96]:
```
# For r = 3
r = 3
values_dict = prk_vals_all[r]
args,_ = curve_fit(exp_func, list(values_dict.keys()), list(values_dict.values()), method = 'dogbox')
a = args[0]
b = args[1]
c = args[2]

print(f"Equation of the Function Fitted is : {a}*e^(-{b}*k) + {c}")
```

```
k_vals = np.linspace(10, 100, 91)
prob_vals = a*np.exp(-b*k_vals) + c

plt.plot(k_vals, prob_vals, label = f'Line Equation: {round(a, 6)}*e^(-{round(b, 6)}*k) + {round(c, 6)}')
plt.plot(values_dict.keys(), values_dict.values(), label = f'r={r}')
plt.xlabel('$k$')
plt.ylabel('$p_rk values$')
plt.title(f'Probability of getting <0|^{r} vs. k')
plt.legend()

plt.show()
```

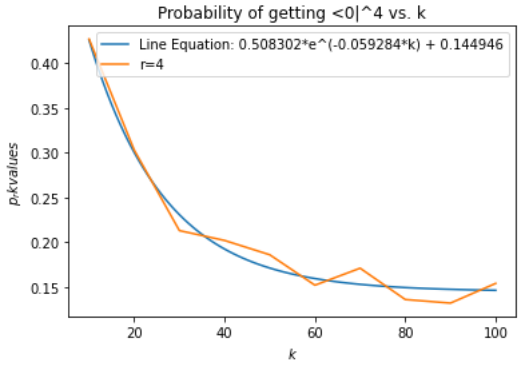
```
Equation of the Function Fitted is : 0.6371666133947282*e^(-0.07602071097749769*k) + 0.2206732671825904
```



In [97]:

```
# For r = 4
r = 4
values_dict = prk_vals_all[r]
args,_ = curve_fit(exp_func, list(values_dict.keys()), list(values_dict.values()), method = 'dogbox')
a = args[0]
b = args[1]
c = args[2]

print(f"Equation of the Function Fitted is : {a}*e^(-{b}*k) + {c}")

k_vals = np.linspace(10, 100, 91)
prob_vals = a*np.exp(-b*k_vals) + c

plt.plot(k_vals, prob_vals, label = f'Line Equation: {round(a, 6)}*e^(-{round(b, 6)}*k) + {round(c, 6)}')
plt.plot(values_dict.keys(), values_dict.values(), label = f'r={r}')
plt.xlabel('$k$')
plt.ylabel('$p_rk values$')
plt.title(f'Probability of getting <0|^{r} vs. k')
plt.legend()

plt.show()
```

```
Equation of the Function Fitted is : 0.5083024286403495*e^(-0.059284336188076*k) + 0.14494603050585864
```

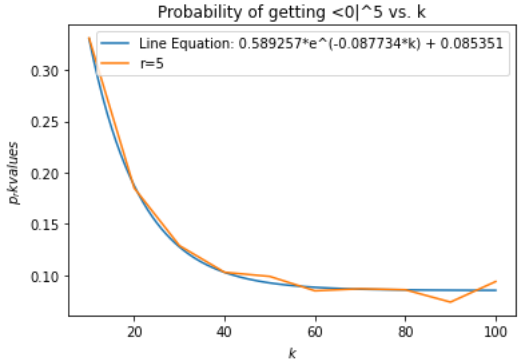


In [98]:

```
# For r = 5
r = 5
values_dict = prk_vals_all[r]
args,_ = curve_fit(exp_func, list(values_dict.keys()), list(values_dict.values()), method = 'dogbox')
a = args[0]
b = args[1]
c = args[2]

print(f"Equation of the Function Fitted is : {a}*e^(-{b}*k) + {c}")

k_vals = np.linspace(10, 100, 91)
prob_vals = a*np.exp(-b*k_vals) + c

plt.plot(k_vals, prob_vals, label = f'Line Equation: {round(a, 6)}*e^(-{round(b, 6)}*k) + {round(c, 6)}')
plt.plot(values_dict.keys(), values_dict.values(), label = f'r={r}')
plt.xlabel('$k$')
```

```
plt.ylabel('$p_rk values$')
plt.title(f'Probability of getting <0|^{r} vs. k')
plt.legend()

plt.show()
```

```
Equation of the Function Fitted is : 0.5892566303718016*e^(-0.0877335886099356*k) + 0.08535115270679851
```



Solution

By looking at the plot above, we can clearly see that the behavior of the probability for decay is that of an exponential function.

Also, by using the code above, I found the functions to be:

For r = 2: 0.5089754373848295*e^(-0.04886494449080688*k) + 0.31473443695748876

For r = 3: 0.6371666133947282*e^(-0.07602071097749769*k) + 0.2206732671825904

For r = 4: 0.5083024286403495*e^(-0.059284336188076*k) + 0.14494603050585864

For r = 5: 0.5892566303718016*e^(-0.0877335886099356*k) + 0.08535115270679851

Also, this is much worse than the single qubit case. Just considering the case when there was just a single qubit, the minimum value of probability for qubit 2 (and k = 100) was 0.782 while here the minimum's are as follows for different number of entangled qubits (and for k = 100):

For 2 entangled bits case: 0.310

For 3 entangled bits case: 0.189

For 4 entangled bits case: 0.154

For 5 entangled bits case: 0.094

Problem 4.5 - Running this on an *actual* quantum computer

So far you've been running all this on a simulated version of the 5-qubit `ibmq_essex` device (which has been retired). Now let's actually run it on a *real* quantum computer -- how exciting!

If you look at the available systems on https://quantum-computing.ibm.com/services/resources, you will see that there are a number of devices (`ibmq_belem` , `ibm_perth` , `ibm_lagos` , `ibm_nairobi` , etc) with varying numbers of qubits, and with different levels of busy-ness (some have more jobs in the queue than others).

If you haven't already, please add your IBM Quantum login email to the spreadsheet

https://docs.google.com/spreadsheets/d/1tB27ZyoDozMzBN9Ya1-bEx6f_pa-VfibDlXVrkA9XCQ/edit?usp=sharing

so that you can get the special education access to their machines (your jobs are processed faster).

For this problem, you will need to run this on the IBM Quantum Lab. The next two commands will show whether you have the special education access.

In [99]:
```
IBMQ.load_account()
IBMQ.providers()
```

Out[99]:
```
[<AccountProvider for IBMQ(hub='ibm-q', group='open', project='main')>,
 <AccountProvider for IBMQ(hub='ibm-q-education', group='columbia-uni-2', project='computer-science')>]
```

If you don't see something like `<AccountProvider for IBMQ(hub='ibm-q-education', group='columbia-uni-2', project='computer-science')>` in the printed list, then e-mail Prof. Yuen and he will add you as a collaborator.

The next code will get the machines that are available to you:

In [100…
```
provider = IBMQ.get_provider(hub='ibm-q-education')
provider.backends()
```

Out[100…
```
[<IBMQSimulator('ibmq_qasm_simulator') from IBMQ(hub='ibm-q-education', group='columbia-uni-2', project='computer-science')>,
 <IBMQBackend('ibmq_lima') from IBMQ(hub='ibm-q-education', group='columbia-uni-2', project='computer-science')>,
 <IBMQBackend('ibmq_belem') from IBMQ(hub='ibm-q-education', group='columbia-uni-2', project='computer-science')>,
 <IBMQBackend('ibmq_quito') from IBMQ(hub='ibm-q-education', group='columbia-uni-2', project='computer-science')>,
 <IBMQSimulator('simulator_statevector') from IBMQ(hub='ibm-q-education', group='columbia-uni-2', project='computer-science')>,
 <IBMQSimulator('simulator_mps') from IBMQ(hub='ibm-q-education', group='columbia-uni-2', project='computer-science')>,
 <IBMQSimulator('simulator_stabilizer') from IBMQ(hub='ibm-q-education', group='columbia-uni-2', project='computer-science')>,
 <IBMQBackend('ibmq_jakarta') from IBMQ(hub='ibm-q-education', group='columbia-uni-2', project='computer-science')>,
 <IBMQBackend('ibmq_manila') from IBMQ(hub='ibm-q-education', group='columbia-uni-2', project='computer-science')>,
 <IBMQBackend('ibm_lagos') from IBMQ(hub='ibm-q-education', group='columbia-uni-2', project='computer-science')>,
 <IBMQBackend('ibm_nairobi') from IBMQ(hub='ibm-q-education', group='columbia-uni-2', project='computer-science')>,
 <IBMQBackend('ibm_perth') from IBMQ(hub='ibm-q-education', group='columbia-uni-2', project='computer-science')>,
 <IBMQBackend('ibm_oslo') from IBMQ(hub='ibm-q-education', group='columbia-uni-2', project='computer-science')>]
```

Next, pick a machine to use. In our simulated code above, we choose `ibmq_essex`, but you have to choose some other device (because essex has been retired). You should check https://quantum-computing.ibm.com/services/resources to see which ones have the shortest line.

In [108…
```
real_device = provider.get_backend('ibm_perth')
real_device.status()
```

Out[108…
```
<qiskit.providers.models.backendstatus.BackendStatus object at 0x7f81e42c8bb0>
```
name: ibm_perth
version: 1.1.38, **pending jobs**: 43
status: active

Now that you've loaded a device, now you can run the same code (`benchmark_qubit` and `benchmark_circuit`), except by passing `real_device` to the functions this time your circuits will be run on some qubits somewhere in upstate New York! Your goal in this part of the problem is to do the same benchmarking, and compare the results with simulation. Since time on the IBM computers are limited, we only ask you to plot p_{qk} versus k for just one of the qubits, and to only plot p_{rk} versus k for $r = 3$.

Unlike with the simulation code, your circuits will be queued after you call `benchmark_qubit` or `benchmark_circuit`. Each call will return a job that takes time to run (usually a few minutes per job). A hundred circuits will likely take a whole day to finish. (So be sure to test your code first in simulation before submitting jobs to the IBM quantum machines!)

To avoid situations where you lose the result of your jobs, in this part you should structure your code as follows:

1. Call the `benchmark_qubit` and `benchmark_circuit` code all at once, and save the job IDs to a file.
2. Periodically check the status of your jobs (either programmatically or using the "Jobs" control panel in IBM Quantum Lab).
3. Once you see they're done, retrieve the results as before.

Below we provide some helper functions to save jobs to a file.

In [109…
```
##########
# This function takes as input
#    - list_of_jobs: self-explanatory
#    - filename: file to store the job_ids
##########
def save_jobs_to_file(list_of_jobs,filename):

    #open the file
    with open(filename,'w') as f:
        for job in list_of_jobs:
            f.write(job.job_id())
            f.write('\n')

##########
# This function takes as input
#    - name_of_device: this is the name of the device you submitted the jobs to, such as `ibmq_belem` or `ibm_nairobi`, etc.
#    - filename: file that has all the job_ids
#
# Returns: a list_of_jobs
##########
def load_jobs_from_file(name_of_device,filename):
    IBMQ.load_account()
    provider = IBMQ.get_provider(hub='ibm-q-education')
    device = provider.get_backend(name_of_device)

    list_of_jobs = []
    with open(filename,'r') as f:
        job_ids = f.readlines()
        for job_id in job_ids:
            job = device.retrieve_job(job_id.strip())
            list_of_jobs.append(job)

    return list_of_jobs


##########
# This function takes as input
#    - list_of_jobs: self-explanatory
##########
def print_job_statuses(list_of_jobs):
    for job in list_of_jobs:
        print("id: ",job.job_id()," status: ", job.status())
```

Next, write code for Step 1. You should use `save_jobs_to_file` to store your work.

Warning: If you execute it multiple times, it will resubmit the same jobs! We suggest testing it out with one or two circuit jobs first before scaling up.

In [110…]
```
### INSERT YOUR STEP 1 CODE HERE ####################
list_of_jobs = list()

for k in range(10, 101, 10):
    qubit_job = benchmark_qubit(4, k, 1000, real_device)
    list_of_jobs.append(qubit_job)

for k in range(10, 101, 10):
    circuit_job = benchmark_circuit(3, k, 1000, real_device)
    list_of_jobs.append(circuit_job)

save_jobs_to_file(list_of_jobs, "jobs_to_run.txt")

############################################
```

Next, write some code to load the jobs ids and check on their status. You can run this block even if you've re-opened the Jupyter notebook.

In [116…]
```
### INSERT YOUR STEP 2 CODE HERE ####################

list_of_jobs = load_jobs_from_file('ibm_perth','jobs_to_run.txt')
print_job_statuses(list_of_jobs)

############################################
```

```
ibmqfactory.load_account:WARNING:2022-12-02 13:40:04,304: Credentials are already in use. The existing account in the session wi
ll be replaced.
id:  6389ce6873ce0615ba0656a3  status:  JobStatus.DONE
id:  6389ce69c67086428690bf28  status:  JobStatus.DONE
id:  6389ce6ae763b13da3cb96c5  status:  JobStatus.DONE
id:  6389ce6b68d7fa9319451ff0  status:  JobStatus.DONE
id:  6389ce6ca6b95041db59207c  status:  JobStatus.DONE
id:  6389ce6de763b14916cb96c6  status:  JobStatus.DONE
id:  6389ce6ea6b950330159207d  status:  JobStatus.DONE
id:  6389ce6fc505e12ee0657416  status:  JobStatus.DONE
id:  6389ce70f9a8a50db5a5677d  status:  JobStatus.DONE
id:  6389ce71d60353248ef249b3  status:  JobStatus.DONE
id:  6389ce72e763b1f305cb96c7  status:  JobStatus.DONE
id:  6389ce7373ce06026e0656a4  status:  JobStatus.DONE
id:  6389ce749588b02147d602ad  status:  JobStatus.DONE
id:  6389ce75a6b950314359207e  status:  JobStatus.DONE
id:  6389ce76e763b19e05cb96c8  status:  JobStatus.DONE
id:  6389ce77a6b9505ec659207f  status:  JobStatus.DONE
id:  6389ce78e763b106a6cb96c9  status:  JobStatus.DONE
id:  6389ce789588b0578cd602ae  status:  JobStatus.DONE
id:  6389ce7973ce0659360656a6  status:  JobStatus.DONE
id:  6389ce7a92dadcd92f19a7c3  status:  JobStatus.DONE
```

Finally, once you see all the job statuses are done, retrieve the job results. Remember that `retrieve_job_results` takes in an argument called `blocking`, which you should set to `False` here.

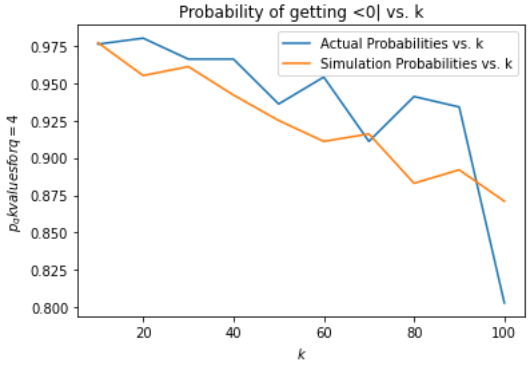
In [118…]
```
### INSERT YOUR STEP 3 CODE HERE ####################

single_qubit_jobs = list_of_jobs[:10]
circuit_jobs = list_of_jobs[10:]

single_qubit_jobs_counts = dict()
k = 10
for job in single_qubit_jobs:
    counts = retrieve_job_results(job, blocking=False)
    single_qubit_jobs_counts[k] = float(counts['0']) / 1000
    k += 10

circuit_jobs_counts = dict()
k = 10
for job in circuit_jobs:
    counts = retrieve_job_results(job, blocking=False)
    circuit_jobs_counts[k] = float(counts['000']) / 1000
    k += 10

print("For Single Qubit Jobs for qubit 1 and for different k's, the probabilities are: ")
print(single_qubit_jobs_counts)

print("For Circuit Jobs with r = 3 entangled states and for different k's, the probabilities are: ")
print(circuit_jobs_counts)
############################################
```

```
For Single Qubit Jobs for qubit 1 and for different k's, the probabilities are:
{10: 0.976, 20: 0.98, 30: 0.966, 40: 0.966, 50: 0.936, 60: 0.954, 70: 0.911, 80: 0.941, 90: 0.934, 100: 0.803}
For Circuit Jobs with r = 3 entangled states and for different k's, the probabilities are:
{10: 0.343, 20: 0.197, 30: 0.204, 40: 0.215, 50: 0.233, 60: 0.21, 70: 0.227, 80: 0.177, 90: 0.184, 100: 0.196}
```

In [125…]
```
import statistics
```

```
probs_vals_q_4 = pqk_vals_all[4]

print(f"For Simulations: the mean: {statistics.mean(probs_vals_q_4.values())}, \
      median: {statistics.median(probs_vals_q_4.values())}  of the data!")

print(f"For Actual Hardware: the mean: {statistics.mean(single_qubit_jobs_counts.values())}, \
        median: {statistics.median(single_qubit_jobs_counts.values())}  of the data!")

plt.plot(single_qubit_jobs_counts.keys(), single_qubit_jobs_counts.values(), label = f'Actual Probabilities vs. k')
plt.plot(probs_vals_q_4.keys(), probs_vals_q_4.values(), label = f'Simulation Probabilities vs. k')
plt.xlabel('$k$')
plt.ylabel('$p_qk values for q = 4$')
plt.title(f'Probability of getting <0| vs. k')
plt.legend()

plt.show()
```

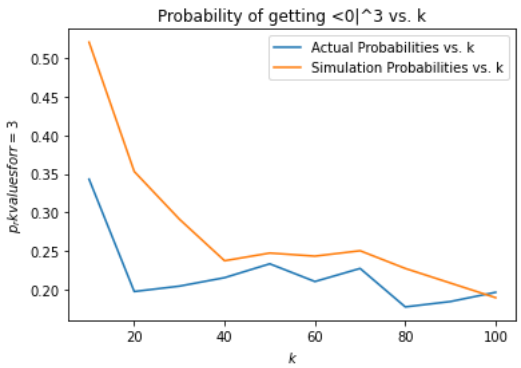
```
For Simulations: the mean: 0.9233,       median: 0.9205000000000001  of the data!
For Actual Hardware: the mean: 0.9367,         median: 0.9475  of the data!
```



In [126…

```
probs_vals_r_3 = prk_vals_all[3]

print(f"For Simulations: the mean: {statistics.mean(probs_vals_r_3.values())}, \
      median: {statistics.median(probs_vals_r_3.values())}  of the data!")

print(f"For Actual Hardware: the mean: {statistics.mean(circuit_jobs_counts.values())}, \
        median: {statistics.median(circuit_jobs_counts.values())}  of the data!")

plt.plot(circuit_jobs_counts.keys(), circuit_jobs_counts.values(), label = f'Actual Probabilities vs. k')
plt.plot(probs_vals_r_3.keys(), probs_vals_r_3.values(), label = f'Simulation Probabilities vs. k')
plt.xlabel('$k$')
plt.ylabel('$p_rk values for r = 3$')
plt.title(f'Probability of getting <0|^3 vs. k')
plt.legend()

plt.show()
```

```
For Simulations: the mean: 0.2766,       median: 0.245  of the data!
For Actual Hardware: the mean: 0.21860000000000002,         median: 0.207  of the data!
```



How do the results from the actual hardware compare with simulation? Quantify the differences, if any.

Solution

For Single qubit case:

The probability of getting $|0\rangle$ back is marginally higher for the actual hardware case. Although at $K = 100$, the probability drops a lot for the actual hardware case but other than that it's marginally better (higher probability).

For Simulations: the mean and median of the data are 0.9233 and 0.9205 respectively.

For Actual hardware: the mean and median of the data are 0.9367 and 0.9475 respectively.

For the (3) entangled states case:

The probability of getting $|0\rangle^{\otimes 3}$ back is greater in general for the simulation case than on the actual hardware. Although at $k = 100$, the probability is nearly the same for both of them (a litter higher probability on actual hardware), in general, the actual hardware peforms poorly in comparison to the simulations.

For Simulations: the mean and median of the data are 0.2766 and 0.245 respectively.

For Actual hardware: the mean and median of the data are 0.2186 and 0.207 respectively.

Problem 5: Quantum Tug-of-War!

In this part of the problem set you will write a strategy for a quantum video game. See the accompanying Jupyter notebook `QTugofwar.ipynb` for details.

In [ ]: