Quantum Computing

Prof. Dr. Christoph Lehner

Lecture for Winter of 2022/23

Contents

1 Ideal quantum computing **5**

1.1 Introduction to digital quantum computing 5

1.1.1 Qubits, gates, measurements 5

1.1.2 Common one-qubit gates 6

1.1.3 Common two-qubit gates 7

1.1.4 Quantum circuits 8

1.1.5 Example: Bell states 9

1.1.6 Quantum simulator 10

1.1.7 Many-qubit gates 10

1.1.8 Quantum parallelism 12

1.2 Quantum algorithms 13

1.2.1 Deutsch–Jozsa algorithm 13

1.2.2 Grover algorithm 16

1.2.3 Parenthesis: Arithmetic gates, ancilla and garbage qubits, uncomputing 18

1.2.4 Example: Solving an equation using Grover's algorithm . 23

1.2.5 The quantum Fourier transform 29

1.2.6 Example: Addition through Fourier transform 31

1.2.7 Phase estimation 33

1.2.8 Example: Order-finding 41

1.2.9 Application: Shor's algorithm 56

1.2.10 Parenthesis: RSA encryption 60

2 Real quantum computing **65**

2.1 Real quantum hardware 65

2.1.1 Characteristics of real quantum computers 65

2.1.2 Survery of universal digital quantum computers (2020) . . 68

2.1.3 Example of hardware implementation: IBM 5 qubit systems 70

2.2 Formal treatment of quantum noise 73

2.2.1 Principal system and environment 73

2.2.2 Mixed states, density matrix, and quantum operations . . 74

2.2.3 Noise channels 77

2.2.4 Fidelity and trace distance 79

2.2.5 Simulating quantum noise 80

2.3 Quantum Error Correction 91
2.3.1 The three-qubit bit flip code 91
2.3.2 The three-qubit phase flip code 96
2.3.3 The Shor code 99
2.3.4 Stabilizer formalism 103
2.3.5 Gottesman Knill Theorem 108
2.3.6 Fault tolerance – general idea 110
2.3.7 Threshold theorem 111
2.3.8 Fault tolerant operations 112

3 Scientific quantum computing **115**
3.1 One-dimensional spin-chain 115
3.2 One-dimensional free particle 126
3.3 One-dimensional particle in time-independent potential 133
3.4 One-dimensional real scalar quantum field theory 140

Chapter 1

Ideal quantum computing

1.1 Introduction to digital quantum computing

1.1.1 Qubits, gates, measurements

- A single qubit is a vector of unit length in the two-dimensional complex vector space S spanned by

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \qquad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}. \tag{1.1}$$

- The direct product of N vector spaces S creates the vector space S^N

$$S^N \equiv \bigotimes_{i=1}^{N} S \tag{1.2}$$

with dimensionality 2^N. A state of N qubits is represented by a vector of unit length in S^N.

- Example: A two-qubit vector space is spanned by

$$|00\rangle = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad |01\rangle = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \quad |10\rangle = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \quad |11\rangle = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}. \tag{1.3}$$

- In general, the i-th component of a basis vector of S^N will be written as

$$(|b_{N-1}b_{N-2}\cdots b_0\rangle)_i = \begin{cases} 1 & \text{for } i = \sum_{j=0}^{N-1} b_j 2^j \,, \\ 0 & \text{else} \end{cases} \tag{1.4}$$

with $0 \le i < 2^N$. From left-to-right we therefore write the most-significant to least-significant bits. On occasion, we will write $|i\rangle$ for $|b_{N-1}b_{N-2}\cdots b_0\rangle$ with $i = \sum_{j=0}^{N-1} b_j 2^j$ and use the function $\mathrm{bit}_j(i) = b_j$.

- A quantum gate is a unitary matrix acting on S^N.
- For a general state

$$|\Psi\rangle = \sum_{i=0}^{2^N-1} \Psi_i |i\rangle \tag{1.5}$$

with $\Psi_i \in \mathbb{C}$ and $\sum_{i=0}^{2^N-1} |\Psi_i|^2 = 1$ we define

$$P_j\left(|\Psi\rangle\right) = \sum_{i=0;\mathrm{bit}_j(i)=1}^{2^N-1} |\Psi_i|^2 \tag{1.6}$$

and a probabilistic measurement operation $M_j : S^N \to S^N \otimes \{0,1\}$ that with likelihood $P_j(|\Psi\rangle)$ maps $|\Psi\rangle$ to

$$\sum_{i=0;\mathrm{bit}_j(i)=1}^{2^N-1} \frac{\Psi_i}{\sqrt{P_j\left(|\Psi\rangle\right)}} |i\rangle \otimes 1 \tag{1.7}$$

or else to

$$\sum_{i=0;\mathrm{bit}_j(i)=0}^{2^N-1} \frac{\Psi_i}{\sqrt{1-P_j\left(|\Psi\rangle\right)}} |i\rangle \otimes 0\,. \tag{1.8}$$

This operation therefore maps a qubit to a qubit and a classical bit in a probabilistic manner.

- The classical bit returned by M_j is insensitive to a global complex phase of the state. This is important since the classical bits are the only way to extract information from the system. Global phases are therefore not important.
- A set of quantum gates that when combined can generate an arbitrary unitary matrix up to an arbitrary global phase is called **universal**.
- A digital quantum computer is a machine that can apply quantum gates on N-qubit states and perform such measurements that provide a mapping to classical bits. Such a machine is called **universal** if it can apply a universal set of quantum gates.

1.1.2 Common one-qubit gates

- Hadamard

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{1.9}$$

Action: $|0\rangle \to \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$, $|1\rangle \to \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$

- Phase shift gate

$$R_\phi = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{pmatrix} . \tag{1.10}$$

Action: $|0\rangle \to |0\rangle$, $|1\rangle \to e^{i\phi}|1\rangle$

- Pauli-X or NOT gate

$$X = \text{NOT} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{1.11}$$

Action: $|0\rangle \to |1\rangle$, $|1\rangle \to |0\rangle$

- Pauli-Y

$$Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \tag{1.12}$$

Action: $|0\rangle \to i|1\rangle$, $|1\rangle \to -i|0\rangle$

- Pauli-Z

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} = R_\pi \tag{1.13}$$

- $T = R_{\pi/4}$, $S = R_{\pi/2}$

1.1.3 Common two-qubit gates

- Swap

$$\text{SWAP} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{1.14}$$

Action: $|00\rangle \to |00\rangle$, $|01\rangle \to |10\rangle$, $|10\rangle \to |01\rangle$, $|11\rangle \to |11\rangle$

- CNOT

$$\text{CNOT} = cX = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{1.15}$$

Action: $|00\rangle \to |00\rangle$, $|01\rangle \to |01\rangle$, $|10\rangle \to |11\rangle$, $|11\rangle \to |10\rangle$;
Here: least-significant bit is target bit, most-significant bit is control bit
Version for switched target and control bits in exercises.

1.1.4 Quantum circuits

- A quantum circuit is a specific combination of gates and measurements. We introduce a graphical representation.
- A qubit is represented by a single line

$$|\psi\rangle \; \text{———}$$

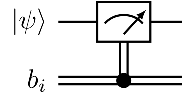
- N qubits may be grouped and written as

$$|\psi\rangle \; \text{—}\!/^{N}\text{—}$$

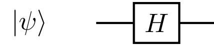
- A classical bit is represented by a double line

$$b_i \quad =$$

- The measurement of a qubit is represented as



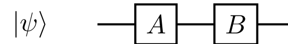
- A one-qubit gate operation is represented as



  for the Hadamard gate and analog for other gates.
- The NOT gate has the special representation

$$|\psi\rangle \quad -\!\oplus\!-$$

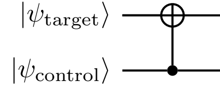
- Gate applications are read from left to right, so



  is represented as

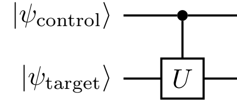
$$BA|\psi\rangle \tag{1.16}$$

  in matrix notation.
- CNOT with least-significant target bit is represented as



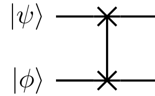
One typically writes the least-significant qubit at the top. The case with least-significant control bit will be discussed in the exercises.

- A general controlled one-qubit U operation is represented as



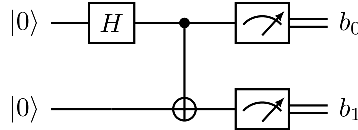
where U could, e.g., be H or R_ϕ, which is only applied to target bit if control bit is 1.

- SWAP is represented as



1.1.5 Example: Bell states

The circuit



performs the Bell experiment. After the Hadamard the system is in state

$$\frac{1}{\sqrt{2}}\left(|00\rangle + |01\rangle\right) \tag{1.17}$$

and then after the CNOT in

$$\frac{1}{\sqrt{2}}\left(|00\rangle + |11\rangle\right) . \tag{1.18}$$

The four possible classical results then are measured with probability

$$P(00) = P(11) = \frac{1}{2}\,, \qquad\qquad P(10) = P(01) = 0\,. \tag{1.19}$$

The bits are maximally correlated and we always find $b_0 = b_1$. Such correlation between qubits is called entanglement and is at the core of many quantum algorithms.

1.1.6 Quantum simulator

- A digital quantum computer can be simulated on a classical computer.
- On the course webpage, there is a link to http://github.com/lehner/sqc, a simple Python implementation of such a simulator; In this week's exercises, we will discuss the usage of this library and we will use it to implement some of the algorithms introduced in the lecture.
- Caveat: with increasing number of qubits N, the problem size for the classical simulation increases as 2^N.

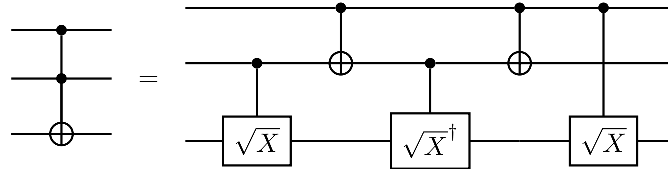
1.1.7 Many-qubit gates

- It is useful to define

$$\sqrt{X} = HR_{\pi/2}H = \frac{1}{2}\begin{pmatrix} 1+i & 1-i \\ 1-i & 1+i \end{pmatrix} \tag{1.20}$$

  with $\sqrt{X}\sqrt{X} = X$ =NOT.

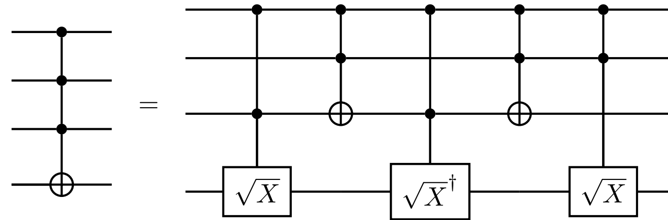
- Extend the CNOT gate by another control gate to the C^{2}NOT or Toffoli gate. It can be implemented as



- For controlled-U gate, replace $\sqrt{X}$ by $\sqrt{U}$; see also problem set 2. For this, we will need

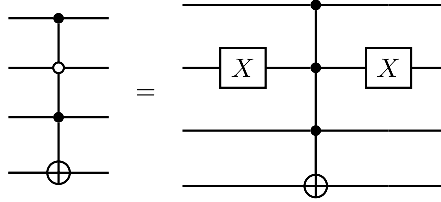
$$X^{1/n} = HR_{\pi/n}H\,. \tag{1.21}$$

- Similarly the C^{n}NOT can be obtained from the C^{n-1}NOT gate. For C^{3}NOT



  With worker bits, we can implement this more efficiently, see problem set 2.

- We can also trigger on a 0 bit value by adding additional NOT gates. We write this with an empty circle



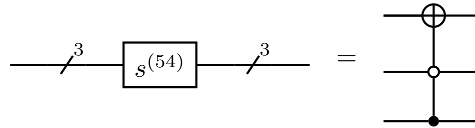
- Now we are ready to find a representation of a gate $s^{(ij)}$ on S^N with $s^{(ij)}|i\rangle = |j\rangle$, $s^{(ij)}|j\rangle = |i\rangle$, $s^{(ij)}|l\rangle = |l\rangle$ for $0 \leq i, j, l < 2^N$ and $l \neq i, j$. This matrix plays a crucial role in the proof of universality done in the exercises

- Consider the bit-wise representation of i and j. Let i and j differ in n bits, then we can define so-called Gray codes g_m, where $g_0 = i$, $g_n = j$, and g_m and g_{m-1} only differ in one bit.

  Example: $i = 3$ (011), $j = 4$ (100); $g_0 = 3$ (011), $g_1 = 7$ (111), $g_2 = 5$ (101), $g_3 = 4$ (100)

- We can implement $s^{(g_m, g_{m-1})}$ by the C^{N-1}NOT gate with target bit being the differing bit and triggering on the values of the common bits.

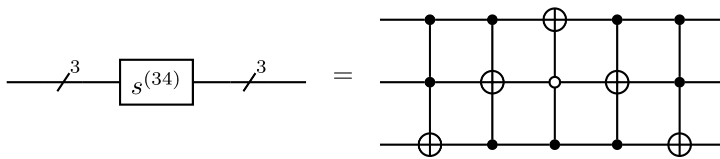
  Example: $g_2 = 5$ and $g_3 = 4$ from above



- Then

$$s^{(ij)} = s^{(g_1, g_0)} \dots s^{(g_{n-1}, g_{n-2})} s^{(g_n, g_{n-1})} \dots s^{(g_1, g_0)} \tag{1.22}$$

  Example: $i = 3$, $j = 4$



1.1.8 Quantum parallelism

- For a general function

  $$f : \{0,1\}^N \to \{0,1\} \tag{1.23}$$

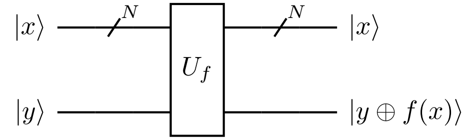
  it will be useful to define

  $$U_f : S^N \otimes S \to S^N \otimes S, |x\rangle \otimes |y\rangle \mapsto |x\rangle \otimes |y \oplus f(x)\rangle\,, \tag{1.24}$$

  where $\oplus$ here means the modulo 2 operation. Such a function effectively evaluates f for all possible values in parallel. We will also write $|x\rangle \otimes |y\rangle = |x,y\rangle$. Note that the action on a general element of $S^N \otimes S$ is defined by the action on each basis element $|x,y\rangle$.

- Since we cannot implement a non-reversible gate operation (quantum gates are unitary), we cannot map simply to $|x\rangle \otimes |f(x)\rangle$. The simple reversible choice of $y \oplus f(x)$ will shown to be useful below.

- The circuit representation is



- For $N = 1$, the state

  $$|\Psi\rangle = \Psi_{0,0}|0,0\rangle + \Psi_{0,1}|0,1\rangle + \Psi_{1,0}|1,0\rangle + \Psi_{1,1}|1,1\rangle \tag{1.25}$$

  then would be mapped to

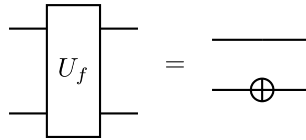
  $$\begin{aligned} U_f\,|\Psi\rangle = \Psi_{0,0}|0, 0 \oplus f(0)\rangle + \Psi_{0,1}|0, 1 \oplus f(0)\rangle \\ + \Psi_{1,0}|1, 0 \oplus f(1)\rangle + \Psi_{1,1}|1, 1 \oplus f(1)\rangle\,. \end{aligned} \tag{1.26}$$

- Example: $N = 1$, $f(x) = 1$

  We have

  $$U_f\,|\Psi\rangle = \Psi_{0,0}|0,1\rangle + \Psi_{0,1}|0,0\rangle + \Psi_{1,0}|1,1\rangle + \Psi_{1,1}|1,0\rangle\,. \tag{1.27}$$

  or

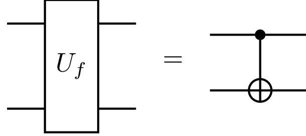


- Example: $N = 1,\ f(x) = x$

  We have

$$U_f\,|\Psi\rangle = \Psi_{0,0}|0,0\rangle + \Psi_{0,1}|0,1\rangle + \Psi_{1,0}|1,1\rangle + \Psi_{1,1}|1,0\rangle\,. \tag{1.28}$$

  or



1.2 Quantum algorithms

1.2.1 Deutsch–Jozsa algorithm

- A simple example for a problem that can be solved exponentially faster on a quantum computer than on a classical computer

- Problem: Given a function

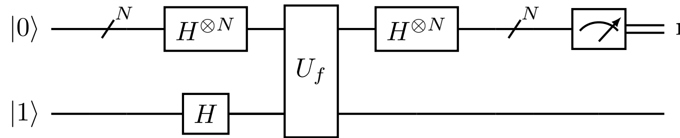
$$f : \{0,1\}^N \to \{0,1\} \tag{1.29}$$

  that is either constant (maps always to either 0 or 1) or balanced (returns 0 for exactly half of the input domain and 1 else), find out if it is constant or balanced.

- Classical solution: evaluate function for all possible inputs and stop as soon as answer is known

  - Best case: first two evaluations differ $\Rightarrow$ function is balanced
  - Worst case: first 2^{N-1} evaluations are identical $\Rightarrow$ answer known after $2^{N-1}+1$ evaluations
  - For randomized selection of inputs: probability of misidentification after n evaluations is $\leq \frac{1}{2^n}$

- Quantum algorithm (guaranteed result after one evaluation):



  The measured $r = 0$ if and only if f is constant. If f is balanced $r \neq 0$ will be measured.

- Explanation:

  We start with the input state (Mark states in diagram above)

$$\Psi_0 = |0\rangle^{\otimes N}|1\rangle \,. \tag{1.30}$$

  After the Hadamard operations, we have

$$\Psi_1 = \sum_{x=0}^{2^N-1} \frac{1}{\sqrt{2^{N+1}}}|x\rangle\big(|0\rangle - |1\rangle\big) \,. \tag{1.31}$$

  To illustrate the mechanism, we remind ourself that for $N = 2$

$$(H \otimes H)|00\rangle = (H \otimes \mathbf{1})\frac{1}{\sqrt{2}}\big(|00\rangle + |01\rangle\big) \,, \tag{1.32}$$

$$= \frac{1}{2}\big(|00\rangle + |10\rangle + |01\rangle + |11\rangle\big) \tag{1.33}$$

  and

$$H|1\rangle = \frac{1}{\sqrt{2}}\big(|0\rangle - |1\rangle\big) \,. \tag{1.34}$$

  Then U_f produces

$$\Psi_2 = \sum_{x=0}^{2^N-1} \frac{1}{\sqrt{2^{N+1}}}|x\rangle\big(|0 \oplus f(x)\rangle - |1 \oplus f(x)\rangle\big) \tag{1.35}$$

$$= \sum_{x=0}^{2^N-1} \frac{(-1)^{f(x)}}{\sqrt{2^{N+1}}}|x\rangle\big(|0\rangle - |1\rangle\big) \,. \tag{1.36}$$

  Next we note

$$H|b\rangle = \frac{1}{\sqrt{2}}\sum_{b'=0}^{1}(-1)^{bb'}|b'\rangle \tag{1.37}$$

  with $b \in \{0, 1\}$ or generally

$$H^{\otimes N}|x\rangle = \frac{1}{\sqrt{2^N}}\sum_{z=0}^{2^N-1}(-1)^{x\cdot z}|z\rangle \,, \tag{1.38}$$

  where

$$x \cdot z \equiv \left(\sum_{j=0}^{N-1} \mathrm{bit}_j(x)\mathrm{bit}_j(z)\right) \bmod 2 \tag{1.39}$$

  and $\mathrm{bit}_j(x)$ is the value of the j-th bit of x.

Therefore the final state is

$$\Psi_3 = \sum_{x,z=0}^{2^N-1} \frac{(-1)^{z\cdot x+f(x)}}{2^N}|z\rangle \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right) . \tag{1.40}$$

Now if $f(x)$ is constant, the amplitude of the $z=0$ state is

$$\left|\sum_{x=0}^{2^N-1} \frac{(-1)^{f(x)}}{2^N}\right|^2 = 1 \tag{1.41}$$

and therefore with certainty $r=0$ will be measured. If $f(x)$ is balanced the $z=0$ amplitude is

$$\left|\sum_{x=0}^{2^N-1} \frac{(-1)^{f(x)}}{2^N}\right|^2 = 0 \tag{1.42}$$

and due to the normalization of the state a value $r \neq 0$ will be measured. So with a single application of the quantum circuit, we know the answer.

- Run algorithm on sqc

In [1]:
```
import sqc
```

In [4]:
```
Nbits=2

def Uf_const(o): # N=1, f(x)=1
    return o.NOT(1)

def Uf_balanced(o): # N=1, f(x)=x
    return o.CNOT(0,1)
```

In [5]:
```
for Uf,n in [ (Uf_const,"f(x)=1"), (Uf_balanced,"f(x)=x") ]:
    print("Run Deutsch-Jozsa for %s" % n)
    print("-------------------------------------")

    # Psi0
    psi0=sqc.operator(Nbits).X(1) * sqc.state(Nbits, basis=[ "|%d>|%d>" % (i%2,i//2) for i in rang
e(4) ])
    print("|Psi0> = ")
    print(psi0)
    print("")

    # Psi1
    psi1=sqc.operator(Nbits).H(0).H(1) * psi0
    print("|Psi1> = ")
    print(psi1)
    print("")

    # Psi2
    psi2=Uf(sqc.operator(Nbits)) * psi1
    print("|Psi2> = ")
    print(psi2)
    print("")

    # Psi3
    psi3=sqc.operator(Nbits).H(0) * psi2
    print("|Psi3> = ")
    print(psi3)
    print("")

    # Measurement
    psi4,r=psi3.measure(0)
    print("Result: %d, %s is %s" % (r,n,["constant","balanced"][r]))

    print("")
    print("")
```

1.2.2 Grover algorithm

- Problem:

  Given a function

  $$f : \{0, 1\}^N \rightarrow \{0, 1\} \tag{1.43}$$

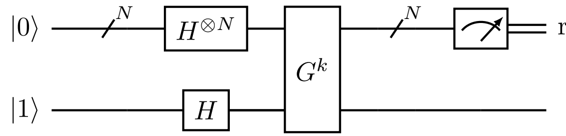
  that is 1 for one value of x_0 with $0 \leq x_0 < 2^N$ and 0 for all other values, find x_0.

- Applications: Database search, cryptographic hash functions, factor $n = p * q$ with p and q prime (classical test division up to $\sqrt{n}$)

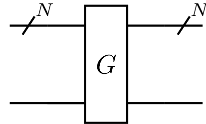
- Classical solution requires $O(2^N)$ evaluations of the function

- Here, we illustrate a quantum algorithm which requires $O(\sqrt{2^N})$ function evaluations

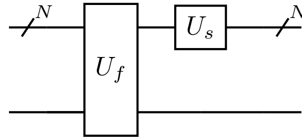
- The quantum circuit is:



with



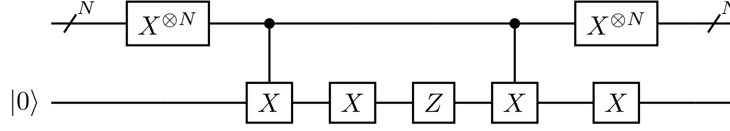
=



and

$$\boxed{U_s} = \boxed{H^{\otimes N}} — \boxed{2\left|0\right\rangle\left\langle 0\right| - 1} — \boxed{H^{\otimes N}} .$$

The unitary operation $2\left|0\right\rangle\left\langle 0\right| - 1$ acting on a N qubit subspace can be implemented using a single ancilla qubit $\left|0\right\rangle$:



- For $k \approx \sqrt{2^N}$, we measure $r = x_0$ with high likelihood $1 - O(1/2^N)$
- Explanation:

  We start with the input state (Mark states in diagram above)

$$\Psi_0 = |0\rangle^{\otimes N} |1\rangle \, . \tag{1.44}$$

After the Hadamard operations, we have

$$\Psi_1 = \sum_{x=0}^{2^N-1} \frac{1}{\sqrt{2^{N+1}}} |x\rangle \big(|0\rangle - |1\rangle\big) \, . \tag{1.45}$$

Neglecting the trivial last qubit, we can say that at this point the system is in state

$$|\psi\rangle \equiv |\psi_1\rangle = \frac{1}{\sqrt{2^N}} \sum_{x=0}^{2^N-1} |x\rangle = \sqrt{\frac{2^N - 1}{2^N}} \, |\alpha\rangle + \frac{1}{\sqrt{2^N}} \, |\beta\rangle \tag{1.46}$$

with useful basis

$$|\alpha\rangle \equiv \frac{1}{\sqrt{2^N - 1}} \sum_{x \neq x_0} |x\rangle \ , \tag{1.47}$$

$$|\beta\rangle \equiv |x_0\rangle \ . \tag{1.48}$$

We will find that the remaining operations live in the space spanned by $|\alpha\rangle$ and $|\beta\rangle$.

It is easy to verify that $U_f : |x\rangle \, |y\rangle \mapsto |x\rangle \, |y \oplus f(x)\rangle$ performs a reflection about $|\alpha\rangle$ in this space, i.e.,

$$U_f(a \, |\alpha\rangle + b \, |\beta\rangle) = a \, |\alpha\rangle - b \, |\beta\rangle \ . \tag{1.49}$$

Similarly

$$U_s = H^{\otimes N}(2 \, |0\rangle \, \langle 0| - 1) H^{\otimes N} = 2 \, |\psi\rangle \, \langle \psi| - 1 \tag{1.50}$$

i.e., U_s is a reflection about the vector $|\psi\rangle$.

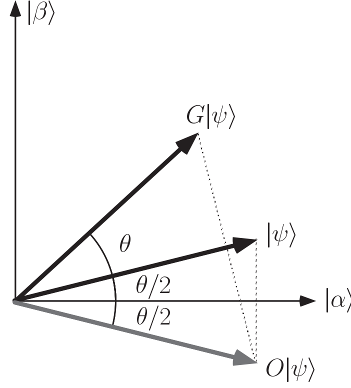
Combined G therefore performs a rotation in this space by

$$\cos(\theta/2) = \sqrt{\frac{2^N - 1}{2^N}} \tag{1.51}$$

or

$$\theta = \frac{2}{\sqrt{2^N}}\Big(1 + O(1/2^N)\Big)\,. \tag{1.52}$$

Illustrate:



After k applications

$$|\psi_{1+k}\rangle = G^k\,|\psi_1\rangle = \cos(\theta_k/2)\,|\alpha\rangle + \sin(\theta_k/2)\,|\beta\rangle\ . \tag{1.53}$$

We find

$$\theta_k = (1+2k)\theta\,. \tag{1.54}$$

For $\theta_k \approx \pi$ we have the optimal solution or

$$k \approx (\pi/4)\sqrt{2^N}\,. \tag{1.55}$$

1.2.3 Parenthesis: Arithmetic gates, ancilla and garbage qubits, uncomputing

- We now address how to implement generic arithmetic functions $f(x)$ with

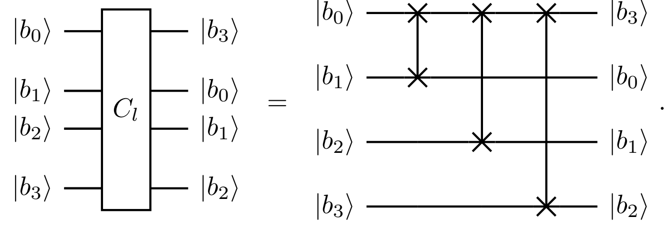
$$x = \sum_{j=0}^{N-1} b_j 2^j\,, \tag{1.56}$$

where $b_j \in \{0,1\}$.

- A circular left shift C_l with

$$C_l\,|b_{N-1},\ldots,b_0\rangle = |b_{N-2},\ldots,b_0,b_{N-1}\rangle \tag{1.57}$$

can be implemented with SWAP gates. Example $N=4$:



If $b_{N-1} = 0$, this implements

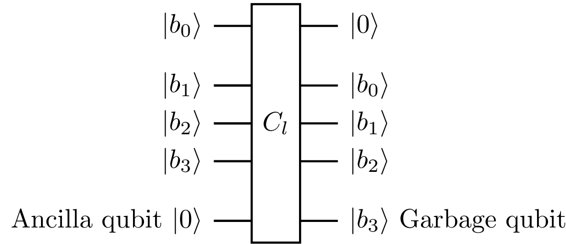
$$f(x) = 2x\,. \tag{1.58}$$

If $b_{N-1} = 1$, the circular left shift results in an odd number, however, which we may not wish for an implementation of $f(x) = 2x$.

- A left shift

$$S_l\,|b_{N-1},\ldots,b_0\rangle = |b_{N-2},\ldots,b_0,0\rangle \tag{1.59}$$

can be implemented if we add a so-called **ancilla** qubit with fixed value $|0\rangle$ to the circuit, resulting in an additional variable **garbage qubit**. We again illustrate for $N = 4$:

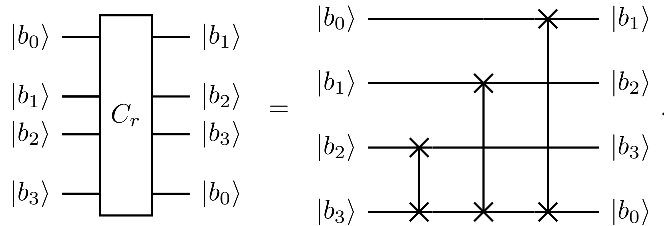


This operation again coincides with $f(x) = 2x$, however, has a more gracious overflow behavior (known from, e.g., C code) for $b_{N-1} = 1$.

- A circular right shift C_r with

$$C_r\,|b_{N-1},\ldots,b_0\rangle = |b_0,b_{N-1},\ldots,b_1\rangle \tag{1.60}$$

can again be implemented with SWAP gates. Example $N = 4$:



If $b_0 = 0$, this implements

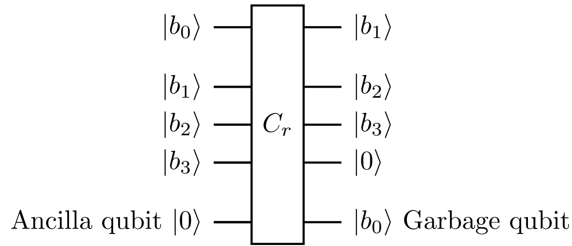
$$f(x) = x/2\,. \tag{1.61}$$

However for $b_0 = 1$ (for odd numbers), we do not obtain a sensible result for division by two.

- A right shift

$$S_r\,|b_{N-1}, \ldots, b_0\rangle = |0, b_{N-1}, \ldots, b_1\rangle \tag{1.62}$$

can again be implemented with ancilla and garbage qubits. We again illustrate for $N = 4$:



This operation coincides with $f(x) = x/2$ for all x.

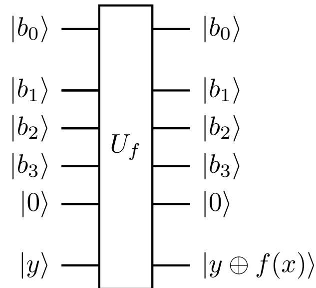
- The need for ancilla and garbage qubits originates from the restriction to use reversible (unitary) gates on a quantum computer.

- Since the value of a garbage qubit is variable, we cannot reuse it as an ancilla bit for another gate. We can, however, use intermediate information and then **uncompute** it to restore the ancilla bit.

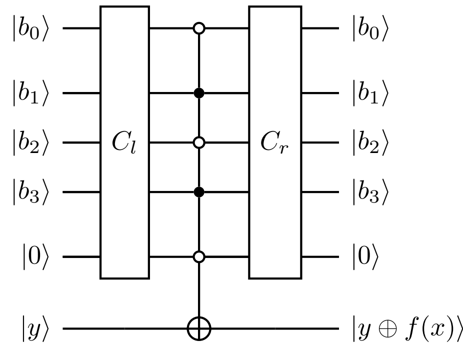
  Example: Implement U_f for

$$f(x) = \begin{cases} 1 & \text{for } 2x = 10\,, \\ 0 & \text{else }\;. \end{cases} \tag{1.63}$$

For $N = 4$, this can be implemented as



=



,

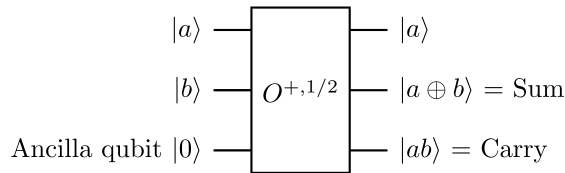
where the C^4NOT operation only acts for $x = 10$ (binary 1010). The control on the ancilla bit checks for an integer overflow.

This "uncomputation" now allows for multiple applications of this U_f with only a single overall ancilla qubit.

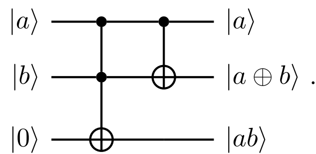
- Next, we consider the function

$$f(x) = x + 1\,. \tag{1.64}$$

This can be implemented using elementary "half-adder" gates

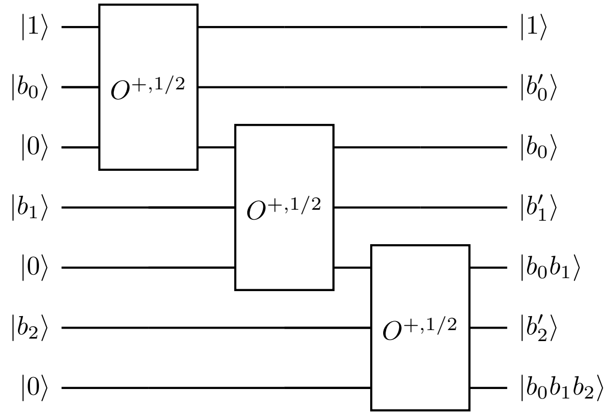


=



.

We combine half-adders to implement the function. For N=3, e.g.,



with

$$x + 1 = \sum_{j=0}^{N-1} b'_j 2^j \, . \tag{1.65}$$

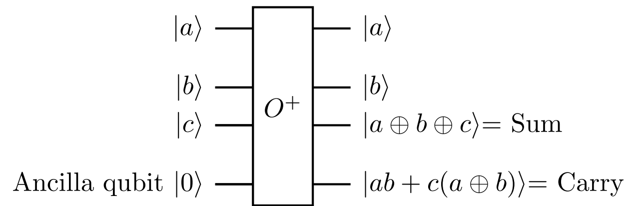
- We can also add more generally

$$f(x) = x + z \tag{1.66}$$

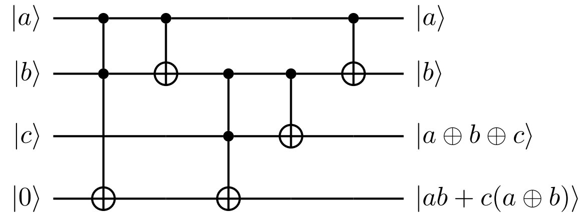
with

$$z = \sum_{j=0}^{N-1} c_j 2^j \, . \tag{1.67}$$

For this we need to define a full-adder gate

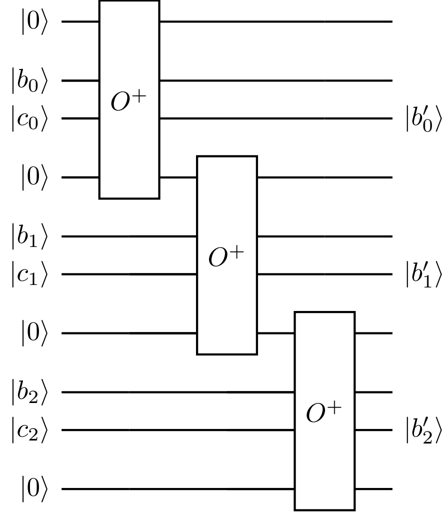


=



.

We combine full-adders to implement the function. For N=3, e.g.,



with

$$x + z = \sum_{j=0}^{N-1} b'_j 2^j \,, \tag{1.68}$$

where we do not write the values of the garbage qubits.

- Similarly, we can implement subtraction.
- Finally, we consider multiplication of x and z, i.e.,

$$f(x) = xz = \sum_{j,j'=0}^{N-1} b_j c_{j'} 2^{j+j'} = \sum_{j=0}^{N-1} b_j S_l^j(z) \,, \tag{1.69}$$

where $S_l^j(z)$ implements j left-shifts of the number z as defined above. We note that we can implement binary multiplication as a combination of such shifts and addition.

1.2.4 Example: Solving an equation using Grover's algorithm

In [1]:
```
import sqc
import numpy as np
import matplotlib.pyplot as plt
```

Control-U gates (see also Problem Set 2, Problem 1)

In [2]:
```
def A(i,o):
    return o.Rz(i,np.pi/4.0).H(i).Rz(i,np.pi/2.0)

def Adg(i,o):
    return o.Rz(i,-np.pi/2.0).H(i).Rz(i,-np.pi/4.0)

def CH(i,t,o): # i=control, t=target
    return A(t,Adg(t,o).CNOT(i,t))

def CRz(i,t,phi,o):
    return o.Rz(t,phi/2.).CNOT(i,t).Rz(t,-phi/2.).Rz(i,phi/2.).CNOT(i,t)

op=sqc.operator(2)

for i in range(4):
    st=sqc.state(2,v = [ 1 if i == j else 0 for j in range(4) ])

    print("Applying CH(0,1) to %s gives:\n%s\n" % (str(st),str(CH(0,1,op)*st)))
    print("Applying CRz(0,1,Pi/2) to %s gives:\n%s\n" % (str(st),str(CRz(0,1,np.pi/2.,op)*st)))
```

```
Applying CH(0,1) to    1 * |00> gives:
   1 * |00>

Applying CRz(0,1,Pi/2) to    1 * |00> gives:
   1 * |00>

Applying CH(0,1) to    1 * |01> gives:
   0.707107 * |01>
 + 0.707107 * |11>

Applying CRz(0,1,Pi/2) to    1 * |01> gives:
   1 * |01>

Applying CH(0,1) to    1 * |10> gives:
   1 * |10>

Applying CRz(0,1,Pi/2) to    1 * |10> gives:
   1 * |10>

Applying CH(0,1) to    1 * |11> gives:
   0.707107    * |01>
 + (-0.707107) * |11>

Applying CRz(0,1,Pi/2) to    1 * |11> gives:
   1j * |11>
```

$C^2X^{1/n}$ gate (see also Lecture Section 1.1.7)

$X^{1/n} = HR_{\pi/n}H$

In [3]:
```
def CrootX(i,t,n,o):
    return CH(i,t,CRz(i,t,np.pi/n,CH(i,t,o)))

def CNOT(i,j,o):
    return o.CNOT(i,j)

def C2rootX(i,j,t,n,o): #i,j=control; t=target
    return CrootX(i,t,2.*n,CNOT(i,j,CrootX(j,t,-2.*n,CNOT(i,j,CrootX(j,t,2.*n,o)))))

op=sqc.operator(3)
for i in range(8):
    st=sqc.state(3,v = [ 1 if i == j else 0 for j in range(8) ])

    print("Applying C2NOT(0,1,2) to %s gives:\n%s\n" % (str(st),str(C2rootX(0,1,2,1,op)*st)))
```

```
Applying C2NOT(0,1,2) to    1 * |000> gives:
   1 * |000>

Applying C2NOT(0,1,2) to    1 * |001> gives:
   1 * |001>

Applying C2NOT(0,1,2) to    1 * |010> gives:
   1 * |010>

Applying C2NOT(0,1,2) to    1 * |011> gives:
   1 * |111>

Applying C2NOT(0,1,2) to    1 * |100> gives:
   1 * |100>

Applying C2NOT(0,1,2) to    1 * |101> gives:
   1 * |101>

Applying C2NOT(0,1,2) to    1 * |110> gives:
   1 * |110>

Applying C2NOT(0,1,2) to    1 * |111> gives:
   1 * |011>
```

$C^nX^{1/m}$ (see also Lecture Section 1.1.7)

In Problem Set 02, we will learn a much faster implementation, that requires additional worker qubits.

In [4]:
```
def CnrootX(c,t,m,o): #c=array of n control qubits; t=target qubit
    assert(len(c)>0)
    if len(c) == 1:
        return CrootX(c[0],t,m,o)
    i=[ c[0] ]
    j=[ c[1] ]
    r=c[2:]
    return CnrootX(r + i,t,2.*m,CnrootX(r+i,j[0],1.,CnrootX(r+j,t,-2.*m,CnrootX(r+i,j[0],1.,CnrootX(r+j,t,2.*m,o)))))

op=sqc.operator(4)
for i in range(16):
    st=sqc.state(4,v = [ 1 if i == j else 0 for j in range(16) ])

    print("Applying CnNOT([0,1,2],3) to %s gives:\n%s\n" % (str(st),str(CnrootX([0,1,2],3,1,op)*st)))
```

```
Applying CnNOT([0,1,2],3) to    1 * |0000> gives:
   1 * |0000>

Applying CnNOT([0,1,2],3) to    1 * |0001> gives:
   1 * |0001>

Applying CnNOT([0,1,2],3) to    1 * |0010> gives:
   1 * |0010>

Applying CnNOT([0,1,2],3) to    1 * |0011> gives:
   1 * |0011>

Applying CnNOT([0,1,2],3) to    1 * |0100> gives:
   1 * |0100>

Applying CnNOT([0,1,2],3) to    1 * |0101> gives:
   1 * |0101>

Applying CnNOT([0,1,2],3) to    1 * |0110> gives:
   1 * |0110>

Applying CnNOT([0,1,2],3) to    1 * |0111> gives:
   1 * |1111>

Applying CnNOT([0,1,2],3) to    1 * |1000> gives:
   1 * |1000>

Applying CnNOT([0,1,2],3) to    1 * |1001> gives:
   1 * |1001>

Applying CnNOT([0,1,2],3) to    1 * |1010> gives:
   1 * |1010>

Applying CnNOT([0,1,2],3) to    1 * |1011> gives:
   1 * |1011>

Applying CnNOT([0,1,2],3) to    1 * |1100> gives:
   1 * |1100>

Applying CnNOT([0,1,2],3) to    1 * |1101> gives:
   1 * |1101>

Applying CnNOT([0,1,2],3) to    1 * |1110> gives:
   1 * |1110>

Applying CnNOT([0,1,2],3) to    1 * |1111> gives:
   1 * |0111>
```

Circular shift

In [5]:
```
def SWAP(i,j,o):
    return o.CNOT(i,j).CNOT(j,i).CNOT(i,j)

def Cl(mask,o):
```

```
    for i in mask[1:]:
        o=SWAP(mask[0],i,o)
    return o

def Cr(mask,o):
    for i in reversed(mask[:-1]):
        o=SWAP(mask[-1],i,o)
    return o

op=sqc.operator(4)
st=sqc.state(4,v=[0,1] + [0]*14)
for i in range(4):
    print("Cl^%d |0001> = %s" % (i,str(st)))
    st=Cl([0,1,2],op)*st

st=sqc.state(4,v=[0,1] + [0]*14)
for i in range(4):
    print("Cr^%d |0001> = %s" % (i,str(st)))
    st=Cr([0,1,2],op)*st
```

```
Cl^0 |0001> =    1 * |0001>
Cl^1 |0001> =    1 * |0010>
Cl^2 |0001> =    1 * |0100>
Cl^3 |0001> =    1 * |0001>
Cr^0 |0001> =    1 * |0001>
Cr^1 |0001> =    1 * |0100>
Cr^2 |0001> =    1 * |0010>
Cr^3 |0001> =    1 * |0001>
```

Grover search for 2x=10 (see Lecture Section 1.2.2)

In [8]:

```
nqubits=4
expected_theta=2.*np.arccos(np.sqrt((2**nqubits - 1.)/(2**nqubits)))
optimal_k = int(np.round(np.pi / 4. * np.sqrt(2**nqubits)))

print("Optimal_k = %d" % optimal_k)

plt.figure(figsize=(6,6))
plt.xlim(-1,1)
plt.ylim(-1,1)
plt.quiver([0,0],[0,0],[0,1],[1,0],   color=['gray'], scale=3)

plt.text(0.75,0,"|alpha>")

plt.text(0,0.75,"|beta>")

# TODO: theta_k
plt.quiver([0]*(optimal_k+1),[0]*(optimal_k+1),[np.cos(expected_theta*(i+0.5)) for i in range(optimal_k+1)],
        [np.sin(expected_theta*(i+0.5)) for i in range(optimal_k+1)],   color=['r','b','g','black'], scale=4)
plt.show()

expected_theta_deg=expected_theta / np.pi * 180.
print("Optimal_angle = %g" % ((optimal_k+0.5)*expected_theta_deg))
```

```
Optimal_k = 3
```

|beta>
|alpha>
1.00 0.75 0.50 0.25 0.00 −0.25 −0.50 −0.75 −1.00
−1.00 −0.75 −0.50 −0.25 0.00 0.25 0.50 0.75 1.00

```
Optimal_angle = 101.343
```

In [9]:

```
print("Startup Grover Search")
print("---------------------------------------")

Nbits=nqubits+2 # bits for x, 1 ancilla bit, 1 for y

def pr(i,start,width):
    return (i//2**start)%(2**width)

# Psi0
psi0=sqc.operator(Nbits).X(nqubits+1) * sqc.state(Nbits,
            basis=[ "|%d>|%d>|%d>" % (pr(i,0,nqubits),pr(i,nqubits,1),pr(i,nqubits+1,i)) for i in range(2**Nbits) ])
print("|Psi0> = ")
print(psi0)
print("") # TODO: continue here

# Psi1
psi1=sqc.operator(Nbits).H(0).H(1).H(2).H(3).H(5) * psi0
print("|Psi1> = ")
print(psi1)
print("")
```

```
Startup Grover Search
---------------------------------------
|Psi0> = 
   1 * |0>|0>|1>

|Psi1> = 
   0.176777    * |0>|0>|0>
 + 0.176777    * |1>|0>|0>
 + 0.176777    * |2>|0>|0>
 + 0.176777    * |3>|0>|0>
 + 0.176777    * |4>|0>|0>
 + 0.176777    * |5>|0>|0>
 + 0.176777    * |6>|0>|0>
 + 0.176777    * |7>|0>|0>
 + 0.176777    * |8>|0>|0>
 + 0.176777    * |9>|0>|0>
 + 0.176777    * |10>|0>|0>
 + 0.176777    * |11>|0>|0>
 + 0.176777    * |12>|0>|0>
 + 0.176777    * |13>|0>|0>
 + 0.176777    * |14>|0>|0>
 + 0.176777    * |15>|0>|0>
 + (-0.176777) * |0>|0>|1>
 + (-0.176777) * |1>|0>|1>
 + (-0.176777) * |2>|0>|1>
 + (-0.176777) * |3>|0>|1>
 + (-0.176777) * |4>|0>|1>
 + (-0.176777) * |5>|0>|1>
 + (-0.176777) * |6>|0>|1>
 + (-0.176777) * |7>|0>|1>
 + (-0.176777) * |8>|0>|1>
 + (-0.176777) * |9>|0>|1>
 + (-0.176777) * |10>|0>|1>
 + (-0.176777) * |11>|0>|1>
 + (-0.176777) * |12>|0>|1>
 + (-0.176777) * |13>|0>|1>
 + (-0.176777) * |14>|0>|1>
 + (-0.176777) * |15>|0>|1>
```

In [11]:

```
def Uf(o): # additional 4 bit in CnrootX is to check for overflow in f(x)=2x
    return Cr([0,1,2,3,4],CnrootX([0,1,2,3,4],5,1.,Cl([0,1,2,3,4],o).X(0).X(2).X(4)).X(0).X(2).X(4))

def Usdiag(o): # 2|0><0| - sum_i |i><i|, draw circuit?
    print("Usdiag applied to")
    return CnrootX([0,1,2,3],4,1.,CnrootX([0,1,2,3],4,1.,o.X(0).X(1).X(2).X(3)).X(4).Rz(4,np.pi)).X(0).X(1).X(2).X(3).X(4)

uf=Uf(sqc.operator(Nbits))
print("Uf |psi1> = ")
print(uf * psi1)
print("")

print("Usdiag |psi1> = ")
print(Usdiag(sqc.operator(Nbits)) * psi1)
print("")

us=Usdiag(sqc.operator(Nbits).H(0).H(1).H(2).H(3)).H(0).H(1).H(2).H(3)
print("Us |psi1> = ")
print(us * psi1)
print("")
```

```
Uf |psi1> = 
   0.176777    * |0>|0>|0>
 + 0.176777    * |1>|0>|0>
 + 0.176777    * |2>|0>|0>
 + 0.176777    * |3>|0>|0>
 + 0.176777    * |4>|0>|0>
 + (-0.176777) * |5>|0>|0>
 + 0.176777    * |6>|0>|0>
 + 0.176777    * |7>|0>|0>
 + 0.176777    * |8>|0>|0>
 + 0.176777    * |9>|0>|0>
 + 0.176777    * |10>|0>|0>
 + 0.176777    * |11>|0>|0>
 + 0.176777    * |12>|0>|0>
 + 0.176777    * |13>|0>|0>
 + 0.176777    * |14>|0>|0>
 + 0.176777    * |15>|0>|0>
 + (-0.176777) * |0>|0>|1>
 + (-0.176777) * |1>|0>|1>
 + (-0.176777) * |2>|0>|1>
```

```
 + (-0.176777) * |3>|0>|1>
 + (-0.176777) * |4>|0>|1>
 + 0.176777    * |5>|0>|1>
 + (-0.176777) * |6>|0>|1>
 + (-0.176777) * |7>|0>|1>
 + (-0.176777) * |8>|0>|1>
 + (-0.176777) * |9>|0>|1>
 + (-0.176777) * |10>|0>|1>
 + (-0.176777) * |11>|0>|1>
 + (-0.176777) * |12>|0>|1>
 + (-0.176777) * |13>|0>|1>
 + (-0.176777) * |14>|0>|1>
 + (-0.176777) * |15>|0>|1>

Usdiag |psi1> =
Usdiag applied to
   0.176777    * |0>|0>|0>
 + (-0.176777) * |1>|0>|0>
 + (-0.176777) * |2>|0>|0>
 + (-0.176777) * |3>|0>|0>
 + (-0.176777) * |4>|0>|0>
 + (-0.176777) * |5>|0>|0>
 + (-0.176777) * |6>|0>|0>
 + (-0.176777) * |7>|0>|0>
 + (-0.176777) * |8>|0>|0>
 + (-0.176777) * |9>|0>|0>
 + (-0.176777) * |10>|0>|0>
 + (-0.176777) * |11>|0>|0>
 + (-0.176777) * |12>|0>|0>
 + (-0.176777) * |13>|0>|0>
 + (-0.176777) * |14>|0>|0>
 + (-0.176777) * |15>|0>|0>
 + (-0.176777) * |0>|0>|1>
 + 0.176777    * |1>|0>|1>
 + 0.176777    * |2>|0>|1>
 + 0.176777    * |3>|0>|1>
 + 0.176777    * |4>|0>|1>
 + 0.176777    * |5>|0>|1>
 + 0.176777    * |6>|0>|1>
 + 0.176777    * |7>|0>|1>
 + 0.176777    * |8>|0>|1>
 + 0.176777    * |9>|0>|1>
 + 0.176777    * |10>|0>|1>
 + 0.176777    * |11>|0>|1>
 + 0.176777    * |12>|0>|1>
 + 0.176777    * |13>|0>|1>
 + 0.176777    * |14>|0>|1>
 + 0.176777    * |15>|0>|1>

Usdiag applied to
Us |psi1> =
   0.176777    * |0>|0>|0>
 + 0.176777    * |1>|0>|0>
 + 0.176777    * |2>|0>|0>
 + 0.176777    * |3>|0>|0>
 + 0.176777    * |4>|0>|0>
 + 0.176777    * |5>|0>|0>
 + 0.176777    * |6>|0>|0>
 + 0.176777    * |7>|0>|0>
 + 0.176777    * |8>|0>|0>
 + 0.176777    * |9>|0>|0>
 + 0.176777    * |10>|0>|0>
 + 0.176777    * |11>|0>|0>
 + 0.176777    * |12>|0>|0>
 + 0.176777    * |13>|0>|0>
 + 0.176777    * |14>|0>|0>
 + 0.176777    * |15>|0>|0>
 + (-0.176777) * |0>|0>|1>
 + (-0.176777) * |1>|0>|1>
 + (-0.176777) * |2>|0>|1>
 + (-0.176777) * |3>|0>|1>
 + (-0.176777) * |4>|0>|1>
 + (-0.176777) * |5>|0>|1>
 + (-0.176777) * |6>|0>|1>
 + (-0.176777) * |7>|0>|1>
 + (-0.176777) * |8>|0>|1>
 + (-0.176777) * |9>|0>|1>
 + (-0.176777) * |10>|0>|1>
 + (-0.176777) * |11>|0>|1>
 + (-0.176777) * |12>|0>|1>
 + (-0.176777) * |13>|0>|1>
 + (-0.176777) * |14>|0>|1>
 + (-0.176777) * |15>|0>|1>
```

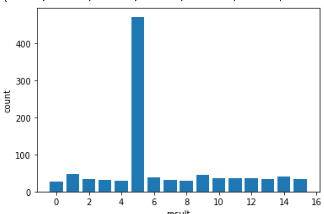
In [12]:

```python
psiA=psi1

def measure(state):
    # Measurement
    res=sqc.sample(psiA, n=1000, mask=[0,1,2,3])
    print(res)
    plt.bar(res.keys(),res.values())
    plt.xlabel('result')
    plt.ylabel('count')
    plt.show()

measure(psiA)

for i in range(optimal_k):
    print("Start Grover Iteration %d" % i)

    # Psi2
    psi2=uf * psiA
    print("|Psi2> = ")
    print(psi2)
    print("")

    # Psi3
    psi3=us * psi2
    print("|Psi3> = ")
    print(psi3)
    print("")

    psiA=psi3

    measure(psiA)
```

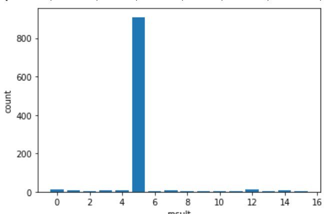
```
{6: 48, 9: 56, 12: 63, 15: 57, 14: 58, 0: 73, 1: 62, 5: 66, 3: 70, 13: 67, 2: 71, 10: 82, 7: 67, 4: 55, 8: 54, 11: 51}
```

```
Start Grover Iteration 0
|Psi2> =
   0.176777    * |0>|0>|0>
 + 0.176777    * |1>|0>|0>
 + 0.176777    * |2>|0>|0>
 + 0.176777    * |3>|0>|0>
 + 0.176777    * |4>|0>|0>
 + (-0.176777) * |5>|0>|0>
 + 0.176777    * |6>|0>|0>
 + 0.176777    * |7>|0>|0>
 + 0.176777    * |8>|0>|0>
 + 0.176777    * |9>|0>|0>
 + 0.176777    * |10>|0>|0>
 + 0.176777    * |11>|0>|0>
 + 0.176777    * |12>|0>|0>
 + 0.176777    * |13>|0>|0>
 + 0.176777    * |14>|0>|0>
 + 0.176777    * |15>|0>|0>
 + (-0.176777) * |0>|0>|1>
 + (-0.176777) * |1>|0>|1>
 + (-0.176777) * |2>|0>|1>
 + (-0.176777) * |3>|0>|1>
 + (-0.176777) * |4>|0>|1>
 + 0.176777    * |5>|0>|1>
 + (-0.176777) * |6>|0>|1>
 + (-0.176777) * |7>|0>|1>
 + (-0.176777) * |8>|0>|1>
 + (-0.176777) * |9>|0>|1>
 + (-0.176777) * |10>|0>|1>
 + (-0.176777) * |11>|0>|1>
 + (-0.176777) * |12>|0>|1>
 + (-0.176777) * |13>|0>|1>
 + (-0.176777) * |14>|0>|1>
 + (-0.176777) * |15>|0>|1>

|Psi3> =
   0.132583    * |0>|0>|0>
 + 0.132583    * |1>|0>|0>
 + 0.132583    * |2>|0>|0>
 + 0.132583    * |3>|0>|0>
 + 0.132583    * |4>|0>|0>
 + 0.486136    * |5>|0>|0>
 + 0.132583    * |6>|0>|0>
 + 0.132583    * |7>|0>|0>
 + 0.132583    * |8>|0>|0>
 + 0.132583    * |9>|0>|0>
 + 0.132583    * |10>|0>|0>
 + 0.132583    * |11>|0>|0>
 + 0.132583    * |12>|0>|0>
 + 0.132583    * |13>|0>|0>
```

```
 + 0.132583    * |14>|0>|0>
 + 0.132583    * |15>|0>|0>
 + (-0.132583) * |0>|0>|1>
 + (-0.132583) * |1>|0>|1>
 + (-0.132583) * |2>|0>|1>
 + (-0.132583) * |3>|0>|1>
 + (-0.132583) * |4>|0>|1>
 + (-0.486136) * |5>|0>|1>
 + (-0.132583) * |6>|0>|1>
 + (-0.132583) * |7>|0>|1>
 + (-0.132583) * |8>|0>|1>
 + (-0.132583) * |9>|0>|1>
 + (-0.132583) * |10>|0>|1>
 + (-0.132583) * |11>|0>|1>
 + (-0.132583) * |12>|0>|1>
 + (-0.132583) * |13>|0>|1>
 + (-0.132583) * |14>|0>|1>
 + (-0.132583) * |15>|0>|1>

{6: 39, 9: 45, 0: 28, 3: 31, 5: 472, 8: 30, 13: 33, 7: 32, 14: 40, 1: 48, 11: 35, 2: 33, 15: 33, 10: 35, 4: 30, 12: 36}
```



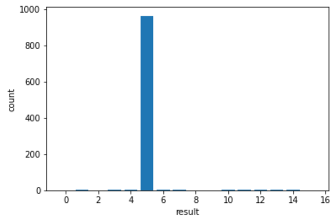

```
Start Grover Iteration 1
|Psi2> =
   0.132583    * |0>|0>|0>
 + 0.132583    * |1>|0>|0>
 + 0.132583    * |2>|0>|0>
 + 0.132583    * |3>|0>|0>
 + 0.132583    * |4>|0>|0>
 + (-0.486136) * |5>|0>|0>
 + 0.132583    * |6>|0>|0>
 + 0.132583    * |7>|0>|0>
 + 0.132583    * |8>|0>|0>
 + 0.132583    * |9>|0>|0>
 + 0.132583    * |10>|0>|0>
 + 0.132583    * |11>|0>|0>
 + 0.132583    * |12>|0>|0>
 + 0.132583    * |13>|0>|0>
 + 0.132583    * |14>|0>|0>
 + 0.132583    * |15>|0>|0>
 + (-0.132583) * |0>|0>|1>
 + (-0.132583) * |1>|0>|1>
 + (-0.132583) * |2>|0>|1>
 + (-0.132583) * |3>|0>|1>
 + (-0.132583) * |4>|0>|1>
 + 0.486136    * |5>|0>|1>
 + (-0.132583) * |6>|0>|1>
 + (-0.132583) * |7>|0>|1>
 + (-0.132583) * |8>|0>|1>
 + (-0.132583) * |9>|0>|1>
 + (-0.132583) * |10>|0>|1>
 + (-0.132583) * |11>|0>|1>
 + (-0.132583) * |12>|0>|1>
 + (-0.132583) * |13>|0>|1>
 + (-0.132583) * |14>|0>|1>
 + (-0.132583) * |15>|0>|1>

|Psi3> =
   0.0552427    * |0>|0>|0>
 + 0.0552427    * |1>|0>|0>
 + 0.0552427    * |2>|0>|0>
 + 0.0552427    * |3>|0>|0>
 + 0.0552427    * |4>|0>|0>
 + 0.673961     * |5>|0>|0>
 + 0.0552427    * |6>|0>|0>
 + 0.0552427    * |7>|0>|0>
 + 0.0552427    * |8>|0>|0>
 + 0.0552427    * |9>|0>|0>
 + 0.0552427    * |10>|0>|0>
 + 0.0552427    * |11>|0>|0>
 + 0.0552427    * |12>|0>|0>
 + 0.0552427    * |13>|0>|0>
 + 0.0552427    * |14>|0>|0>
 + 0.0552427    * |15>|0>|0>
 + (-0.0552427) * |0>|0>|1>
 + (-0.0552427) * |1>|0>|1>
 + (-0.0552427) * |2>|0>|1>
 + (-0.0552427) * |3>|0>|1>
 + (-0.0552427) * |4>|0>|1>
 + (-0.673961)  * |5>|0>|1>
 + (-0.0552427) * |6>|0>|1>
 + (-0.0552427) * |7>|0>|1>
 + (-0.0552427) * |8>|0>|1>
 + (-0.0552427) * |9>|0>|1>
 + (-0.0552427) * |10>|0>|1>
 + (-0.0552427) * |11>|0>|1>
 + (-0.0552427) * |12>|0>|1>
 + (-0.0552427) * |13>|0>|1>
 + (-0.0552427) * |14>|0>|1>
 + (-0.0552427) * |15>|0>|1>

{5: 909, 14: 7, 7: 6, 10: 4, 9: 4, 0: 11, 12: 11, 11: 4, 2: 4, 4: 9, 3: 8, 13: 3, 15: 4, 8: 4, 6: 5, 1: 7}
```




```
Start Grover Iteration 2
|Psi2> =
   0.0552427    * |0>|0>|0>
 + 0.0552427    * |1>|0>|0>
 + 0.0552427    * |2>|0>|0>
 + 0.0552427    * |3>|0>|0>
 + 0.0552427    * |4>|0>|0>
 + (-0.673961)  * |5>|0>|0>
 + 0.0552427    * |6>|0>|0>
 + 0.0552427    * |7>|0>|0>
 + 0.0552427    * |8>|0>|0>
 + 0.0552427    * |9>|0>|0>
 + 0.0552427    * |10>|0>|0>
 + 0.0552427    * |11>|0>|0>
 + 0.0552427    * |12>|0>|0>
 + 0.0552427    * |13>|0>|0>
 + 0.0552427    * |14>|0>|0>
 + 0.0552427    * |15>|0>|0>
 + (-0.0552427) * |0>|0>|1>
 + (-0.0552427) * |1>|0>|1>
 + (-0.0552427) * |2>|0>|1>
 + (-0.0552427) * |3>|0>|1>
 + (-0.0552427) * |4>|0>|1>
 + 0.673961     * |5>|0>|1>
 + (-0.0552427) * |6>|0>|1>
 + (-0.0552427) * |7>|0>|1>
 + (-0.0552427) * |8>|0>|1>
 + (-0.0552427) * |9>|0>|1>
 + (-0.0552427) * |10>|0>|1>
 + (-0.0552427) * |11>|0>|1>
 + (-0.0552427) * |12>|0>|1>
 + (-0.0552427) * |13>|0>|1>
 + (-0.0552427) * |14>|0>|1>
 + (-0.0552427) * |15>|0>|1>

|Psi3> =
   (-0.0359078) * |0>|0>|0>
 + (-0.0359078) * |1>|0>|0>
 + (-0.0359078) * |2>|0>|0>
 + (-0.0359078) * |3>|0>|0>
 + (-0.0359078) * |4>|0>|0>
 + 0.693296     * |5>|0>|0>
 + (-0.0359078) * |6>|0>|0>
 + (-0.0359078) * |7>|0>|0>
 + (-0.0359078) * |8>|0>|0>
 + (-0.0359078) * |9>|0>|0>
 + (-0.0359078) * |10>|0>|0>
 + (-0.0359078) * |11>|0>|0>
 + (-0.0359078) * |12>|0>|0>
 + (-0.0359078) * |13>|0>|0>
 + (-0.0359078) * |14>|0>|0>
 + (-0.0359078) * |15>|0>|0>
 + 0.0359078    * |0>|0>|1>
 + 0.0359078    * |1>|0>|1>
 + 0.0359078    * |2>|0>|1>
 + 0.0359078    * |3>|0>|1>
 + 0.0359078    * |4>|0>|1>
 + (-0.693296)  * |5>|0>|1>
 + 0.0359078    * |6>|0>|1>
 + 0.0359078    * |7>|0>|1>
 + 0.0359078    * |8>|0>|1>
 + 0.0359078    * |9>|0>|1>
```

```
+ 0.0359078    * |10>|0>|1>
+ 0.0359078    * |11>|0>|1>
+ 0.0359078    * |12>|0>|1>
+ 0.0359078    * |13>|0>|1>
+ 0.0359078    * |14>|0>|1>
+ 0.0359078    * |15>|0>|1>
```

```
{5: 963, 1: 4, 12: 5, 13: 3, 11: 2, 14: 3, 6: 4, 4: 3, 7: 2, 10: 4, 15: 1, 2: 1, 8: 1, 9: 1, 0: 1, 3: 2}
```



In [ ]:

In [ ]:

1.2.5 The quantum Fourier transform

- We define the quantum Fourier transform by its action on the basis $|x\rangle$ with $x \in \{0, \ldots, 2^N - 1\}$ by

$$FT\,|x\rangle = \frac{1}{\sqrt{2^N}} \sum_{k=0}^{2^N-1} e^{2\pi i x k/2^N}\,|k\rangle\ . \tag{1.70}$$

- In the exercises, problem set 3, you will show that this is a unitary operation with

$$FT^{-1}\,|x\rangle = FT^{\dagger}\,|x\rangle = \frac{1}{\sqrt{2^N}} \sum_{k=0}^{2^N-1} e^{-2\pi i x k/2^N}\,|k\rangle\ . \tag{1.71}$$

- It is useful to consider the bit-wise representation of

$$x = \sum_{j=1}^{N} x_j 2^{N-j}\,, \qquad k = \sum_{j=1}^{N} k_j 2^{N-j}\,, \tag{1.72}$$

with $x_j, k_j \in \{0, 1\}$. Note that here for convenience $j = 1$ corresponds to the most-significant digit!

Then

$$FT\,|x\rangle = \frac{1}{\sqrt{2^N}} \sum_{k=0}^{2^N-1} e^{2\pi i x k/2^N}\,|k\rangle \tag{1.73}$$

$$= \frac{1}{\sqrt{2^N}} \sum_{k_1,\ldots,k_N=0}^{1} e^{2\pi i x \sum_{j=1}^{N} k_j/2^j}\,|k_1, \ldots, k_N\rangle \tag{1.74}$$

$$= \frac{1}{\sqrt{2^N}} \bigotimes_{j=1}^{N} \left[\sum_{k_j=0}^{1} e^{2\pi i x k_j/2^j}\,|k_j\rangle \right] \tag{1.75}$$

$$= \bigotimes_{j=1}^{N} \left[\frac{|0\rangle + e^{2\pi i x/2^j}\,|1\rangle}{\sqrt{2}} \right] . \tag{1.76}$$

- Since

$$e^{2\pi i n} = 1 \tag{1.77}$$

for $n \in \mathbb{N}$,

$$e^{2\pi i x/2^j} = \prod_{l=1}^{N} e^{2\pi i x_l 2^{N-l-j}} = \prod_{l=N-j+1}^{N} e^{2\pi i x_l 2^{N-l-j}}\ . \tag{1.78}$$

We therefore define the notation

$$0.x_l x_{l+1} \dots x_m \equiv \frac{x_l}{2} + \frac{x_{l+1}}{4} + \dots + \frac{x_m}{2^{m-l+1}} \tag{1.79}$$

and write

$$e^{2\pi i x/2^j} = e^{2\pi i 0.x_{N-j+1} \dots x_N} . \tag{1.80}$$

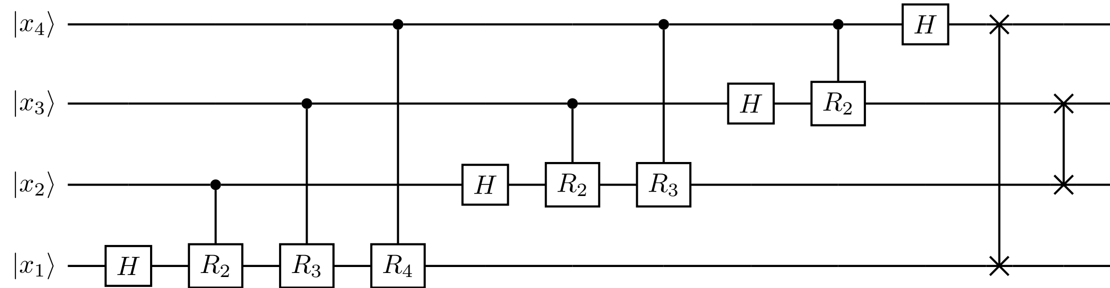
- We therefore have a product representation

$$FT\,|x\rangle = \bigotimes_{j=1}^{N} \left[\frac{|0\rangle + e^{2\pi i 0.x_{N-j+1} \dots x_N} |1\rangle}{\sqrt{2}} \right] \tag{1.81}$$

$$= \left[\frac{|0\rangle + e^{2\pi i 0.x_N} |1\rangle}{\sqrt{2}} \right] \left[\frac{|0\rangle + e^{2\pi i 0.x_{N-1} x_N} |1\rangle}{\sqrt{2}} \right] \cdots \left[\frac{|0\rangle + e^{2\pi i 0.x_1 \dots x_N} |1\rangle}{\sqrt{2}} \right] .$$

- From this representation a circuit can be read off. Example $N = 4$:

$|x\rangle$ —/4— FT —/4— =



with

$$R_k \equiv R_{2\pi/2^k} . \tag{1.82}$$

Note, we write again the least-significant bit on top.

Explanation:

Applying the Hadamard to the least-significant qubit $|x_4\rangle$ produces

$$H\,|x_4\rangle = \frac{|0\rangle + (-1)^{x_4} |1\rangle}{\sqrt{2}} = \frac{|0\rangle + e^{2\pi i 0.x_4} |1\rangle}{\sqrt{2}} , \tag{1.83}$$

which then needs to be placed as the most-significant result qubit by SWAPs.

Similarly,

$$H\,|x_3\rangle = \frac{|0\rangle + e^{2\pi i 0.x_3} |1\rangle}{\sqrt{2}} , \tag{1.84}$$

and therefore the controlled R_2 gate with control qubit $|x_4\rangle$ yields

$$\frac{|0\rangle + e^{2\pi i 0.x_3} e^{2\pi i/2^2 x_4} |1\rangle}{\sqrt{2}} = \frac{|0\rangle + e^{2\pi i 0.x_3 x_4} |1\rangle}{\sqrt{2}}, \tag{1.85}$$

which we then also need to place as second most-significant result qubit by SWAPs. Continuing this procedure provides the general circuit for FT for arbitrary N.

- We therefore need to apply $O(N^2)$ gates to implement the quantum Fourier transform for N qubits. Classically, the naive discrete Fourier transform requires $O(2^{2N})$ operations and even the Fast Fourier Transform algorithm requires $O(N2^N)$ operations.

1.2.6 Example: Addition through Fourier transform

- We now show that FT, FT†, and R_ϕ gates suffice to implement

$$U_y |x\rangle = \left|(x+y) \bmod 2^N\right\rangle \tag{1.86}$$

with $x, y \in \{0, \ldots, 2^N - 1\}$.

- Consider

$$FT|x\rangle = \frac{1}{\sqrt{2^N}} \sum_{k=0}^{2^N-1} e^{2\pi i x k/2^N} |k\rangle \tag{1.87}$$

and applying R_ϕ to the l-th qubit, counting $l = 1$ for the most-significant qubit and $l = N$ for the least-significant qubit,

$$R^{(l)}_{2\pi/2^l} FT|x\rangle = \frac{1}{\sqrt{2^N}} \sum_{k=0}^{2^N-1} e^{2\pi i (xk + k_l 2^{N-l})/2^N} |k\rangle \tag{1.88}$$

with

$$k = \sum_{j=1}^{N} k_j 2^{N-j}. \tag{1.89}$$

Then

$$\left[\prod_{l=1}^{N} R^{(l)}_{2\pi/2^l}\right] FT|x\rangle = \frac{1}{\sqrt{2^N}} \sum_{k=0}^{2^N-1} e^{2\pi i (xk+k)/2^N} |k\rangle \tag{1.90}$$

$$= \frac{1}{\sqrt{2^N}} \sum_{k=0}^{2^N-1} e^{2\pi i (x+1)k/2^N} |k\rangle. \tag{1.91}$$

Therefore

$$FT^{\dagger}\left[\prod_{l=1}^{N} R_{2\pi/2^l}^{(l)}\right] FT\,|x\rangle = \left|(x+1) \bmod 2^N\right\rangle . \tag{1.92}$$

It follows that

$$U_y = FT^{\dagger}\left[\prod_{l=1}^{N} R_{2\pi/2^l}^{(l)}\right]^y FT = FT^{\dagger}\left[\prod_{l=1}^{N} R_{2\pi y/2^l}^{(l)}\right] FT\,. \tag{1.93}$$

- Illustrate for $N = 3$:

$$\begin{aligned}
U_1\,|3\rangle &= U_1\,|011\rangle & (1.94)\\
&= FT^{\dagger} R_{\pi/4}^{(3)} R_{\pi/2}^{(2)} R_{\pi}^{(1)} FT\,|011\rangle & (1.95)\\
&= \frac{1}{\sqrt{8}} FT^{\dagger} R_{\pi/4}^{(3)} R_{\pi/2}^{(2)} R_{\pi}^{(1)} \Big[|0\rangle + e^{2\pi i3/8}\,|1\rangle + e^{2\pi i6/8}\,|2\rangle + e^{2\pi i9/8}\,|3\rangle \\
&\quad + e^{2\pi i12/8}\,|4\rangle + e^{2\pi i15/8}\,|5\rangle + e^{2\pi i18/8}\,|6\rangle + e^{2\pi i21/8}\,|7\rangle\Big] & (1.96)\\
&= \frac{1}{\sqrt{8}} FT^{\dagger} R_{\pi/4}^{(3)} R_{\pi/2}^{(2)} R_{\pi}^{(1)} \Big[|000\rangle + e^{2\pi i3/8}\,|001\rangle + e^{2\pi i6/8}\,|010\rangle + e^{2\pi i9/8}\,|011\rangle \\
&\quad + e^{2\pi i12/8}\,|100\rangle + e^{2\pi i15/8}\,|101\rangle + e^{2\pi i18/8}\,|110\rangle + e^{2\pi i21/8}\,|111\rangle\Big] & (1.97)\\
&= \frac{1}{\sqrt{8}} FT^{\dagger} R_{\pi/4}^{(3)} R_{\pi/2}^{(2)} \Big[|000\rangle + e^{2\pi i3/8}\,|001\rangle + e^{2\pi i6/8}\,|010\rangle + e^{2\pi i9/8}\,|011\rangle \\
&\quad + e^{2\pi i16/8}\,|100\rangle + e^{2\pi i19/8}\,|101\rangle + e^{2\pi i22/8}\,|110\rangle + e^{2\pi i25/8}\,|111\rangle\Big] & (1.98)\\
&= \frac{1}{\sqrt{8}} FT^{\dagger} R_{\pi/4}^{(3)} \Big[|000\rangle + e^{2\pi i3/8}\,|001\rangle + e^{2\pi i8/8}\,|010\rangle + e^{2\pi i11/8}\,|011\rangle \\
&\quad + e^{2\pi i16/8}\,|100\rangle + e^{2\pi i19/8}\,|101\rangle + e^{2\pi i24/8}\,|110\rangle + e^{2\pi i27/8}\,|111\rangle\Big] & (1.99)\\
&= \frac{1}{\sqrt{8}} FT^{\dagger} \Big[|000\rangle + e^{2\pi i4/8}\,|001\rangle + e^{2\pi i8/8}\,|010\rangle + e^{2\pi i12/8}\,|011\rangle \\
&\quad + e^{2\pi i16/8}\,|100\rangle + e^{2\pi i20/8}\,|101\rangle + e^{2\pi i24/8}\,|110\rangle + e^{2\pi i28/8}\,|111\rangle\Big] \\
&= FT^{\dagger} FT\,|4\rangle = |4\rangle\ . & (1.100)
\end{aligned}$$

- Implementation on sqc:

In [1]:
```
import sqc
import numpy as np
from exercises import qft
```

Addition through Fourier transform

In [10]:
```
def add(y,op):
    op=qft(op)
    N=op.nbits
    for i in range(N):
        op=op.Rz(i,2.*np.pi*y/2.**(N-i))
    op=qft(op,inverse=True)
    return op

st=sqc.state(5,basis=["|%d>" % i for i in range(2**5)])

print("Initial = 0\n",st)
st=add(3,sqc.operator(5))*st
print("Initial + 3\n",st)
st=add(7,sqc.operator(5))*st
print("Initial + 3 + 7\n",st)
```

```
Initial = 0
    1 * |0>
Initial + 3
    1 * |3>
Initial + 3 + 7
    1 * |10>
```


1.2.7 Phase estimation

- Consider a unitary operation U with eigenstate $|u\rangle$ such that

$$U|u\rangle = \lambda|u\rangle \,. \tag{1.101}$$

Taking the squared norm of both sides, we find

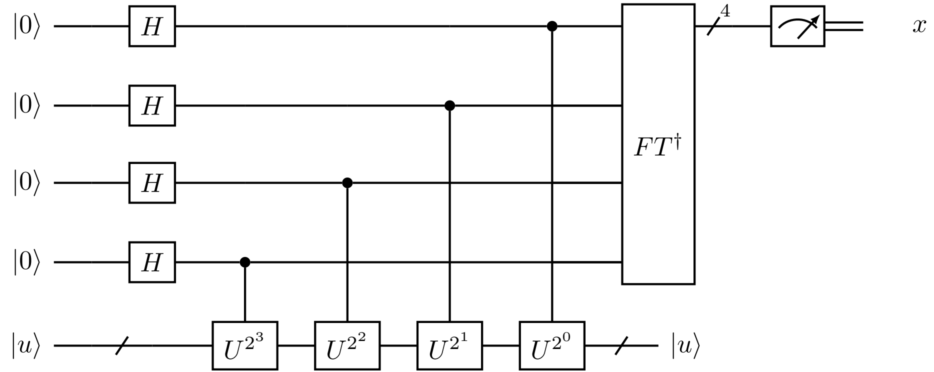
$$\begin{aligned}\langle u|u\rangle &= \langle u| U^\dagger U |u\rangle \\ &= |\lambda|^2 \langle u|u\rangle\end{aligned} \tag{1.102}$$

such that $|\lambda| = 1$ and we can always write

$$U |u\rangle = e^{2\pi i\phi} |u\rangle \tag{1.103}$$

with $0 \leq \phi < 1$.

- Task: find a quantum algorithm to approximate ϕ for a general unitary U. We attempt an N-bit approximation $\tilde{\phi} = x/2^N \approx \phi$ with $0 \leq x < 2^N$ given $|u\rangle$.
- We require the implementation of a controlled-U^j gate with non-negative j. Exercises: how to obtain such a gate from the H, R_ϕ, and CNOT gates.
- Circuit for $N = 4$, applied to eigenstate $|u\rangle$:



with least-significant qubit on top.

Explanation:

After the initial Hadamard gates, the system is in state

$$\frac{1}{\sqrt{2^N}} \sum_{x=0}^{2^N-1} |x\rangle |u\rangle = \frac{1}{\sqrt{2^N}} [|0\rangle + |1\rangle] \cdots [|0\rangle + |1\rangle] |u\rangle \ . \tag{1.104}$$

The controlled-U^j gates, add a phase to the $|1\rangle$ components, leaving the system in state

$$\frac{1}{\sqrt{2^N}} \left[|0\rangle + e^{2\pi i\phi 2^{N-1}} |1\rangle\right] \cdots \left[|0\rangle + e^{2\pi i\phi 2^0} |1\rangle\right] |u\rangle = \frac{1}{\sqrt{2^N}} \sum_{k=0}^{2^N-1} e^{2\pi i\phi k} |k\rangle |u\rangle \ . \tag{1.105}$$

The U^j gates only act if the respective control bits are set, and then produce the phase $e^{2\pi i\phi j}$.

If there was an integer x such that $\phi = x/2^N$, then applying the inverse Fourier transform, leaves the system in state $|x\rangle$ and thus measuring the first N qubits yields x and thus ϕ.

In general, there may only be $\tilde{\phi} = x/2^N \approx \phi$. It can be shown that for $|\phi - \tilde{\phi}| \leq 2^{-n}$ and probability of success at least $1 - \varepsilon$, we need

$$N = n + \log_2 \left(2 + \frac{1}{2\varepsilon}\right) \tag{1.106}$$

qubits.

- If we apply this algorithm to a general state

$$|\psi\rangle = \sum_n c_n |u_n\rangle \tag{1.107}$$

with eigenstates $|u_n\rangle$, we will measure an approximation of eigenvalue n with probability $|c_n|^2$.

For sufficiently many measurements can obtain all eigenvalues ϕ.

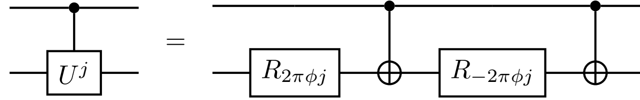
- Example 1: compute eigenvalues of

$$U = \begin{pmatrix} e^{-2\pi i\phi} & 0 \\ 0 & e^{2\pi i\phi} \end{pmatrix} \tag{1.108}$$

with $\phi \in [0, 1[$. First need U^j, which is straightforward

$$U = \begin{pmatrix} e^{-2\pi i\phi j} & 0 \\ 0 & e^{2\pi i\phi j} \end{pmatrix} . \tag{1.109}$$

Then, we need a circuit to implement the controlled-U^j. Analog to problem set 02,



In [1]:
```
import sqc
import numpy as np
from exercises import qft
import matplotlib.pyplot as plt
```

Phase estimation

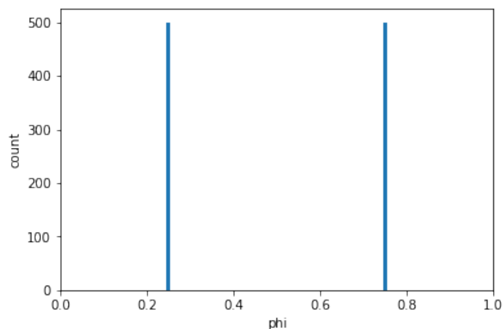
In [10]:
```
def phaseEstimate(op,xbits,cuj):
    N=len(xbits)
    for i in reversed(range(N)):
        op=op.H(xbits[i])
        op=cuj(xbits[i],2**i,op)
    op=qft(op,mask=xbits,inverse=True)
    return op

# Simple U = {{ Exp[I 2pi phi], 0}, { 0, Exp[-I 2pi phi] }}, always acting on LSB
def CU(i,k,op,phi): # i is control qubit, k is power
    return op.Rz(0,2.*np.pi*phi*k).CNOT(i,0).Rz(0,-2.*np.pi*phi*k).CNOT(i,0)

def measure(Nxbits,Nmeasure,cuj):
    Nbits=Nxbits+1
    xbits=list(range(1,Nbits))

    st0=sqc.state(Nbits,basis=["|%g>|%d>" % ( (i//2) / 2**Nxbits,i%2) for i in range(2**Nbits)])
    print("Initial = 0\n",st0)

    st1=phaseEstimate(sqc.operator(Nbits).H(0),xbits,cuj)*st0

    if Nmeasure == 0:
        print("State after phaseEstimate\n",st1)
    else:
        res=sqc.sample(st1,Nmeasure,mask=xbits)

        plt.bar([ (x//2) / 2**Nxbits for x in res.keys() ],res.values(),width=0.01)
        plt.xlabel('phi')
        plt.xlim(0,1)
        plt.ylabel('count')
        plt.show()
```

In [11]:
```
measure(4,0,lambda i,k,op: CU(i,k,op,0.25))
```

```
Initial = 0
    1 * |0>|0>
State after phaseEstimate
    0.707107 * |0.25>|1>
 + 0.707107 * |0.75>|0>
```

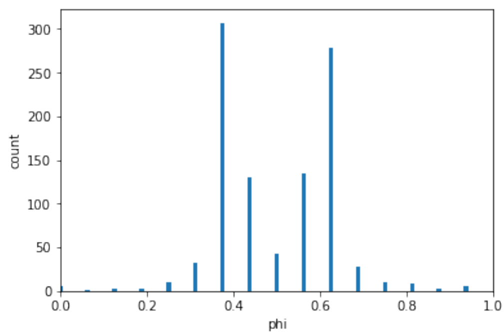
In [12]:
```
measure(4,1000,lambda i,k,op: CU(i,k,op,0.25))
```

```
Initial = 0
    1 * |0>|0>
```



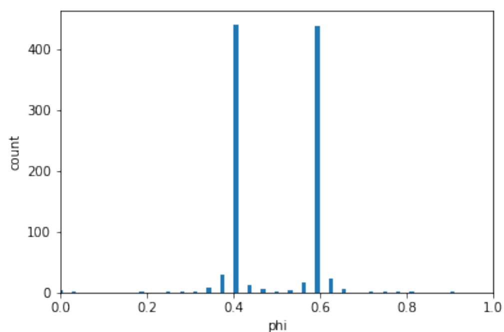
In [13]:
```
measure(4,1000,lambda i,k,op: CU(i,k,op,0.4))
```

```
Initial = 0
    1 * |0>|0>
```



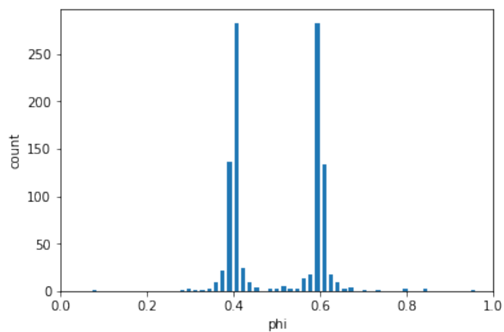
In [14]:
```
measure(5,1000,lambda i,k,op: CU(i,k,op,0.4))
```

```
Initial = 0
    1 * |0>|0>
```



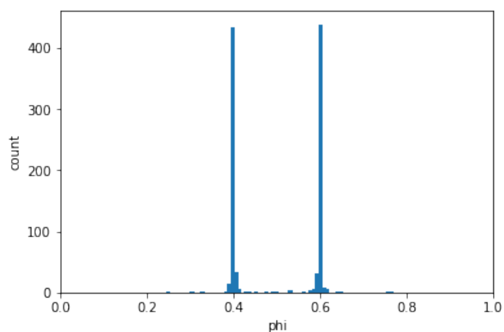
In [15]:
```
measure(6,1000,lambda i,k,op: CU(i,k,op,0.4))
```

```
Initial = 0
    1 * |0>|0>
```



In [16]:
```
measure(7,1000,lambda i,k,op: CU(i,k,op,0.4))
```

```
Initial = 0
    1 * |0>|0>
```

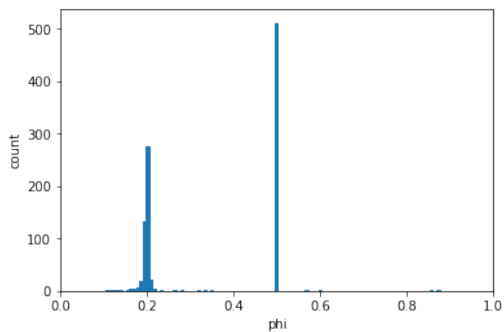


More general one-qubit matrices

In [1]:
```
def CU2(i,k,op,phi,theta): # i is control qubit, k is power
    return op.Rz(0,np.pi*(phi-theta)*k).CNOT(i,0).Rz(
        0,np.pi*(theta-phi)*k).CNOT(i,0).Rz(i,np.pi*(theta+phi)*k)
```


In [19]:
```
measure(7,1000,lambda i,k,op: CU2(i,k,op,0.2,0.5))
```

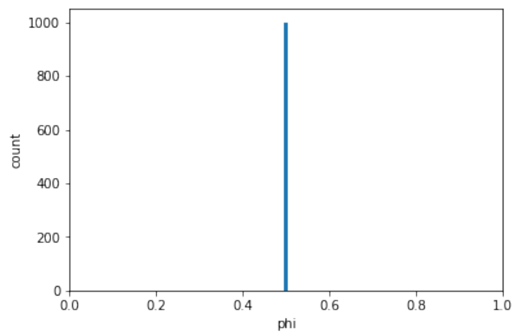
```
Initial = 0
    1 * |0>|0>
```



In [2]:
```
def CU2H(i,k,op,phi,theta): # i is control qubit, k is power
    return op.H(0).Rz(0,np.pi*(phi-theta)*k).CNOT(i,0).Rz(
        0,np.pi*(theta-phi)*k).CNOT(i,0).Rz(i,np.pi*(theta+phi)*k).H(0)
```

In [21]:
```
measure(7,1000,lambda i,k,op: CU2H(i,k,op,0.2,0.5))
```

```
Initial = 0
    1 * |0>|0>
```

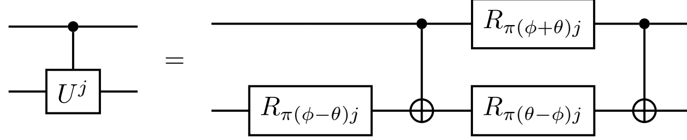


In [ ]:

- Example 2: a more general

$$U = \begin{pmatrix} e^{2\pi i\theta} & 0 \\ 0 & e^{2\pi i\phi} \end{pmatrix} \tag{1.110}$$

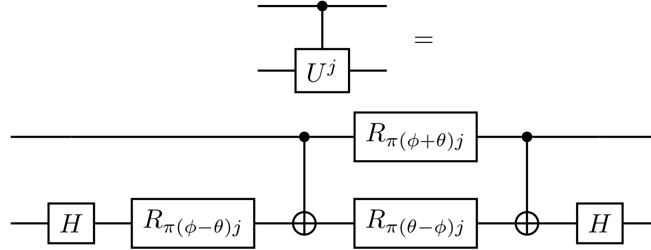
with $\phi, \theta \in [0, 1[$ can be implemented with



- Example 3: A basis change such as

$$U \to HUH = \frac{1}{2} \begin{pmatrix} e^{2\pi i\phi} + e^{2\pi i\theta} & -e^{2\pi i\phi} + e^{2\pi i\theta} \\ -e^{2\pi i\phi} + e^{2\pi i\theta} & e^{2\pi i\phi} + e^{2\pi i\theta} \end{pmatrix} \tag{1.111}$$

which should leave the eigenvalues unchanged can be implemented as



1.2.8 Example: Order-finding

- Consider the problem of finding the least positive integer r for which

$$x^r = 1 \bmod n \tag{1.112}$$

given positive integers x and n.

- Example: $n = 21$, $x = 5$

$$5^1 = 5 \bmod 21\,, \qquad 5^2 = 4 \bmod 21\,, \qquad 5^3 = 20 \bmod 21\,, \tag{1.113}$$

$$5^4 = 16 \bmod 21\,, \qquad 5^5 = 17 \bmod 21\,, \qquad 5^6 = 1 \bmod 21\,, \tag{1.114}$$

such that $r = 6$.

- No classical algorithm of complexity $\mathrm{O}(\log n)$ is known. Here, devise quantum algorithm of $\mathrm{O}(\log n)$.

- We will use this as part of Shor's algorithm to factor integers.

- Strategy:
  1. Consider operator $U^{(\times)}$ defined through
  $$U_{x,n}^{(\times)} |y\rangle \equiv \begin{cases} |xy \bmod n\rangle & \text{for } y < n\,, \\ |y\rangle & \text{else} \end{cases} \tag{1.115}$$
  with integers $0 \leq x, y, n < 2^N$.
  2. We can show that
  $$|u_s\rangle \equiv \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} e^{-2\pi i k s/r} \left|x^k \bmod n\right\rangle \tag{1.116}$$
  for $0 \leq s < r$ are eigenstates of $U_{x,n}^{(\times)}$ with eigenvalues
  $$\lambda_s \equiv e^{2\pi i s/r} \tag{1.117}$$
  since
  $$U_{x,n}^{(\times)} |u_s\rangle = \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} e^{-2\pi i k s/r} \left|x^{k+1} \bmod n\right\rangle \tag{1.118}$$
  $$= e^{2\pi i s/r} \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} e^{-2\pi i (k+1) s/r} \left|x^{k+1} \bmod n\right\rangle \tag{1.119}$$
  $$= e^{2\pi i s/r} \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} e^{-2\pi i k s/r} \left|x^{k} \bmod n\right\rangle \tag{1.120}$$
  $$= e^{2\pi i s/r} |u_s\rangle\ . \tag{1.121}$$
  3. If we now perform the phase estimation algorithm with initial state
  $$|\psi\rangle = \frac{1}{\sqrt{r}} \sum_{s=0}^{r-1} |u_s\rangle = \frac{1}{r} \sum_{k,s=0}^{r-1} e^{-2\pi i k s/r} \left|x^k \bmod n\right\rangle \tag{1.122}$$
  $$= \sum_{k=0}^{r-1} \delta_{0,k} \left|x^k \bmod n\right\rangle = |1\rangle \tag{1.123}$$
  we will find eigenvalues λ_s for $s = 0, \ldots, r-1$ from which we can infer r.

- To find the eigenvalues of $U = U_{x,n}^{(\times)}$ using the phase estimation algorithm, we need a circuit to peform the controlled U^{2^j} operation.
  1. To this end, we first implement a controlled-addition of a constant integer. We have shown in Sec. 1.2.6 how to implement
  $$U_x |y\rangle = \left|(x+y) \bmod 2^N\right\rangle \tag{1.124}$$

with $x, y \in \{0, \ldots, 2^N - 1\}$ using only R_ϕ and FT gates. Replacing the R_ϕ and H gates in this circuit by controlled versions yields to desired controlled-addition. We illustrate in the simulator.

In [1]:
```
import sqc
import numpy as np
from exercises import cqft
```

Addition through Fourier transform with control bit

In [8]:
```
def CRz(i,t,phi,o):
    return o.Rz(t,phi/2.).CNOT(i,t).Rz(t,-phi/2.).Rz(i,phi/2.).CNOT(i,t)

def add(c,y,op,xbits):
    op=cqft(c,op,mask=xbits)
    N=len(xbits)
    for i in range(N):
        op=CRz(c,xbits[i],2.*np.pi*y/2.**(N-i),op)
    op=cqft(c,op,inverse=True,mask=xbits)
    return op

st=sqc.state(6,basis=["|%d>|%d>" % (i % 2**5,i//2**5) for i in range(2**6)])

xbits=[0,1,2,3,4]
cbit=5

st=sqc.operator(6).H(cbit)*st
print("Initial = 0\n",st)

st=add(cbit,3,sqc.operator(6),xbits=xbits)*st
print("Initial + 3\n",st)
st=add(cbit,7,sqc.operator(6),xbits=xbits)*st
print("Initial + 3 + 7\n",st)
```

```
Initial = 0
    0.707107 * |0>|0>
 + 0.707107 * |0>|1>
Initial + 3
    0.707107 * |0>|0>
 + 0.707107 * |3>|1>
Initial + 3 + 7
    0.707107 * |0>|0>
 + 0.707107 * |10>|1>
```

In [ ]:

2. Next, we need a controlled-addition by a constant x modulo $n < 2^N$

with $x < n$. Specifically, we want a gate that performs

$$U_{x,n}\,|y\rangle = \begin{cases} |y\rangle \ \text{ for } y \geq n\,, \\ |x+y-n\rangle \ \text{ for } x+y \geq n\,, \\ |x+y\rangle \ \text{ else} \end{cases} \tag{1.125}$$

The matrix representation of this gate is a simple permutation and therefore unitary. The circuit will be discussed later.

3. Need to implement a (controlled) multiplication by constant x modulo n. Consider

$$y = \sum_{i=0}^{N-1} a_i 2^i\,, \qquad\qquad x = \sum_{i=0}^{N-1} b_i 2^i \tag{1.126}$$

then the desired result is

$$z \equiv xy \bmod n = \sum_{i}^{N-1} a_i c_i \bmod n \tag{1.127}$$

with

$$c_i = \sum_{j=0}^{N-1} b_j 2^{i+j} \bmod n\,. \tag{1.128}$$

Note that c_i only depend on x and n and can therefore be precomputed classically. Example:

$$x = 5\,, \qquad n = 21\,, \qquad N = 5\,, \tag{1.129}$$
$$c_0 = 5\,, \qquad c_1 = 10\,, \qquad c_2 = 20\,, \tag{1.130}$$
$$c_3 = 19\,, \qquad c_4 = 17 \tag{1.131}$$

This can therefore be implemented with a gate for controlled addition modulo n. First example: $n = 2^N$

In [7]:
```
import sqc
import numpy as np
from exercises import cqft
```

Addition through Fourier transform with control bit

In [8]:
```
def CRz(i,t,phi,o):
    return o.Rz(t,phi/2.).CNOT(i,t).Rz(t,-phi/2.).Rz(i,phi/2.).CNOT(i,t)

def add(c,y,op,xbits):
    op=cqft(c,op,mask=xbits)
    N=len(xbits)
    for i in range(N):
        op=CRz(c,xbits[i],2.*np.pi*y/2.**(N-i),op)
    op=cqft(c,op,inverse=True,mask=xbits)
    return op

st=sqc.state(6,basis=["|%d>|%d>" % (i % 2**5,i//2**5) for i in range(2**6)])

xbits=[0,1,2,3,4]
cbit=5

st=sqc.operator(6).H(cbit)*st
print("Initial = 0\n",st)

st=add(cbit,3,sqc.operator(6),xbits=xbits)*st
print("Initial + 3\n",st)
st=add(cbit,7,sqc.operator(6),xbits=xbits)*st
print("Initial + 3 + 7\n",st)
```

```
Initial = 0
    0.707107 * |0>|0>
 + 0.707107 * |0>|1>
Initial + 3
    0.707107 * |0>|0>
 + 0.707107 * |3>|1>
Initial + 3 + 7
    0.707107 * |0>|0>
 + 0.707107 * |10>|1>
```

Precompute ci

In [9]:
```
def bits(y,N):
    return [ (y//2**j) % 2 for j in range(N) ]

def ci(x,n,N):
    b=bits(x,N)
    return [ sum([ b[j]*2**(i+j) for j in range(N) ]) % n for i in range(N) ]
```

In [5]:
```
print(ci(5,21,5))
```

```
[5, 10, 20, 19, 17]
```

Multiplication by constant (without mod n)

In [17]:
```
def mult(x,op,xbits,tbits):
    Nbits=len(xbits)
    c=ci(x,2**Nbits,Nbits)
    for i in range(Nbits):
        op=add(xbits[i],c[i],op,tbits)
    return op

st=sqc.state(6,basis=["|%d>|%d>" % (i % 2**3,i//2**3) for i in range(2**6)])

xbits=[0,1,2]
tbits=[3,4,5]

st=sqc.operator(6).X(1)*st
print("Initial state\n",st)

st=mult(3,sqc.operator(6),xbits=xbits,tbits=tbits)*st
print("Mult by 3\n",st)
```

```
Initial state
    1 * |2>|0>
Mult by 3
    1 * |2>|6>
```

In [ ]:

- Recapitulation/next steps:

1. For Shor's algorithm, we need to find the smallest integer r for which

$$x^r = 1 \bmod n \tag{1.132}$$

for positive integers x and n.

2. We have shown that the phase estimation algorithm applied to the gate

$$U_{x,n}^{(\times)} |y\rangle \equiv \begin{cases} |xy \bmod n\rangle & \text{for } y < n\,, \\ |y\rangle & \text{else} \end{cases} \tag{1.133}$$

for integers $0 \leq x, y, n < 2^N$ can be used to efficiently solve for r. We therefore need a circuit to implement a controlled version of this gate.

3. Have shown how to implement

$$U_x^{(+)} |y\rangle = \left|(x+y) \bmod 2^N\right\rangle \tag{1.134}$$

as well as

$$U_x^{(\times)} |y\rangle = \left|xy \bmod 2^N\right\rangle \tag{1.135}$$

with $x, y \in \{0, \ldots, 2^N - 1\}$ using only R_ϕ and FT gates.

4. Next, we implement an auxilliary gate

$$U_x^{(<)} |y\rangle |t\rangle = |y\rangle |t \oplus (y < x)\rangle\ , \tag{1.136}$$

where $(y < x)$ evaluates to 1 if $y < x$ and 0 else.

Do this in the simulator

5. With this auxilliary gate, we then implement

$$U_{x,n}^{(+)} |y\rangle = \begin{cases} |y\rangle & \text{for } y \geq n\,, \\ |x + y - n\rangle & \text{for } x + y \geq n\,, \\ |x + y\rangle & \text{else} \end{cases} \tag{1.137}$$

for $n < 2^N$ and $x < n$.

Do this in the simulator

6. With this gate, we can then finally implement

$$U_{x,n}^{(\times)} \tag{1.138}$$

and can estimate its eigenvalues.

Do this in the simulator

In [1]:
```
import sqc
import numpy as np
from exercises import cqft, qft, CRz, cadd, add, SWAP, CSWAP, C2NOT, phaseEstimate
import matplotlib.pyplot as plt
```

If $y < c$

In [2]:
```
def smallerThan(c,t,xbits,a,op):
    # performs NOT operation on t qubit if x<c, a is ancilla qubit |0>
    pbits=xbits + [a]
    N=len(pbits)
    op=add(2**N - c,op,pbits)
    op=op.CNOT(a,t)
    op=add(c,op,pbits)
    return op
```

In [3]:
```
def pr(i,start,width):
    return (i//2**start)%(2**width)

st=sqc.state(6,basis = ["|%d>|%d>|%d>" % (pr(i,0,4),pr(i,4,1),pr(i,5,1))
                        for i in range(2**6)])
```

In [4]:
```
st=sqc.operator(6).X(0).X(2)*st
par_xbits=[0,1,2,3]
par_tbit=5
par_abit=4
print(st)
```

```
1 * |5>|0>|0>
```

In [5]:
```
for c in range(2**4):
```



```
    st2=smallerThan(c,par_tbit,par_xbits,par_abit,sqc.operator(6))*st
    print("|x>|0>|(x < %d)> =\n" % c,st2)
```

```
|x>|0>|(x < 0)> =
  1 * |5>|0>|0>
|x>|0>|(x < 1)> =
  1 * |5>|0>|0>
|x>|0>|(x < 2)> =
  1 * |5>|0>|0>
|x>|0>|(x < 3)> =
  1 * |5>|0>|0>
|x>|0>|(x < 4)> =
  1 * |5>|0>|0>
|x>|0>|(x < 5)> =
  1 * |5>|0>|0>
|x>|0>|(x < 6)> =
  1 * |5>|0>|1>
|x>|0>|(x < 7)> =
  1 * |5>|0>|1>
|x>|0>|(x < 8)> =
  1 * |5>|0>|1>
|x>|0>|(x < 9)> =
  1 * |5>|0>|1>
|x>|0>|(x < 10)> =
  1 * |5>|0>|1>
|x>|0>|(x < 11)> =
  1 * |5>|0>|1>
|x>|0>|(x < 12)> =
  1 * |5>|0>|1>
|x>|0>|(x < 13)> =
  1 * |5>|0>|1>
|x>|0>|(x < 14)> =
  1 * |5>|0>|1>
|x>|0>|(x < 15)> =
  1 * |5>|0>|1>
```

Addition modulo n

In [6]:

```
def modadd(x,op,ybits,n,abits):
    assert(len(abits)==3)
    assert(x<n)
    a=abits[0]
    c0=abits[1] # y < n
    c1=abits[2] # y < n-x

    op=smallerThan(n,c0,ybits,a,op)
    op=smallerThan(n-x,c1,ybits,a,op)

    # if c0 and not c1: y = y + x - n  ->
    op=C2NOT(c0,c1,a,op.X(c1)).X(c1)
    op=cadd(a,x-n,op,ybits)
    op=C2NOT(c0,c1,a,op.X(c1)).X(c1)

    # if c0 and c1: y = y + x
    op=C2NOT(c0,c1,a,op)
    op=cadd(a,x,op,ybits)
    op=C2NOT(c0,c1,a,op)

    # untrigger c0 and c1 based on result in y
    op=smallerThan(x,c1,ybits,a,op)
    # (x+y) % n < x   -> c1 was not set, after this c1 will be equal to c0
    op=op.CNOT(c0,c1) # clear c1
    op=smallerThan(n,c0,ybits,a,op) # clear c0
    return op

st=sqc.state(7,basis = ["|%d>|a=%d>|c0=%d>|c1=%d>" %
                        (pr(i,0,4),pr(i,4,1),pr(i,5,1),pr(i,6,1)) for i in range(2**7)])
```

In [7]:

```
print(st)
par_ybits=[0,1,2,3]
```



```
par_abits=[4,5,6]
op=modadd(3,sqc.operator(7),par_ybits,11,par_abits)
opinc=add(1,sqc.operator(7),par_ybits)
st0=st
for y in range(2**len(par_ybits)):
    st1=op*st0
    print("(%d + 3) %% 11 = \n" % y,st1)
    st0=opinc*st0
```

```
   1 * |0>|a=0>|c0=0>|c1=0>
(0 + 3) % 11 =
    1 * |3>|a=0>|c0=0>|c1=0>
(1 + 3) % 11 =
    1 * |4>|a=0>|c0=0>|c1=0>
(2 + 3) % 11 =
    1 * |5>|a=0>|c0=0>|c1=0>
(3 + 3) % 11 =
    1 * |6>|a=0>|c0=0>|c1=0>
(4 + 3) % 11 =
    1 * |7>|a=0>|c0=0>|c1=0>
(5 + 3) % 11 =
    1 * |8>|a=0>|c0=0>|c1=0>
(6 + 3) % 11 =
    1 * |9>|a=0>|c0=0>|c1=0>
(7 + 3) % 11 =
    1 * |10>|a=0>|c0=0>|c1=0>
(8 + 3) % 11 =
    1 * |0>|a=0>|c0=0>|c1=0>
(9 + 3) % 11 =
    1 * |1>|a=0>|c0=0>|c1=0>
(10 + 3) % 11 =
    1 * |2>|a=0>|c0=0>|c1=0>
(11 + 3) % 11 =
    1 * |11>|a=0>|c0=0>|c1=0>
(12 + 3) % 11 =
    1 * |12>|a=0>|c0=0>|c1=0>
(13 + 3) % 11 =
```

```
    1 * |13>|a=0>|c0=0>|c1=0>
(14 + 3) % 11 =
    1 * |14>|a=0>|c0=0>|c1=0>
(15 + 3) % 11 =
    1 * |15>|a=0>|c0=0>|c1=0>
```

Controlled addition modulo n

In [8]:
```
def csmallerThan(c0,c,t,xbits,a,op): # control bit c0
    pbits=xbits + [a]
    N=len(pbits)
    op=cadd(c0,2**N - c,op,pbits)
    op=op.CNOT(a,t)
    op=cadd(c0,c,op,pbits)
    return op

def cmodadd(c,x,op,ybits,n,abits):
    assert(len(abits)==3)
    assert(x<n)
    a=abits[0]
    c0=abits[1] # y < n
    c1=abits[2] # y < n-x

    op=csmallerThan(c,n,c0,ybits,a,op)
    op=csmallerThan(c,n-x,c1,ybits,a,op)

    # if c0 and not c1: y = y + x - n  ->
    op=C2NOT(c0,c1,a,op.X(c1)).X(c1)
    op=cadd(a,x-n,op,ybits)
    op=C2NOT(c0,c1,a,op.X(c1)).X(c1)

    # if c0 and c1: y = y + x
    op=C2NOT(c0,c1,a,op)
    op=cadd(a,x,op,ybits)
```



```
    op=C2NOT(c0,c1,a,op)

    # untrigger c0 and c1 based on result in y
    op=csmallerThan(c,x,c1,ybits,a,op)
    # (x+y) % n < x   -> c1 was not set, after this c1 will be equal to c0

    op=op.CNOT(c0,c1) # clear c1
    op=csmallerThan(c,n,c0,ybits,a,op) # clear c0
    return op

st=sqc.state(8,basis = ["|%d>|a=%d>|c0=%d>|c1=%d>|c=%d>" %
                        (pr(i,0,4),pr(i,4,1),pr(i,5,1),pr(i,6,1),pr(i,7,1)) for i in range(2**8)])
```

In [9]:
```
print(st)
par_ybits=[0,1,2,3]
par_abits=[4,5,6]
par_cbit=7
op=cmodadd(par_cbit,3,sqc.operator(8),par_ybits,11,par_abits)
opinc=add(1,sqc.operator(8),par_ybits)
st0=sqc.operator(8).H(par_cbit)*st
for y in range(2**len(par_ybits)):
    st1=op*st0
    print("(%d + 3) %% 11 = \n" % y,st1)
    st0=opinc*st0
```

```
   1 * |0>|a=0>|c0=0>|c1=0>|c=0>
(0 + 3) % 11 =
    0.707107 * |0>|a=0>|c0=0>|c1=0>|c=0>
 + 0.707107 * |3>|a=0>|c0=0>|c1=0>|c=1>
(1 + 3) % 11 =
    0.707107 * |1>|a=0>|c0=0>|c1=0>|c=0>
 + 0.707107 * |4>|a=0>|c0=0>|c1=0>|c=1>
(2 + 3) % 11 =
    0.707107 * |2>|a=0>|c0=0>|c1=0>|c=0>
 + 0.707107 * |5>|a=0>|c0=0>|c1=0>|c=1>
(3 + 3) % 11 =
    0.707107 * |3>|a=0>|c0=0>|c1=0>|c=0>
```

```
 + 0.707107 * |6>|a=0>|c0=0>|c1=0>|c=1>
(4 + 3) % 11 =
    0.707107 * |4>|a=0>|c0=0>|c1=0>|c=0>
 + 0.707107 * |7>|a=0>|c0=0>|c1=0>|c=1>
(5 + 3) % 11 =
    0.707107 * |5>|a=0>|c0=0>|c1=0>|c=0>
 + 0.707107 * |8>|a=0>|c0=0>|c1=0>|c=1>
(6 + 3) % 11 =
    0.707107 * |6>|a=0>|c0=0>|c1=0>|c=0>
 + 0.707107 * |9>|a=0>|c0=0>|c1=0>|c=1>
(7 + 3) % 11 =
    0.707107 * |7>|a=0>|c0=0>|c1=0>|c=0>
 + 0.707107 * |10>|a=0>|c0=0>|c1=0>|c=1>
(8 + 3) % 11 =
    0.707107 * |8>|a=0>|c0=0>|c1=0>|c=0>
 + 0.707107 * |0>|a=0>|c0=0>|c1=0>|c=1>
(9 + 3) % 11 =
    0.707107 * |9>|a=0>|c0=0>|c1=0>|c=0>
 + 0.707107 * |1>|a=0>|c0=0>|c1=0>|c=1>
(10 + 3) % 11 =
    0.707107 * |10>|a=0>|c0=0>|c1=0>|c=0>
 + 0.707107 * |2>|a=0>|c0=0>|c1=0>|c=1>
(11 + 3) % 11 =
    0.707107 * |11>|a=0>|c0=0>|c1=0>|c=0>
 + 0.707107 * |11>|a=0>|c0=0>|c1=0>|c=1>
(12 + 3) % 11 =
    0.707107 * |12>|a=0>|c0=0>|c1=0>|c=0>
 + 0.707107 * |12>|a=0>|c0=0>|c1=0>|c=1>
(13 + 3) % 11 =
    0.707107 * |13>|a=0>|c0=0>|c1=0>|c=0>
 + 0.707107 * |13>|a=0>|c0=0>|c1=0>|c=1>
(14 + 3) % 11 =
    0.707107 * |14>|a=0>|c0=0>|c1=0>|c=0>
 + 0.707107 * |14>|a=0>|c0=0>|c1=0>|c=1>
(15 + 3) % 11 =
    0.707107 * |15>|a=0>|c0=0>|c1=0>|c=0>
 + 0.707107 * |15>|a=0>|c0=0>|c1=0>|c=1>
```


Modulo n multiply accumulator

In [10]:
```
def bits(y,N):
    return [ (y//2**j) % 2 for j in range(N) ]

def ci(x,n,N):
    b=bits(x,N)
    return [ sum([ b[j]*2**(i+j) for j in range(N) ]) % n for i in range(N) ]

# Now mod multiply
def modmultacc(x,n,op,ybits,tbits,abits):
    assert(len(abits)==3)
    Nbits=len(ybits)
    assert(len(tbits)==len(ybits))
    c=ci(x,n,2**Nbits)
    for i in range(Nbits):
        op=cmodadd(ybits[i],c[i],op,tbits,n,abits)
    return op

par_ybits=[0,1,2]
par_tbits=[3,4,5]
par_abits=[6,7,8]

st=sqc.state(9,basis = ["|%d>|%d>|a=%d>" %
                        (pr(i,0,3),pr(i,3,3),pr(i,6,3)) for i in range(2**9)])
st=sqc.operator(9).X(1)*st

print("Start with\n",st)

op=modmultacc(3,5,sqc.operator(9),ybits=par_ybits,tbits=par_tbits,abits=par_abits)
st=op*st
print("Mult by 3 mod 5\n",st)
```

```
Start with
    1 * |2>|0>|a=0>
Mult by 3 mod 5
    1 * |2>|1>|a=0>
```


In [11]:
```
st=op*st
print("Mult by 3 mod 5\n",st)
```

```
Mult by 3 mod 5
    1 * |2>|2>|a=0>
```

Modulo n multiply

In [12]:
```
def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, y, x = egcd(b % a, a)
        return (g, x - (b // a) * y, y)

def modinv(a, m): # calculate modular inverse using the greatest common divisor
    g, x, y = egcd(a, m)
    if g != 1:
        raise Exception('modular inverse does not exist')
    else:
        return x % m
```

In [15]:
```
def modmult(x,n,op,ybits,abits):
    assert(len(abits)==3+len(ybits))
    tbits=abits[:-3]
    abits=abits[-3:]
    Nbits=len(ybits)
    nix=n - modinv(x,n)

    op=modmultacc(x,n,op,ybits,tbits=tbits,abits=abits)
    for i in range(Nbits):
        op=SWAP(ybits[i],tbits[i],op)

    op=modmultacc(nix,n,op,ybits,tbits=tbits,abits=abits)
    return op

st=sqc.state(9,basis = ["|%d>|%d>|a=%d>" %
                        (pr(i,0,3),pr(i,3,3),pr(i,6,3)) for i in range(2**9)])
st=sqc.operator(9).X(1)*st
print("Start with\n",st)

par_abits=[3,4,5,6,7,8]

op=modmult(3,5,sqc.operator(9),ybits=par_ybits,abits=par_abits)
st=op*st
print("Mult by 3 mod 5\n",st)
```

```
Start with
    1 * |2>|0>|a=0>
Mult by 3 mod 5
    1 * |1>|0>|a=0>
```

In [16]:
```
st=op*st
print("Mult by 3 mod 5\n",st)
```

```
Mult by 3 mod 5
    1 * |3>|0>|a=0>
```


In [17]:
```
st=op*st
print("Mult by 3 mod 5\n",st)
```

```
Mult by 3 mod 5
    1 * |4>|0>|a=0>
```

In [18]:
```
st=op*st
print("Mult by 3 mod 5\n",st)
```

```
Mult by 3 mod 5
    1 * |2>|0>|a=0>
```

Controlled modulo n multiplication

In [20]:
```
def cmodmultacc(x,n,op,ybits,tbits,abits,cbit):
    assert(len(abits)==4)

    tmpbit=abits[3]

    Nbits=len(ybits)
    c=ci(x,n,2**Nbits)
    for i in range(Nbits):
        op=C2NOT(ybits[i],cbit,tmpbit,op)
        op=cmodadd(tmpbit,c[i],op,tbits,n,abits[0:3])
        op=C2NOT(ybits[i],cbit,tmpbit,op)
    return op

def cmodmult(x,n,op,ybits,abits,cbit):
    assert(len(abits)==4+len(ybits))
    tbits=abits[:-4]
    abits=abits[-4:]
    Nbits=len(ybits)
    nix=n - modinv(x,n)
```

```
    nix = modinv(x,n)

    op=cmodmultacc(x,n,op,ybits,tbits=tbits,abits=abits,cbit=cbit)
    for i in range(Nbits):
        op=CSWAP(cbit,ybits[i],tbits[i],op)

    op=cmodmultacc(nix,n,op,ybits,tbits=tbits,abits=abits,cbit=cbit)
    return op


par_ybits=[0,1,2]
par_abits=[3,4,5,6,7,8,9]
par_cbit=10

st=sqc.state(11,basis = ["|%d>|a=%d>|c=%d>" %
                         (pr(i,0,3),pr(i,3,7),pr(i,10,1)) for i in range(2**11)])
st=sqc.operator(11).X(1).H(par_cbit)*st
print("Start with\n",st)

op=cmodmult(3,5,sqc.operator(11),ybits=par_ybits,abits=par_abits,cbit=par_cbit)
st=op*st
print("Mult by 3 mod 5\n",st)
```

```
Start with
    0.707107 *  |2>|a=0>|c=0>
 + 0.707107 *  |2>|a=0>|c=1>
Mult by 3 mod 5
    0.707107 *  |2>|a=0>|c=0>
 + 0.707107 *  |1>|a=0>|c=1>
```

In [21]:
```
st=op*st
print("Mult by 3 mod 5\n",st)
```

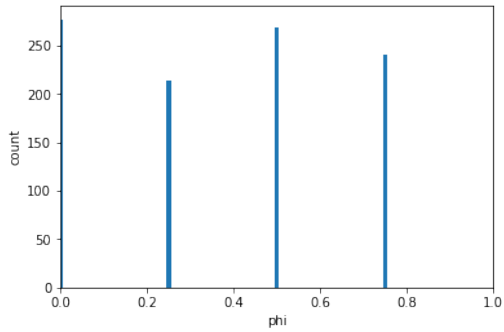
```
Mult by 3 mod 5
    0.707107 *  |2>|a=0>|c=0>
 + 0.707107 *  |3>|a=0>|c=1>
```


Phase estimation

In [46]:
```
par_ybits=[0,1,2]
par_abits=[3,4,5,6,7,8,9]
par_xbits=[10,11,12,13]

st=sqc.state(14,basis = ["|%d>|a=%d>|x=%d>" %
                         (pr(i,0,3),pr(i,3,7),pr(i,10,4)) for i in range(2**14)])
st=sqc.operator(14).X(par_ybits[0])*st
print("Initial state\n",st)
opMM=dict([ (2**j,cmodmult((3**(2**j)) % 5,5,sqc.operator(14),ybits=par_ybits,
                abits=par_abits,cbit=par_xbits[j])) for j in range(len(par_xbits)) ])
```

```
Initial state
    1 *  |1>|a=0>|x=0>
```

In [47]:
```
def cujMM(cbit,j,op):
    return op + opMM[j]

st1=phaseEstimate(sqc.operator(14),par_xbits,cujMM)*st

res=sqc.sample(st1,1000,mask=par_xbits)

plt.bar([ x / 2**len(par_xbits) for x in res.keys() ],res.values(),width=0.01)
plt.xlabel('phi')
plt.xlim(0,1)
plt.ylabel('count')
plt.show()
```



In [48]:
```
for x in sorted(res):
    print(x, res[x])
```

```
0 277
4 214
8 268
12 241
```

In [51]:
```
# Check
for i in range(5):
    print(i,3**i % 5)
```

```
0 1
1 3
2 4
3 2
4 1
```

In [ ]:

1.2.9 Application: Shor's algorithm

- We define the greatest common divisor

$$gcd(a, b) \equiv g \tag{1.139}$$

for positive integers a and b to be the largest integer g that divides both a and b.

- **Shor's algorithm** to find a positive integer larger than 1 that divides an odd, non-prime integer n:

  1. Select a random integer $a \in \{1, \ldots, n-1\}$
  2. Let g=gcd(a,n)
  3. If $g \neq 1$, return g
  4. Find the order r of a modulo n (do this with the quantum algorithm)
  5. If r is odd or $a^{r/2} = -1 \bmod n$, go back to step 1
  6. Let $s_\pm = gcd(a^{r/2} \pm 1, n)$
  7. If s_+ divides n and $s_+ \neq n$ and $s_+ \neq 1$, return s_+
  8. Return s_-

- Example: $n = 91$

  1. Say $a = 87$

  2./3. $g = gcd(a, n) = 1$

  4. $a^6 = 1 \bmod n$, therefore $r = 6$
  5. $a^3 \bmod n = 27 \neq -1 \bmod n$

  6./7. Let $s_+ = gcd(a^3 + 1, n) = 7$. Found a factor!

- One can show that the probability of returning to step 1 in step 5 of the algorithm is bound from above by $1/2^m$, where m is the number of different prime factors in n.

- Once we are in step 6, we know that $x \equiv a^{r/2} \bmod n$ is a non-trivial solution to $x^2 = 1 \bmod n$ with $1 < x < n-1$. The lower bound is guaranteed by the definition of r as being the order of a modulo n, i.e., the smallest number satisfying $a^r = 1 \bmod n$.

- Since $x^2 = 1 \bmod n$, we know that n divides $x^2 - 1 = (x-1)(x+1)$, i.e., n has a common factor with $x - 1$ or $x + 1$ (or generally both). Since $1 < x < n-1$, we have $0 < x - 1 < x + 1 < n$ and therefore the common factor cannot be n itself. So s_+ or s_- (or both) are desired factors.

- Note that in the construction of the modular multiplication gate in the last lecture, we have used the multiplicative inverse of a modulo n. Since gcd(a,n)=1, we are guaranteed that it exists. (This follows from Bézout's lemma.)

- Combined with a primality test, such as the AKS test that can be implemented with cost of $\mathrm{O}(\log^6 n)$, the described algorithm can be used to factor an arbitrary integer into all prime number factors with logarithmic cost in n.

In [404]:
```
import random
```

In [405]:
```
def gcd(a, b):
    return gcd(b, a % b) if b != 0 else a

def findOrder(x,n):
    x0=x % n
    for r in range(1,n):
        if x0 % n == 1:
            return r
        x0=x0 * x % n
    assert(0)
```

In [406]:
```
def findFactor(n):
    for l in range(10):
        a=random.randrange(1,n)
        print("Step 1 (%d): picked random a=%d" % (l,a))
        g=gcd(a,n)
        print("Step 2: gcd(a,n) = %d" % g)
        if g != 1:
            return g
        r=findOrder(a,n)
        print("Step 3: order a^%d == 1 mod n" % r)
        if r % 2 == 0 and a**(r//2) % n != n-1:
            print("Step 4: test if r is even and != -1 mod n")
            break
    if l == 9:
        print("Could not find a factor, %d is very likely prime" % n)
        return(1)
    s0=gcd(a**(r//2)+1,n)
    if n % s0 == 0 and n != s0 and 1 != s0:
        return s0
    return gcd(a**(r//2)-1,n)
```

In [407]:
```
random.seed(19)
print("Factor found: ",findFactor(91))

print("----")

print("Factor found: ",findFactor(15))

print("----")
print("Factor found: ",findFactor(511))
```

```
Step 1 (0): picked random a=87
Step 2: gcd(a,n) = 1
Step 3: order a^6 == 1 mod n
Step 4: test if r is even and != -1 mod n
Factor found:  7
----
Step 1 (0): picked random a=1
Step 2: gcd(a,n) = 1
Step 3: order a^1 == 1 mod n
Step 1 (1): picked random a=13
Step 2: gcd(a,n) = 1
Step 3: order a^4 == 1 mod n
Step 4: test if r is even and != -1 mod n
Factor found:  5
----
Step 1 (0): picked random a=459
Step 2: gcd(a,n) = 1
Step 3: order a^24 == 1 mod n
Step 4: test if r is even and != -1 mod n
Factor found:  73
```

1.2.10 Parenthesis: RSA encryption

- Consider a message of at most N bits, i.e., a number $m \in \{0, 1, \ldots, 2^N - 1\}$.
- We define a public-key encryption system through a method to cheaply create a random pair of public and private keys K_{pub} and K_{priv} that satisfy

$$D(E(m, K_{\text{pub}}), K_{\text{priv}}) = m \tag{1.140}$$

  for computationally cheap functions D and E for any m. At the same time, E shall be computationally prohibitively expensive to invert without knowledge of K_{priv}.
- Here we discuss one of the first and most popular implementations of this idea: the RSA (Rivest–Shamir–Adleman) cryptosystem.
- RSA enjoys widespread use, e.g., as a main protocol in SSL/TLS which encrypts internet traffic sent over HTTPS.
- Key generation:
  1. Select two large prime numbers p and q and let $n = pq$
  2. Then $\varphi(n) \equiv (p-1)(q-1)$
  3. Let e be a random (small) odd integer with $0 \leq e < \varphi(n)$ with $gcd(\varphi(n), e) = 1$
  4. Let d be the multiplicative inverse of e modulo $\varphi(n)$
  5. Let $K_{\text{pub}} = (e, n)$ and $K_{\text{priv}} = (d, n)$
- Encryption and Decryption:

$$E(m, K_{\text{pub}}) \equiv m^e \bmod n\,, \tag{1.141}$$

$$D(c, K_{\text{priv}}) \equiv c^d \bmod n\,. \tag{1.142}$$

- To show that

$$D(E(m, K_{\text{pub}}), K_{\text{priv}}) = m \tag{1.143}$$

  for any $m \in \{0, 1, \ldots, 2^N - 1\}$, we need three basic results of number theory that we state without proof:
  1. **Fermat's little theorem:** Given a prime number p and any integer a, we have $a^p = a \bmod p$.
  2. **Euler's generalization of Fermat's little theorem:** Given integers a and n with $n \equiv \prod_{j=1}^k p_j^{\alpha_j}$ with prime numbers p_j and positive integers α_j and $gcd(a, n) = 1$, we have $a^{\varphi(n)} = 1 \bmod n$ for $\varphi(n) \equiv \prod_{j=1}^k p_j^{\alpha_j - 1}(p_j - 1)$.

3. **Chinese remainder theorem:** Suppose $m_1, \ldots, m_n$ are positive integers such that any $\gcd(m_i, m_j) = 1$ for all $i \neq j$. Then the system of equations

$$x = a_1 \bmod m_1\,, \tag{1.144}$$
$$x = a_2 \bmod m_2\,, \tag{1.145}$$
$$\ldots \tag{1.146}$$
$$x = a_n \bmod m_n \tag{1.147}$$

with integer a_i has a solution that is unique modulo $M = \prod_{i=1}^{n} m_i$.

- We first write

$$D(E(m, K_{\text{pub}}), K_{\text{priv}}) = m^{ed} \bmod n = m^{1+k\varphi(n)} \bmod n \tag{1.148}$$

with proper integer k.

We first consider the case $gcd(m, n) = 1$. Then Euler's generalization of Fermat's little theorem yields $(m^k)^{\varphi(n)} = 1 \bmod n$ and therefore

$$D(E(m, K_{\text{pub}}), K_{\text{priv}}) = m \bmod n = m\,. \tag{1.149}$$

If $gcd(m, n) \neq 1$, then one of p and q divides m. Without loss of generality, assume p divides m. Then

$$m^{ed} = 0 \bmod p = m \bmod p\,. \tag{1.150}$$

Because q does not divide m, $gcd(q, m) = 1$ and therefore $m^{-1} \bmod q$ exists and Fermat's little theorem yields $m^{q-1} = 1 \bmod q$. Therefore also $m^{\varphi(n)} = (m^{q-1})^{p-1} = 1 \bmod q$. Therefore also

$$m^{ed} = m^{1+k\varphi(n)} = m \bmod q\,. \tag{1.151}$$

The Chinese remainder theorem then yields that $m^{ed} = m \bmod n$ and also in this case

$$D(E(m, K_{\text{pub}}), K_{\text{priv}}) = m \bmod n = m\,. \tag{1.152}$$

- Now assume that we can implement Shor's algorithm using a quantum computer and therefore efficiently factor n, which is part of K_{pub}. We can then learn p and q and from e (also part of K_{pub}) cheaply learn d and therefore K_{priv}. Once we can properly run Shor's algorithm on real quantum hardware with, RSA is broken!

In [4]:
```
import random
```

In [5]:
```
def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, y, x = egcd(b % a, a)
        return (g, x - (b // a) * y, y)

def modinv(a, m): # calculate modular inverse using the greatest common
    g, x, y = egcd(a, m)
    if g != 1:
        raise Exception('modular inverse does not exist')
    else:
        return x % m
```

In [9]:
```
def make_rsa_keys(p,q,emax): # p and q are large prime numbers
    n=p*q
    phi=(p-1)*(q-1)
    while True:
        e=1+2*random.randrange(0,emax//2)
        if egcd(phi,e)[0]==1: # phi and e need to be co-prime
            break
    d=modinv(e,phi)
    return( (e,n), (d,n) )

#(pub_key, priv_key)=make_rsa_keys(337,683,100)
(pub_key, priv_key)=make_rsa_keys(20381027,2147483647,1000)

print("pub_key",pub_key)
print("priv_key",priv_key)
```

```
pub_key (61, 43767922191565469)
priv_key (30852796082280889, 43767922191565469)
```

In [10]:
```
def crypt(key, message):
    assert(message >= 0 and message < key[1])

    #return message ** key[0] % key[1]

    # Much faster implementation below
    if key[0] == 0:
        return 1
    elif key[0] % 2 == 0:
        return crypt((key[0]//2,key[1]),message ** 2 % key[1])
    else:
        return (crypt((key[0]-1,key[1]),message) * message) % key[1]

encoded=crypt(pub_key,1231)
print(encoded)
print(crypt(priv_key,encoded))
```

```
28175087500667783
1231
```

In [12]:
```
alphabet=[' '] + [chr(ord('a')+i) for i in range(26)]
print(alphabet)

def str2int(st,alphabet):
    N=len(alphabet)
    r=0
    for s in st:
        r=N*r + alphabet.index(s)
    return r

def int2str(i,alphabet):
    N=len(alphabet)
    r=""
    while i > 0:
        r=alphabet[i % N]+r
        i=i//N
    return r
```

```
[' ', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']
```

In [14]:
```
encstr=int2str(crypt(pub_key,str2int("hello world",alphabet)),alphabet)
print(encstr)
print(int2str(crypt(priv_key,str2int(encstr,alphabet)),alphabet))
```

```
nhuzumhakrp
hello world
```

Chapter 2

Real quantum computing

2.1 Real quantum hardware

2.1.1 Characteristics of real quantum computers

- A single qubit is typically mapped to a setup that can be approximated by a two-level system. The effects of other states must be largely suppressed.
- Let us assume that those two states, $|0\rangle$ and $|1\rangle$, are eigenstates of the Hamiltonian, with energies E_0 and E_1 and $\Delta E = E_1 - E_0 > 0$.
- Through interactions with the environment, the excited state $|1\rangle$ will relax to the ground state $|0\rangle$ with an average **energy relaxation time** T_1. With constant decay rate, the population N_1 of the excited state will deplete exponentially as

$$N_1(t) = N_1(0)e^{-t/T_1} . \tag{2.1}$$

A good qubit therefore needs large T_1. T_1 is sometimes also called **spin relaxation time** or **longitudinal relaxation time**.

- The time-evolution of a general superposition

$$|\psi(t=0)\rangle = c_0 |0\rangle + c_1 |1\rangle \tag{2.2}$$

is given by

$$|\psi(t)\rangle = e^{-iE_0 t} \left[c_0 |0\rangle + e^{-i\Delta E t} c_1 |1\rangle\right] , \tag{2.3}$$

where we have used the convention of $\hbar = 1$.

- Since an overall phase is not observable, time evolution has the same effect as the R_ϕ gate. Also, without loss of generality, we may restrict the discussion to $E_0 = 0$.

- In a digital quantum computer, the phase generated from time evolution must be hidden from the user.
- In practice, the time-evolution is better described by a t-dependent energy

$$|\psi(t)\rangle = c_0 |0\rangle + e^{-i \int_0^t d\tau \Delta E(\tau)} c_1 |1\rangle \tag{2.4}$$

with a time-dependent $\Delta E(\tau)$. Such time-dependence can, e.g., originate from irregularities in a magnetic field.

The deviation from $\Delta E(\tau)$=const is often referred to as decoherence and two common measures are defined:

1. The Ramsey experiment to determine the decoherence time T_2^*: We start with a $|0\rangle$ state and apply the Hadamard, leaving the system in state

$$\frac{1}{\sqrt{2}} |0\rangle + \frac{1}{\sqrt{2}} |1\rangle \ . \tag{2.5}$$

We then let the system evolve in time by t yielding

$$\frac{1}{\sqrt{2}} |0\rangle + \frac{1}{\sqrt{2}} e^{-i \int_0^t d\tau \Delta E(\tau)} |1\rangle \ . \tag{2.6}$$

Then we apply another Hadamard gate yielding

$$\frac{1}{2} \left(1 + e^{-i \int_0^t d\tau \Delta E(\tau)}\right) |0\rangle + \frac{1}{2} \left(1 - e^{-i \int_0^t d\tau \Delta E(\tau)}\right) |1\rangle \ . \tag{2.7}$$

Then we measure the qubit and find 0 with probability

$$P_0(t) = \frac{1}{4} |1 + e^{-i \int_0^t d\tau \Delta E(\tau)}|^2 = \frac{1}{2} + \frac{1}{2} \cos\left(\int_0^t d\tau \Delta E(\tau)\right) \ . \tag{2.8}$$

In an ideal system with no decoherence and $\Delta E(\tau) \equiv \Delta E$, we would find

$$P_0^{\text{ideal}}(t) = \frac{1}{2} + \frac{1}{2} \cos(t\Delta E) \ . \tag{2.9}$$

In a fully decohered system, we would instead average over random phases $\phi \equiv \int_0^t d\tau \Delta E(\tau)$, yielding

$$P_0^{\text{decoh}} = \frac{1}{2\pi} \int_0^{2\pi} d\phi \left[\frac{1}{2} + \frac{1}{2} \cos(\phi)\right] = \frac{1}{2} \, . \tag{2.10}$$

In many cases, the decoherence effects are well described by an exponential onset

$$P_0^{\text{real}}(t) = \frac{1}{2} + \frac{1}{2} \cos(t\Delta E) \, e^{-t/T_2^*} \tag{2.11}$$

defining the Ramsey decoherence time T_2^*.

2. The Hahn echo decoherence time T_2: We start with a $|0\rangle$ state and apply the Hadamard, leaving the system in state

$$\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle \,. \tag{2.12}$$

We then let the system evolve in time by $t/2$ yielding

$$\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}e^{-i\int_0^{t/2} d\tau \Delta E(\tau)}|1\rangle \,. \tag{2.13}$$

We then apply a NOT gate to find

$$\frac{1}{\sqrt{2}}|1\rangle + \frac{1}{\sqrt{2}}e^{-i\int_0^{t/2} d\tau \Delta E(\tau)}|0\rangle \,. \tag{2.14}$$

We then wait another $t/2$ yielding

$$\frac{1}{\sqrt{2}}e^{-i\phi_1}|1\rangle + \frac{1}{\sqrt{2}}e^{-i\phi_0}|0\rangle \,. \tag{2.15}$$

with

$$\phi_1 = \int_{t/2}^{t} d\tau \Delta E(\tau)\,, \tag{2.16}$$

$$\phi_0 = \int_{0}^{t/2} d\tau \Delta E(\tau)\,. \tag{2.17}$$

Finally we apply a Hadamard and find

$$\frac{1}{2}e^{-i\phi_0}\left(1 + e^{-i(\phi_1-\phi_0)}\right)|0\rangle + \frac{1}{2}e^{-i\phi_0}\left(1 - e^{-i(\phi_1-\phi_0)}\right)|1\rangle \,. \tag{2.18}$$

Measuring in this system yields 0 with

$$P_0(t) = \frac{1}{4}\left|1 + e^{-i(\phi_1-\phi_0)}\right|^2 = \frac{1}{2} + \frac{1}{2}\cos(\phi_1 - \phi_0) \,. \tag{2.19}$$

In an ideal system with no decoherence we have $\Delta E(\tau) \equiv \Delta E$ and thus $\phi_1 = \phi_0$ and

$$P_0^{\text{ideal}}(t) = 1\,. \tag{2.20}$$

In a fully decohered system, we have a random $\phi_1 - \phi_0$ and therefore measure

$$P_0^{\text{decoh}}(t) = \frac{1}{2\pi}\int_0^{2\pi} d\phi \left[\frac{1}{2} + \frac{1}{2}\cos(\phi)\right] = \frac{1}{2}\,. \tag{2.21}$$

In many cases, the decoherence effects are again well described by an exponential onset

$$P_0^{\text{real}}(t) = \frac{1}{2} + \frac{1}{2}e^{-t/T_2} \tag{2.22}$$

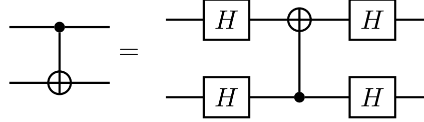
defining the decoherence time T_2.

- The system needs to have a reliable way to prepare the initial state $|0\rangle$.
- The system needs a reliable readout procedure to measure the qubits. If a qubit returns an incorrect bit with probability ε_m, we call ε_m the measurement error.
- The application of a universal set of gates must be defined, e.g., R_ϕ, H, and CNOT.
- The application of such gates must yield only small errors. To this end, gate errors are typically defined using randomized benchmarks. The idea is to apply a random set of n gates and their inverse gate(s) to the initial state of the system $|0\rangle$. We then measure the ground state with probability

$$P_0 = (1-\varepsilon)^n \tag{2.23}$$

  with gate error ε. There are many variations of this protocol including errors for individual gates.
- The system must be sufficiently connected that a CNOT between all qubits can be implemented, given a sufficient number of swap gates.

  On occasion CNOTs are only implemented in one direction and the identity



  is used to implement the reverse gate.

2.1.2 Survery of universal digital quantum computers (2020)

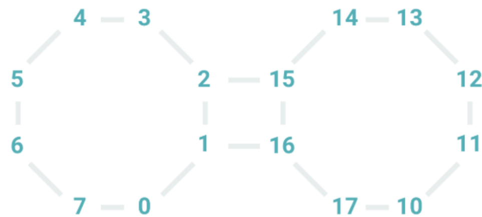
- As of summer of 2020, there are two publicly available universal digital quantum computer vendors: Rigetti and IBM
- Both machines use a variation of charge qubits called transmon qubits that we will describe later
- Rigetti (www.rigetti.com): In summer of 2019 offered a 16 qubit machine with

$$T_1 = 25\mu s\,, \qquad T_2 = 20\mu s\,, \tag{2.24}$$

$$\varepsilon_m = 0.07\,, \qquad \varepsilon_{1-\text{qubit gate, RB}} = 0.045\,, \tag{2.25}$$

$$\varepsilon_{2-\text{qubit}-\text{gate, RB}} = 0.1\,. \tag{2.26}$$

  The topology of the qubits (with lines denoting allowed two-qubit gates) is:



These are 16 qubits with an odd numbering

In summer of 2020, 31 qubit machine with

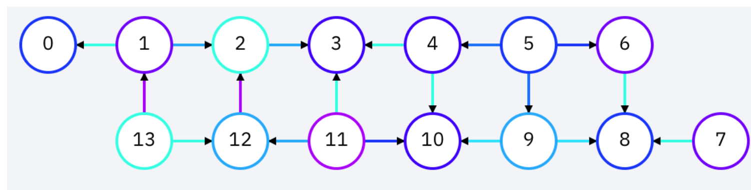
$$T_1 = 29\mu s\,, \qquad T_2 = 18\mu s\,, \tag{2.27}$$
$$\varepsilon_m = ?\,, \qquad \varepsilon_{1-\text{qubit gate, RB}} = 0.0021\,, \tag{2.28}$$
$$\varepsilon_{2-\text{qubit}-\text{gate, RB}} = 0.0434\,. \tag{2.29}$$

- IBM (quantum-computing.ibm.com) offers the following machines to the public:

1. IBM Q Melbourne with 14 qubits

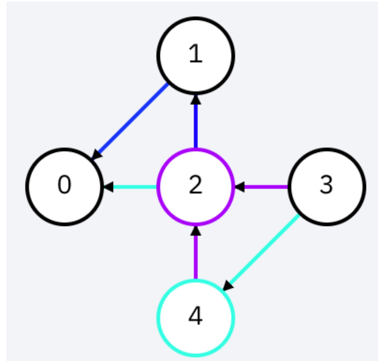
$$T_1 = 50\mu s\,, \qquad T_2 = 23\mu s\,, \tag{2.30}$$
$$\varepsilon_m = 0.026\,, \qquad \varepsilon_{\text{gate, RB}} = 0.0025\,. \tag{2.31}$$

The topology of the qubits (with arrows from A to B allowing a CNOT with control bit A and target bit B) is:



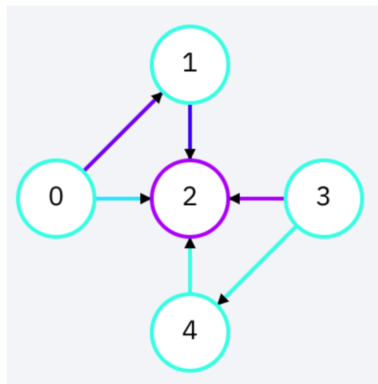
2. IBM Q Tenerife with 5 qubits

$$T_1 = 50\mu s\,, \qquad T_2 = 21\mu s\,, \tag{2.32}$$
$$\varepsilon_m = 0.047\,, \qquad \varepsilon_{\text{gate, RB}} = 0.0006\,. \tag{2.33}$$



3. IBM Q Yorktown with 5 qubits

$$T_1 = 42\mu s\,, \qquad T_2 = 37\mu s\,, \tag{2.34}$$
$$\varepsilon_m = 0.056\,, \qquad \varepsilon_{\text{gate, RB}} = 0.00036\,. \tag{2.35}$$

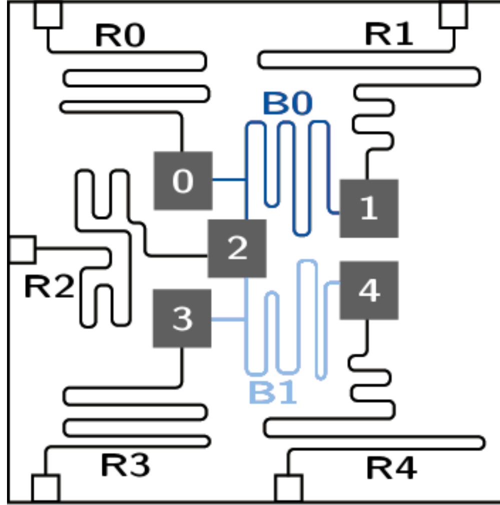


With the approximation that we can apply on the order of $1/\varepsilon_{\text{gate}}$ gates in practice, these systems can apply on the order of 400, 1600, and 2800 gates until errors dominate.

- Give tour of IBM Q Experience, toQASM function, run Deutsch-Jozsa

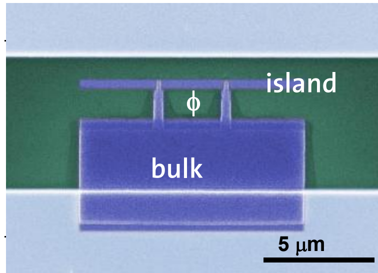
2.1.3 Example of hardware implementation: IBM 5 qubit systems

- Schemeatic of IBM 5 qubit systems



There are five qubits (squares 0 to 4), five read-out resonators (R0 to R4), and two bus resonators (B0 and B1)

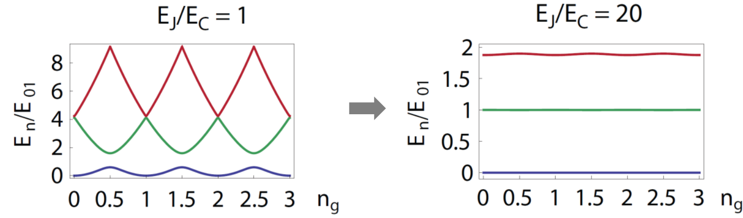
- The qubits are an improved version of a so-called Cooper pair box called transmon qubit.
- The Cooper pair box is a superconducting island connected via Josephson junctions (two Al superconductors coupled through a thin AlOx insulator) to a bulk reservoir:



- In the superconducting island pairs of electrons form so-called bosonic Cooper pairs which can tunnel through the Josephson junction.
- There are two energy scales: the Josephson coupling energy E_J, and the charging energy E_C (the energy to add another Cooper pair)
- The number of Cooper pairs transferred to the island (island minus bulk) is N and with the offset charge N_g the Hamiltonian can be written as

$$H = \sum_{N=-\infty}^{\infty} \left[E_C (N - N_g)^2 |N\rangle \langle N| - \frac{E_J}{2} (|N\rangle \langle N+1| + |N+1\rangle \langle N|) \right] . \tag{2.36}$$

The transmon version of this qubit has large E_J/E_C, which leads to stability against N_g fluctuations:



Use lowest two states for qubit. The energy difference between these states shall be $\hbar\omega_i$ for qubit i.

- Each qubit has slightly different resonance frequencies. For the IBM Tenerife system:

Qubit	$\omega_i/2\pi$ (GHz)
Q0	5.2464
Q1	5.2983
Q2	5.3383
Q3	5.4261
Q4	5.1745

  These frequencies are all in the microwave region.

- The read-out resonators are connected to a single qubit and can inject a microwave tone to the qubits and infer the qubit state from the measured response; such microwave impulses also allow for the effective implementation of the Hadamard gate.

- The R_ϕ gate is implemented virtually, by adjusting in software the accumulated phase of the $|1\rangle$ states.

- The bus resonators use a so-called cross-resonance technique to implement the CNOT between the connected qubits. The idea is to send a microwave tone to the bus B0 or B1 appropriately chosen for the control and target qubits which can be shown to create the desired entanglement between the respective qubits. The details of the process are not symmetric, so the CNOT can only be implemented directly in one direction.

2.2 Formal treatment of quantum noise

Before discussing how to model and then correct noise in a real quantum computer, it is useful to introduce additional notation.

2.2.1 Principal system and environment

- In a real quantum computer, the desired state space S^N (the principal system) is coupled to an environment E such that the total Hilbert space $\mathcal{H}$ is given by

$$\mathcal{H} \equiv S^N \otimes E \,. \tag{2.37}$$

- The principal system and environment interact with each other.
- Without loss of generality, we can embed E in a S^M such that we often will restrict the discussion to $E = S^M$.
- In general, the result of such interactions cannot be written as a product of a vector in S^N and S^M (entanglement), such that the physical state is not well described by an element of S^N by itself. We will, however, see that it may be desribed by a statistical ensemble of states in S^N.
- Example (Bell state):

  Consider 1-qubit principal system (least-significant bit) and a 1-qubit environment (most-significant bit) prepared in the state (color and arrow)

$$|\Psi_{\text{physical}}\rangle \equiv \frac{1}{\sqrt{2}} \left(|00\rangle + |11\rangle\right) \,. \tag{2.38}$$

  Let us demonstrate that even if we restrict measurements and gates to the principal system, there is no state $|\Psi_{\text{principal}}\rangle$ in S^1 that is indistinguishable from $|\Psi_{\text{physical}}\rangle$.

  We first note that measuring the principal qubit in state $|0\rangle$ and $|1\rangle$ will have identical probability $1/2$ such that any candidate state in S^1 needs to be of the form

$$|\Psi_{\text{principal}}\rangle \equiv \frac{1}{\sqrt{2}} \left(|0\rangle + e^{i\phi} |1\rangle\right) \tag{2.39}$$

  with real ϕ.

  If we now apply the Hadamard to the principal qubit 0, we find

$$H^{(0)} |\Psi_{\text{physical}}\rangle = \frac{1}{2} \left(|00\rangle + |01\rangle + |10\rangle - |11\rangle\right) \,, \tag{2.40}$$

$$H^{(0)} |\Psi_{\text{principal}}\rangle = \frac{1}{2} \left((1 + e^{i\phi}) |0\rangle + (1 - e^{i\phi}) |1\rangle\right) \tag{2.41}$$

such that in the full physical system, we would still measure the principal qubit in state $|0\rangle$ and $|1\rangle$ with probability of $1/2$ but for $|\Psi_{\text{principal}}\rangle$, we would measure 0 and 1 with probability

$$P_0 = \frac{1}{2}\left(1+\cos(\phi)\right) , \qquad P_1 = \frac{1}{2}\left(1-\cos(\phi)\right) . \tag{2.42}$$

So only $\phi = \frac{\pi}{2}$ is still allowed.

If we now apply $R_{\pi/2}$ and then another Hadamard to the principal qubit, we find

$$H^{(0)}R^{(0)}_{\pi/2}H^{(0)}\left|\Psi_{\text{physical}}\right\rangle = \frac{1}{\sqrt{2}^3}\left((1+i)\left|00\right\rangle + (1-i)\left|01\right\rangle + (1-i)\left|10\right\rangle + (1+i)\left|11\right\rangle\right) , \tag{2.43}$$

$$H^{(0)}R^{(0)}_{\pi/2}H^{(0)}\left|\Psi_{\text{principal}}\right\rangle \overset{\phi=\pi/2}{=} \frac{1}{\sqrt{2}}(1+i)\left|0\right\rangle . \tag{2.44}$$

In the physical system the principal qubits are still equally likely, however, in state $|\Psi_{\text{principal}}\rangle$ we would always measure 0.

There is no state in the principal system that captures fully the information in the physical system even if we only measure and apply gates in the principal system.

2.2.2 Mixed states, density matrix, and quantum operations

- As demonstrated above, a principal system S^N coupled to its environment cannot be adequately described by a single state in S^N.
- We will show below that a better description is a statistical ensemble of states in S^N.
- For such a statistical ensemble of states $|n\rangle \in S^N$ with probability p_n, we define the density matrix

$$\rho \equiv \sum_n p_n \left|n\right\rangle \left\langle n\right| . \tag{2.45}$$

If the ensemble only has one state, we say that this matrix describes a **pure state** and in any other case say that it describes a **mixed state**.

It is common to refer to the matrix ρ as the state of the total system.

- This matrix has the property

$$\mathrm{Tr}\left[\rho\right] = \sum_{n,i} p_n \langle i|n\rangle\langle n|i\rangle = \sum_n p_n \langle n| \left(\sum_i |i\rangle\langle i|\right) |n\rangle = \sum_n \langle n|n\rangle p_n = \sum_n p_n = 1 , \tag{2.46}$$

where the sum over i is over a full basis.

We can also show that $\rho^2 = \rho$ if and only if it describes a pure state.

- The expectation value with an arbitrary state $|\psi\rangle$ is

$$\langle\psi|\,\rho\,|\psi\rangle = \sum_n p_n \langle\psi|n\rangle\langle n|\psi\rangle = \sum_n p_n |\langle n|\psi\rangle|^2 \geq 0\,. \tag{2.47}$$

- A map $\mathcal{E}$ with

$$\rho' \equiv \mathcal{E}(\rho) \tag{2.48}$$

is called a **quantum operation**. We will call a quantum operation that preserves the trace a **quantum channel**.

- Applying a unitary matrix to the system is given by

$$\mathcal{E}_U(\rho) \equiv \sum_n p_n U\,|n\rangle\,\langle n|\,U^\dagger\,. \tag{2.49}$$

- We define non-unitary projection matrices $P_{i,\bar{b}_i}$ through

$$P_{i,\bar{b}_i}\,|b_{N-1}\cdots b_0\rangle = \delta_{\bar{b}_i,b_i}\,|b_{N-1}\cdots b_0\rangle\ , \tag{2.50}$$

with bits $b_i \in \{0,1\}$ and $0 \leq i < N$ and a corresponding quantum operation

$$\mathcal{E}_{P_{i,\bar{b}_i}}(\rho) \equiv P_{i,\bar{b}_i}\rho P^\dagger_{i,\bar{b}_i}\,. \tag{2.51}$$

- Then measuring the qubit i returns a value $\bar{b}_i$ with probability

$$\overline{P}_{i,\bar{b}_i}(\rho) \equiv \mathrm{Tr}\left[\mathcal{E}_{P_{i,\bar{b}_i}}(\rho)\right] \tag{2.52}$$

and after measuring this value, the system is in state

$$\frac{\mathcal{E}_{P_{i,\bar{b}_i}}(\rho)}{\mathrm{Tr}\left[\mathcal{E}_{P_{i,\bar{b}_i}}(\rho)\right]}\,. \tag{2.53}$$

- Let the qubits of S^N be partitioned in sets A and B, then we can define a reduced

$$\rho^A \equiv \mathrm{Tr}_B[\rho] = \sum_{b\in B}\langle b|\rho|b\rangle \tag{2.54}$$

with sum over a basis in B such that for a general matrix $M = M_A \otimes 1_B$, where the first factor acts on qubits A and the second factor acts on qubits B, we have

$$\mathrm{Tr}_B\left[\mathcal{E}_M(\rho)\right] = \sum_{b\in B}\sum_n p_n M_A\langle b\,|n\rangle\,\langle n|\,b\rangle M_A^\dagger = \mathcal{E}_{M_A}(\rho^A)\,. \tag{2.55}$$

- This yields a procedure to generate the mixed state in the principal system that is indistinguishable from the full physical state by only applying gates in the principal system and measuring in the principal system.

  Consider the following. We first apply in the full physical system ρ an arbitrary gate $U = U^{\text{principal}} \otimes \mathbb{1}$, where the first factor acts on the principal system. We then measure a qubit i of the principal system. The probability of measuring $\bar{b}_i$ is then given by

$$\begin{aligned}
\overline{P}_{i,\bar{b}_i}(\mathcal{E}_U(\rho)) &= \text{Tr}\left[P_{i,\bar{b}_i}(U\rho U^\dagger P^\dagger_{i,\bar{b}_i})\right] && (2.56)\\
&= \text{Tr}_{\text{principal}}\left[\text{Tr}_{\text{environment}}\left[P_{i,\bar{b}_i}(U\rho U^\dagger P^\dagger_{i,\bar{b}_i})\right]\right] && (2.57)\\
&= \text{Tr}_{\text{principal}}\left[P_{i,\bar{b}_i}(U^{\text{principal}}\text{Tr}_{\text{environment}}\left[\rho\right](U^{\text{principal}})^\dagger P^\dagger_{i,\bar{b}_i})\right]\\
&= \text{Tr}_{\text{principal}}\left[\mathcal{E}_{P_{i,\bar{b}_i}}(\mathcal{E}_{U^{\text{principal}}}(\rho^{\text{principal}}))\right] && (2.58)\\
&= \overline{P}_{i,\bar{b}_i}(\mathcal{E}_{U^{\text{principal}}}(\rho^{\text{principal}}))\,. && (2.59)
\end{aligned}$$

  So by construction the state $\rho^{\text{principal}}$ is indistinguishable from ρ^{physical} as long as we only measure and apply gates in the principal system.

- Example: For the Bell-state, discussed above, we have

$$\rho^{\text{physical}} = \frac{1}{2}\left(|00\rangle + |11\rangle\right)\left(\langle 00| + \langle 11|\right) \tag{2.60}$$

  and therefore

$$\begin{aligned}
\rho^{\text{principal}} &= \text{Tr}_{\text{environment}}\left[\rho^{\text{physical}}\right] && (2.61)\\
&= (\langle 0| \otimes \mathbb{1})\rho^{\text{physical}}(|0\rangle \otimes \mathbb{1}) + (\langle 1| \otimes \mathbb{1})\rho^{\text{physical}}(|1\rangle \otimes \mathbb{1}) && (2.62)\\
&= \frac{1}{2}\,|0\rangle\langle 0| + \frac{1}{2}\,|1\rangle\langle 1|\,. && (2.63)
\end{aligned}$$

- Conversely, if we have a mixed state in the principal system, we can extend it by an environment of equal size $E = S^N$ and consider the pure state

$$\left|\Psi^{\text{physical}}\right\rangle = \sum_n \sqrt{p_n}\,|n\rangle\,|n\rangle \tag{2.64}$$

  such that

$$\begin{aligned}
\text{Tr}_E[\left|\Psi^{\text{physical}}\right\rangle\left\langle\Psi^{\text{physical}}\right|] &= \sum_{n,m}\sqrt{p_n p_m}\,|n\rangle\langle m|\,\langle m|\,n\rangle && (2.65)\\
&= \sum_n p_n\,|n\rangle\langle n| = \rho^{\text{principal}}\,. && (2.66)
\end{aligned}$$

  Here we have assumed without loss of generality that the states $|n\rangle$ are orthonormal. (If they are not, we expand them in such a basis and can still write it in the same way.) This procedure is called **purification**.

- Let us now consider a U acting on both principal system and environment. Then we can show that

$$\mathrm{Tr}_{\mathrm{environment}}\left[\mathcal{E}_U(\rho^{\mathrm{physical}})\right] = \sum_k E_k \rho^{\mathrm{principal}} E_k^\dagger \tag{2.67}$$

with E_k acting only on the principal system. We call this the **operator sum representation**. We call E_k the **operation elements** or **Kraus operators**. These operators depend on the state of the environment!

It is in general possible to find different equivalent sets of operation elements, such that the operator sum representation is not unique.

- For a quantum channel the operation elements need to satisfy

$$\mathrm{Tr}\left[\mathcal{E}_U(\rho^{\mathrm{physical}})\right] = \sum_k \mathrm{Tr}\left[E_k \rho^{\mathrm{principal}} E_k^\dagger\right] \tag{2.68}$$

$$= \mathrm{Tr}\left[\rho^{\mathrm{principal}}\left(\sum_k E_k^\dagger E_k\right)\right] \stackrel{!}{=} 1 \tag{2.69}$$

for general $\rho^{\mathrm{principal}}$ such that

$$\sum_k E_k^\dagger E_k \stackrel{!}{=} \mathbb{1}\,. \tag{2.70}$$

2.2.3 Noise channels

- We can use the operator sum representation to represent quantum noise channels involving both the principal system and its environment in terms of the principal system only.

- The phenomenological effects described by T_1 and T_2 as well as the gate errors can on a more fundamental level be described as combinations of individual quantum noise channels. We will give the most important examples below.

- The bit-flip channel, which flips a bit with probability $1-p$, can be written using operation elements

$$E_0 = \sqrt{p}\mathbb{1}\,, \qquad E_1 = \sqrt{1-p}X \tag{2.71}$$

with all matrices acting on the affected qubit. The corresponding noise operation is given by

$$\mathcal{E}(\rho) = p\rho + (1-p)X\rho X\,. \tag{2.72}$$

We can show that $\mathrm{Tr}\mathcal{E}(\rho) = p + (1-p) = 1$.

- The phase-flip channel, which gives the $|1\rangle$ state a relative minus sign with probability $1-p$, can be written using

$$E_0 = \sqrt{p}\mathbb{1}\,, \qquad E_1 = \sqrt{1-p}Z \tag{2.73}$$

- The depolarizing channel single qubit, which moves a pure state closer to a completely mixed state, can be written as

$$\mathcal{E}(\rho) = \frac{1-p}{2}\mathbb{1} + p\rho\,, \tag{2.74}$$

where with probability p, the state remains unchanged. An operator sum representation is given, e.g., by

$$E_0 = \frac{1}{2}\sqrt{1+3p}\mathbb{1}\,, \qquad E_1 = \frac{1}{2}\sqrt{1-p}X\,, \tag{2.75}$$

$$E_2 = \frac{1}{2}\sqrt{1-p}Y\,, \qquad E_3 = \frac{1}{2}\sqrt{1-p}Z\,. \tag{2.76}$$

Similarly for n qubits and $d = 2^n$ a depolarizing channel can be defined with

$$\mathcal{E}(\rho) = \frac{1-p}{d}\mathbb{1} + p\rho\,. \tag{2.77}$$

- The phase decoherence effect can be modelled by applying a random phase to the $|1\rangle$ state, see also problem set 7. For a system in a pure state

$$|\psi\rangle = a\,|0\rangle + b\,|1\rangle \tag{2.78}$$

with $\rho = |\psi\rangle\langle\psi|$ we can write this effect as

$$\mathcal{E}(\rho) = \frac{1}{\sqrt{4\pi\lambda}}\int_{-\infty}^{\infty} d\theta R_\theta\,|\psi\rangle\langle\psi|\,R_\theta^\dagger e^{-\theta^2/(4\lambda)} \tag{2.79}$$

$$= \begin{pmatrix} |a|^2 & ab^*e^{-\lambda} \\ a^*be^{-\lambda} & |b|^2 \end{pmatrix} \tag{2.80}$$

with $\lambda = t/T_2$.

The limiting procedure discussed in problem set 7 can be shown to yield this Gaussian distribution due to the central limit theorem.

The effect of suppressing the off-diagonal terms can be modelled by

$$E_0 = \begin{pmatrix} 1 & 0 \\ 0 & \sqrt{1-p} \end{pmatrix}\,, \qquad E_1 = \begin{pmatrix} 0 & 0 \\ 0 & \sqrt{p} \end{pmatrix} \tag{2.81}$$

with $e^{-\lambda} \overset{!}{=} \sqrt{1-p}$.

As discussed above, the representation is not unique and in fact

$$E_0 = \sqrt{\alpha}\mathbb{1}\,, \qquad E_1 = \sqrt{1-\alpha}Z \tag{2.82}$$

with $\alpha \equiv (1+\sqrt{1-p})/2$ gives an equivalent representation.

Interestingly, this is identical to the phase flip channel. This has the important consequence that **correcting for phase flips is as good as correcting for infinitesimal small phase fluctuations!**

- The energy relaxation process described by T_1 can be described by the amplitude damping operation defined by

$$E_0 = \begin{pmatrix} 1 & 0 \\ 0 & \sqrt{1-\gamma} \end{pmatrix}, \qquad E_1 = \begin{pmatrix} 0 & \sqrt{\gamma} \\ 0 & 0 \end{pmatrix} \tag{2.83}$$

with $\gamma \equiv 1 - e^{-t/T_1}$.

If $\rho = |1\rangle\langle 1|$, then

$$\rho' = \sum_k E_k \rho E_k^\dagger = |0\rangle\langle 0|\,\gamma + |1\rangle\langle 1|\,(1-\gamma) \tag{2.84}$$

such that the excited state population over time is

$$N_1(t) = N_1(0) e^{-t/T_1} . \tag{2.85}$$

2.2.4 Fidelity and trace distance

- Before discussing how to correct errors on a quantum computer, it is useful to quantify the noise introduced by the noise channels. We will use these metrics to show improvement after error correction.

- In general, we will be concerned with measures of distance between the original state ρ and the state $\sigma = \mathcal{E}_U(\rho)$, where U is a quantum channel.

- A simple measure is the **trace distance**

$$D(\rho, \sigma) \equiv \frac{1}{2} \mathrm{Tr}\, |\rho - \sigma| \tag{2.86}$$

with

$$|A| \equiv \sqrt{A^\dagger A} \tag{2.87}$$

and the square root being defined through $\sqrt{A}\sqrt{A} = A$.

- An alternative popular measure is the **fidelity** defined as

$$F(\rho, \sigma) \equiv \mathrm{Tr}\left[\sqrt{\sqrt{\rho}\sigma\sqrt{\rho}}\right] . \tag{2.88}$$

- By definition $F(\rho, \rho) = 1$.

- One can show that

$$F(U\rho U^\dagger, U\sigma U^\dagger) = F(\rho, \sigma) \tag{2.89}$$

for unitary U and density matrices ρ and σ.

- One can also show that $F(\sigma, \rho) = F(\rho, \sigma)$.
- We always have $0 \leq F(\sigma, \rho) \leq 1$.
- In the case of a pure state $\sigma \equiv |\Psi\rangle\langle\Psi|$ one can show that

$$F(\rho, \sigma) = \sqrt{\langle\Psi|\,\rho\,|\Psi\rangle} \tag{2.90}$$

such that the square of the fidelity is given by the overlap between Ψ and ρ.

2.2.5 Simulating quantum noise

- We next discuss how to implement a simple noise model in the simulator (sqc).
- One possibility: replace pure states of N qubits (2^N-dimensional vectors) by density matrices ($2^N \times 2^N$-dimensional matrices). Disadvantage: large memory and computation requirements for large N.
- Alternative: use interpretation of the density matrix as statistical ensemble of pure states and extend this to quantum operations.
- Let the system start in a pure state $\rho = |\Psi\rangle\langle\Psi|$ and let us apply the quantum channel

$$\mathcal{E}(\rho) = \sum_k E_k \rho E_k^\dagger \tag{2.91}$$

with Kraus operators E_k. Then, we define

$$|\Psi'_k\rangle \equiv \frac{1}{\sqrt{p_k}} E_k |\Psi\rangle \tag{2.92}$$

with

$$p_k = \langle\Psi|\, E_k^\dagger E_k \,|\Psi\rangle\ . \tag{2.93}$$

If we now replace with probability p_k the state $|\Psi\rangle$ by $|\Psi'_k\rangle$, on average the system will be left in the state

$$\rho' \equiv \sum_k p_k |\Psi'_k\rangle\langle\Psi'_k| = \sum_k E_k |\Psi\rangle\langle\Psi| E_k^\dagger = \sum_k E_k \rho E_k^\dagger\,. \tag{2.94}$$

This procedure therefore creates on average the correct density matrix.

- In the simulator, we implement a simple noise model in which each gate operation is followed by an energy relaxation process with

$$E_0 = \begin{pmatrix} 1 & 0 \\ 0 & \sqrt{1-\gamma} \end{pmatrix}, \qquad E_1 = \begin{pmatrix} 0 & \sqrt{\gamma} \\ 0 & 0 \end{pmatrix} \tag{2.95}$$

  with $\gamma \equiv 1 - e^{-T_{\text{gate}}/T_1}$ and gate operation time T_{gate}.

- In addition, we will follow each gate and relaxation process with a depolarization channel

$$\mathcal{E}(\rho) = \frac{p_{\text{depol. err}}}{2}\mathbb{1} + (1 - p_{\text{depol. err}})\rho\,, \tag{2.96}$$

  with gate-specific depolarization error $p_{\text{depol. err}}$. We have given the Kraus operators for this channel previously.

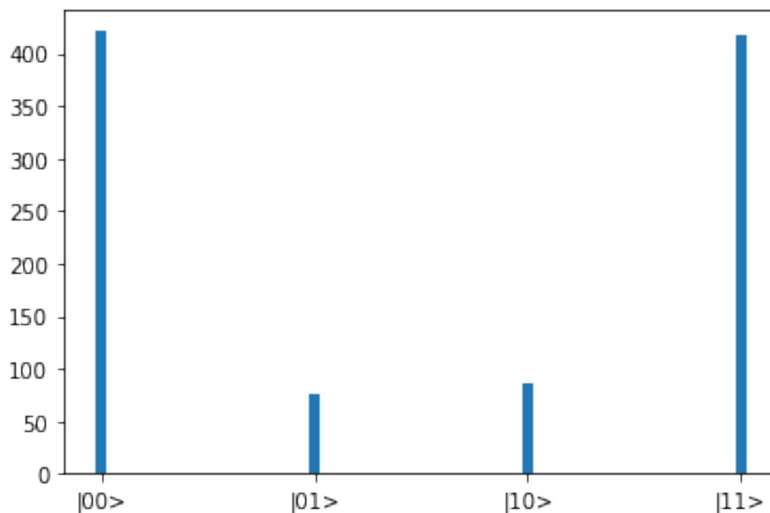
- Finally, readout errors are parametrized by probability p_{readout} with which an additional NOT operation is added before a measurement.

- On the IBM Q experience, $T_1 \approx 50\mu s$ and the gate times for the non-virtual gates are approximately $0.5 - 1\mu s$.

- Example 1: Bell states (noise-bell.ipynb)

In [15]:
```
import sqc
import matplotlib.pyplot as plt

Nbits=2

sqc.seed(13)

# Create noise model
nm=sqc.noise.model.simple(
    T1 = 50,
    gate_times = { "H" : 0.5, "CNOT" : 1.0, "Rz" : 0.0, "X": 0.5 },
    qubit_readout_errors = [ 0.05, 0.05 ],
    gate_depolarization_p = { "X" : 0.05, "CNOT" : 0.1, "Rz" : 0.0, "H
" : 0.1 }
)

# Create Bell state
op=sqc.operator(Nbits).H(0).CNOT(0,1)

# Initial state
s0=sqc.state(Nbits)

# Sample
res=sqc.noise.sample(nm,op,s0,1000)
x=sorted(res.keys())
y=[ res[i] for i in x ]

plt.bar([ s0.basis[i] for i in x ],y,width=0.05)
```

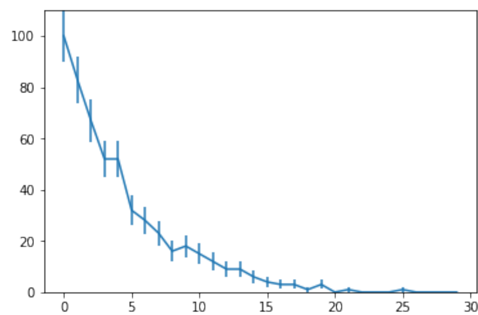
Out[15]:
```
<BarContainer object of 4 artists>
```



- Example 2: Energy relaxation (noise-t1.ipynb)

In [40]:
```
import sqc
import matplotlib.pyplot as plt

Nbits=2

sqc.seed(13)

# Create noise model
nm=sqc.noise.model.simple(
    T1 = 50,
    gate_times = { "H" : 0.0, "CNOT" : 0.0, "Rz" : 10, "X": 0.0 },
    qubit_readout_errors = [ 0.0, 0.0 ],
    gate_depolarization_p = { "X" : 0.0, "CNOT" : 0.0, "Rz" : 0.0, "H" : 0.0 }
)

# Operator
def get_op(t):
    op=sqc.operator(Nbits).X(0)
    for i in range(t):
        op=op.Rz(0,0) # just wait for 1mus
    return op

def val(d,x):
    return d[x] if x in d else 0

# Initial state
s0=sqc.state(Nbits)

# Sample
x=range(30)
y=[ val(sqc.noise.sample(nm,get_op(i),s0,100),1) for i in x ]
yerr=[ n**0.5 for n in y ]

plt.ylim(0,110)
plt.errorbar(x,y,yerr=yerr)
```


Out[40]: <ErrorbarContainer object of 3 artists>

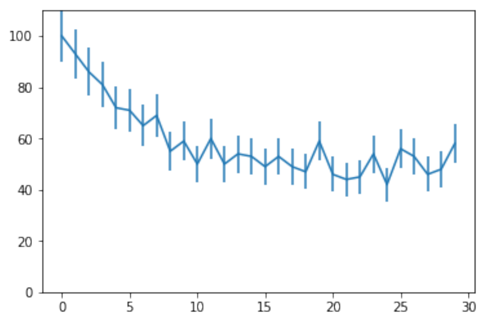


In [ ]:

- Example 2: Deutsch-Jozsa (noise-depol.ipynb)

In [5]:
```
import sqc
import matplotlib.pyplot as plt

Nbits=2

sqc.seed(13)

# Create noise model
nm=sqc.noise.model.simple(
    T1 = 50,
    gate_times = { "H" : 0.0, "CNOT" : 0.0, "Rz" : 0.0, "X": 0.0 },
    qubit_readout_errors = [ 0.0, 0.0 ],
    gate_depolarization_p = { "X" : 0.0, "CNOT" : 0.0, "Rz" : 0.2, "H" : 0.0 }
)

# Operator
def get_op(t):
    op=sqc.operator(Nbits).X(0)
    for i in range(t):
        op=op.Rz(0,0)
    return op

def val(d,x):
    return d[x] if x in d else 0

# Initial state
s0=sqc.state(Nbits)

# Sample
x=range(30)
y=[ val(sqc.noise.sample(nm,get_op(i),s0,100),1) for i in x ]
yerr=[ n**0.5 for n in y ]

plt.ylim(0,110)
plt.errorbar(x,y,yerr=yerr)
```


Out[5]: <ErrorbarContainer object of 3 artists>



In [ ]:

- Example 3: Phase estimation (noise-phase.ipynb)
  Also show run on IBM Q experience

In [2]:
```
import sqc
import numpy as np
from exercises import qft
import matplotlib.pyplot as plt
```

In [27]:
```
# Create noise model for a typical 2019 quantum computer (taken from ibmqx4)
nm2019=sqc.noise.model.simple(
    T1 = 40,
    gate_times = { "H" : 0.5, "CNOT" : 1.0, "Rz" : 0.0, "X": 0.5 },
    qubit_readout_errors = [ 0.043, 0.073, 0.184, 0.35, 0.26 ],
    gate_depolarization_p = { "X" : 0.005, "CNOT" : 0.05, "Rz" : 0.0, "H" : 0.005 }
)
```

Phase estimation

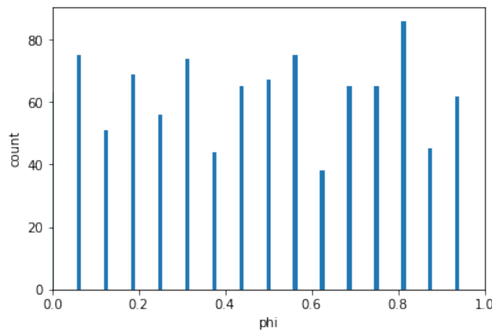
In [29]:
```
def phaseEstimate(op,xbits,cuj):
    N=len(xbits)
    for i in reversed(range(N)):
        op=op.H(xbits[i])
        op=cuj(xbits[i],2**i,op)
    op=qft(op,mask=xbits,inverse=True)
    return op

# Simple U = {{ Exp[I 2pi phi], 0}, { 0, Exp[-I 2pi phi] }}, always acting on LSB
def CU(i,k,op,phi): # i is control qubit, k is power
    return op.Rz(0,2.*np.pi*phi*k).CNOT(i,0).Rz(0,-2.*np.pi*phi*k).CNOT(i,0)

def measure(nm,Nxbits,Nmeasure,cuj):
    Nbits=Nxbits+1
    xbits=list(range(1,Nbits))
```



```
    st0=sqc.state(Nbits,basis=["|%g>|%d>" % ( (i//2) / 2**Nxbits,i%2) for i in range(2**Nbits)])
    print("Initial = 0\n",st0)

    op=phaseEstimate(sqc.operator(Nbits).H(0),xbits,cuj)

    print(len(op.m),"gates")

    #print(op.toQASM())
    if Nmeasure == 0:
        print("State after phaseEstimate\n",st1)
    else:
        res=sqc.noise.sample(nm,op,st0,Nmeasure,mask=xbits)

        plt.bar([ x / 2**Nxbits for x in res.keys() ],res.values(),width=0.01)
        plt.xlabel('phi')
        plt.xlim(0,1)
        plt.ylabel('count')
        plt.show()

        print(res)
```

In [30]:
```
measure(nm2019,4,1000,lambda i,k,op: CU(i,k,op,0.3))
```

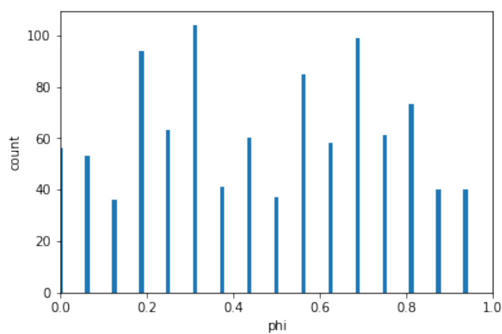
```
Initial = 0
    1 * |0>|0>
61 gates
```




```
{6: 44, 1: 75, 12: 65, 4: 56, 0: 63, 3: 69, 11: 65, 5: 74, 13: 86, 7: 65, 9: 75, 14: 45, 2: 51,
8: 67, 15: 62, 10: 38}
```


In [31]:
```
# Divide readout errors by 10
nm2019div10ro=sqc.noise.model.simple(
    T1 = 40,
    gate_times = { "H" : 0.5, "CNOT" : 1.0, "Rz" : 0.0, "X": 0.5 },
    qubit_readout_errors = [ 0.0043, 0.0073, 0.0184, 0.035, 0.026 ],
    gate_depolarization_p = { "X" : 0.005, "CNOT" : 0.05, "Rz" : 0.0, "H" : 0.005 }
)

measure(nm2019div10ro,4,1000,lambda i,k,op: CU(i,k,op,0.3))
```

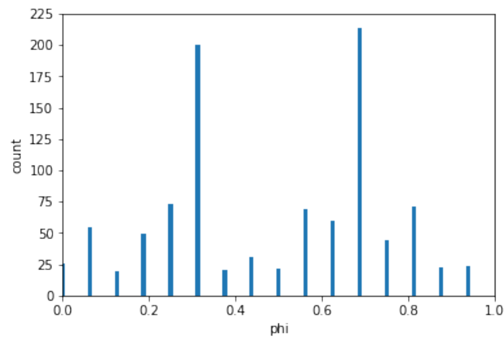
```
Initial = 0
    1 * |0>|0>
61 gates
```




```
{13: 73, 1: 53, 9: 85, 15: 40, 6: 41, 8: 37, 5: 104, 12: 61, 11: 99, 7: 60, 3: 94, 0: 56, 4: 63
, 10: 58, 2: 36, 14: 40}
```

In [32]:
```
# Divide readout errors and gate errors by 10
nm2019div10rog=sqc.noise.model.simple(
    T1 = 40,
    gate_times = { "H" : 0.5, "CNOT" : 1.0, "Rz" : 0.0, "X": 0.5 },
    qubit_readout_errors = [ 0.0043, 0.0073, 0.0184, 0.035, 0.026 ],
    gate_depolarization_p = { "X" : 0.0005, "CNOT" : 0.005, "Rz" : 0.0, "H" : 0.0005 }
)

measure(nm2019div10rog,4,1000,lambda i,k,op: CU(i,k,op,0.3))
```

```
Initial = 0
    1 * |0>|0>
61 gates
```




```
{4: 73, 11: 214, 0: 26, 3: 49, 9: 69, 13: 71, 1: 55, 15: 24, 7: 31, 5: 200, 14: 23, 2: 19, 12:
44, 10: 60, 8: 22, 6: 20}
```

In [ ]:

2.3 Quantum Error Correction

2.3.1 The three-qubit bit flip code

- We start by considering only the bit flip channel. This addresses, e.g., energy relaxation noise.
- In general redundancy is needed to detect and correct an error: replace the **physical** qubits by **logical** qubits

$$|0_L\rangle \equiv |000\rangle \ , \qquad |1_L\rangle \equiv |111\rangle \ , \tag{2.97}$$

i.e., we use three physical qubits to encode a single logical qubit.

- This allows for correction of a single bit flip but not for multiple bit flips at the same time.
- Need to know if a specific qubit was flipped. Formalized by **syndrome** projectors

$$\begin{aligned} P_0 &\equiv |000\rangle\langle 000| + |111\rangle\langle 111| \ , & (2.98)\\ P_1 &\equiv |001\rangle\langle 001| + |110\rangle\langle 110| \ , & (2.99)\\ P_2 &\equiv |010\rangle\langle 010| + |101\rangle\langle 101| \ , & (2.100)\\ P_3 &\equiv |100\rangle\langle 100| + |011\rangle\langle 011| \ , & (2.101)\\ & & (2.102) \end{aligned}$$

indicating no error (P_0) or an error on qubit $i \in \{1,2,3\}$ for P_i.

- Formally, a bit flip on qubit $l \in \{1,2,3\}$ is introduced by channel

$$\mathcal{E}_l(\rho) = (1-p)\rho + pX_l\rho X_l \tag{2.103}$$

with physical qubit error rate p. We will drop the index l when we consider a single physical qubit.

If we consider this channel to act on a logical qubit with three physical qubits, we use

$$\begin{aligned} \overline{\mathcal{E}}(\rho) = \mathcal{E}_1(\mathcal{E}_2(\mathcal{E}_3(\rho))) &= (1-p)^3\rho + (1-p)^2 p \sum_j X_j\rho X_j \\ &+ p^2(1-p)\sum_{j<m} X_jX_m\rho X_mX_j + p^3 X_1X_2X_3\rho X_3X_2X_1 \ , \end{aligned} \tag{2.104}$$

where $[X_i, X_j] = 0$, so the order of application does not matter.

- Similarly, a correcting map acting on a logical qubit can be written as

$$\mathcal{E}_{\text{corr}}(\rho) = P_0\rho P_0 + \sum_{i=1}^{3} X_iP_i\rho P_iX_i = \sum_{\mu=0}^{3} X_\mu P_\mu\rho P_\mu X_\mu \tag{2.105}$$

with $X_0 \equiv \mathbb{1}$.

- Next, we quantify the improvement through error correction using the fidelity metric.
- Recall that for a pure/mixed state combination

$$F(\rho, |\Psi\rangle\langle\Psi|) = \sqrt{\langle\Psi|\rho|\Psi\rangle} \tag{2.106}$$

such that for the original case of a single bit-flip channel acting on a physical qubit, we have

$$F(\mathcal{E}(|\Psi\rangle\langle\Psi|), |\Psi\rangle\langle\Psi|)^2 = 1 - p + p|\langle\Psi| X |\Psi\rangle|^2 \,. \tag{2.107}$$

If we now restrict the state to a physical qubit

$$|\Psi\rangle = a|0\rangle + b|1\rangle \tag{2.108}$$

we find

$$\langle\Psi| X |\Psi\rangle = a^* b + b^* a \tag{2.109}$$

and therefore

$$F(\mathcal{E}(|\Psi\rangle\langle\Psi|), |\Psi\rangle\langle\Psi|)^2 \geq 1 - p \,. \tag{2.110}$$

- If we now consider the case of a logical qubit with both error channel and correction channel, we find

$$\begin{aligned} &F(\mathcal{E}_{\text{corr}}[\overline{\mathcal{E}}(|\Psi\rangle\langle\Psi|)], |\Psi\rangle\langle\Psi|)^2 \\ &= \sum_{\mu=0}^{3} \Bigg[(1-p)^3 |\langle\Psi| P_\mu X_\mu |\Psi\rangle|^2 + (1-p)^2 p \sum_j |\langle\Psi| X_j P_\mu X_\mu |\Psi\rangle|^2 \\ &\quad + p^2(1-p) \sum_{j<m} |\langle\Psi| X_m X_j P_\mu X_\mu |\Psi\rangle|^2 + p^3 |\langle\Psi| X_3 X_2 X_1 P_\mu X_\mu |\Psi\rangle|^2 \Bigg] \,. \end{aligned} \tag{2.111}$$

If we now restrict the discussion to a logical qubit

$$|\Psi\rangle = a|0_L\rangle + b|1_L\rangle = a|000\rangle + b|111\rangle \tag{2.112}$$

we have

$$\langle\Psi| P_\mu X_\mu |\Psi\rangle = \delta_{\mu 0} \,, \qquad \langle\Psi| X_j P_\mu X_\mu |\Psi\rangle = \delta_{j\mu} \,, \tag{2.113}$$

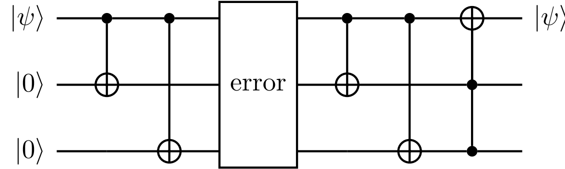
$$\langle\Psi| X_m X_j P_\mu X_\mu |\Psi\rangle = \delta_{\mu 0}\delta_{mj} \,, \quad \langle\Psi| X_3 X_2 X_1 P_\mu X_\mu |\Psi\rangle = \delta_{\mu 0}(a^* b + b^* a) \tag{2.114}$$

such that

$$\begin{aligned} &F(\mathcal{E}_{\text{corr}}[\overline{\mathcal{E}}(|\Psi\rangle\langle\Psi|)], |\Psi\rangle\langle\Psi|)^2 \\ &= (1-p)^3 + 3(1-p)^2 p + p^3 |a^* b + b^* a|^2 \geq 1 - 3p^2 + 2p^3 \,. \end{aligned} \tag{2.115}$$

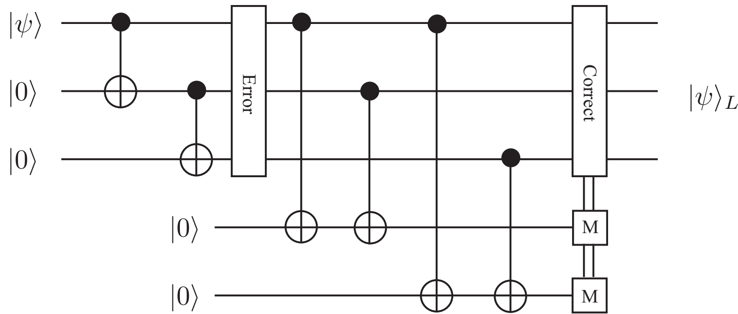
For sufficiently small p, we have therefore reduced the error of order p to an error of order p^2. In fact, we improve the fidelity compared to the original case as long as $p \leq 1/2$.

- To implement the correction channel we cannot simply measure all qubits to detect error since this would destroy the quantum state.
- There is a simple circuit that can restore one physical qubit to the correct value



This circuit, however, does not preserve the logical qubit and can therefore at best be used once. (Need an ancilla reset circuit after this.)

- A better circuit using two ancilla bits and corresponding syndrome measurements is given by



Let a_0 (top) and a_1 (bottom) be the results of the ancilla measurements, then the four possible syndromes are indicated by

	$a_0 = 0$	$a_0 = 1$
$a_1 = 0$	P_0	P_2
$a_1 = 1$	P_3	P_1

Depending on the syndromes, we apply a X_l and depending on the a_i, we apply a NOT gate to the ancilla bits to reset them.

- Example: Let the system start in state

$$|\Psi\rangle \otimes |00\rangle = a\,|0_L\rangle \otimes |00\rangle + b\,|1_L\rangle \otimes |00\rangle = a\,|000\rangle \otimes |00\rangle + b\,|111\rangle \otimes |00\rangle \ , \tag{2.116}$$

where we separate out our logical qubit from the two ancilla bits. Let us now imagine the first bit flips, i.e., the system is in state

$$a\,|001\rangle \otimes |00\rangle + b\,|110\rangle \otimes |00\rangle \tag{2.117}$$

then the ancilla qubits will be modified to

$$a\,|001\rangle \otimes |11\rangle + b\,|110\rangle \otimes |11\rangle \tag{2.118}$$

and measurement of the ancilla bits returns $a_0 = a_1 = 1$, while leaving the superposition intact! This works here because by construction the ancilla bits will always be the same for both terms of the superposition.

Finally, as a result of $a_0 = a_1 = 1$, we apply NOT gates to both ancilla qubits and the first physical qubit such that we are left with our original state

$$|\Psi\rangle \otimes |00\rangle = a\,|0_L\rangle \otimes |00\rangle + b\,|1_L\rangle \otimes |00\rangle\ . \tag{2.119}$$

- There is still a problem if the error occurs during the correction step, we return to this point later.
- Example in sqc:

In [6]:
```
import sqc

Nbits=5

# Initial state
op0=sqc.operator(Nbits).H(0).Rz(0,0.5)
s0=op0*sqc.state(Nbits)
print("Initial physical state")
print(s0)

# Create logical state
op1=sqc.operator(Nbits).CNOT(0,1).CNOT(0,2)
s1=op1*s0
print("Logical state")
print(s1)

# Bit-flip error
op2=sqc.operator(Nbits).X(1)
s2=op2*s1
print("After bit-flip error")
print(s2)

# Correction circuit
opc=sqc.operator(Nbits).CNOT(0,3).CNOT(1,3).CNOT(0,4).CNOT(2,4).M(3,0).M(4,1)

# If
opc=opc.IF(1).X(1).X(3).ENDIF()
opc=opc.IF(2).X(2).X(4).ENDIF()
opc=opc.IF(3).X(0).X(3).X(4).ENDIF()

s3=opc*s2
print("After correction circuit")
print(s3)
```



```
op=op0+op1+op2+opc
print("Entire circuit in QASM")
print(op.toQASM())
```

```
Initial physical state
   0.707107              * |00000>
 + (0.620545+0.339005j) * |00001>
Logical state
   0.707107              * |00000>
 + (0.620545+0.339005j) * |00111>
After bit-flip error
   0.707107              * |00010>
 + (0.620545+0.339005j) * |00101>
After correction circuit
   0.707107              * |00000>
 + (0.620545+0.339005j) * |00111>
Entire circuit in QASM
OPENQASM 2.0;
include "qelib1.inc";
qreg qr[5];
creg cr[5];
h qr[0];
rz(0.159154943091895*pi) qr[0];
cx qr[0],qr[1];
cx qr[0],qr[2];
x qr[1];
cx qr[0],qr[3];
cx qr[1],qr[3];
cx qr[0],qr[4];
cx qr[2],qr[4];
measure qr[3] -> cr[0];
measure qr[4] -> cr[1];
if (cr==1) x qr[1];
if (cr==1) x qr[3];
if (cr==2) x qr[2];
if (cr==2) x qr[4];
```

2.3.2 The three-qubit phase flip code

- Next, we discuss how to correct for phase flips, i.e., spurious Z gates.
- We previously already noted that the quantum channel for a phase flip is identical to the one for infinitesimally small phase fluctuations. So if we protect against sign flip, we protect against small phase fluctuations!
- Note that

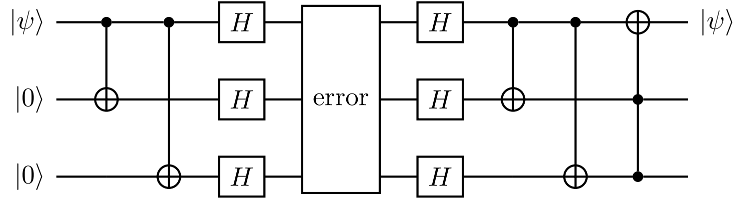
$$Z = HXH \,, \tag{2.120}$$

as well as

$$HZH = X \,, \tag{2.121}$$

i.e., after performing a basis change with H, the X and Z gates switch roles.

- Therefore we can obtain a circuit that protects against phase flips by applying Hadamard gates on all physical qubits of a logical qubit before and after the **error** circuit. Example:



- Write out circuit with two ancilla measurements and additional Hadamard gates!
- Let us demonstrate this for arbitrary spurious R_ϕ in the circuit using two ancilla qubits. We again start in state

$$|\Psi\rangle \otimes |00\rangle = a\,|0_L\rangle \otimes |00\rangle + b\,|1_L\rangle \otimes |00\rangle = a\,|000\rangle \otimes |00\rangle + b\,|111\rangle \otimes |00\rangle \,. \tag{2.122}$$

We then perform the Hadamards and are in state

$$\frac{1}{\sqrt{8}} \sum_{n_0,n_1,n_2=0}^{1} (a + (-1)^{n_0+n_1+n_2} b)\,|n_2 n_1 n_0\rangle \otimes |00\rangle \tag{2.123}$$

After an R_ϕ on the first qubit, we have

$$\frac{1}{\sqrt{8}} \sum_{n_0,n_1,n_2=0}^{1} (a + (-1)^{n_0+n_1+n_2} b) e^{i\phi n_0}\,|n_2 n_1 n_0\rangle \otimes |00\rangle \tag{2.124}$$

and after applying the next Hadamards

$$\begin{aligned} &\frac{1}{8} \sum_{m_0,m_1,m_2,n_0,n_1,n_2=0}^{1} (-1)^{m_0 n_0 + m_1 n_1 + m_2 n_2} \\ &\quad \times (a + (-1)^{n_0+n_1+n_2} b) e^{i\phi n_0} |m_2 m_1 m_0\rangle \otimes |00\rangle \qquad (2.125) \\ &= a z_+ |000\rangle \otimes |00\rangle + a z_- |001\rangle \otimes |00\rangle \\ &\quad + b z_+ |111\rangle \otimes |00\rangle + b z_- |110\rangle \otimes |00\rangle \qquad (2.126) \end{aligned}$$

with

$$z_\pm \equiv \frac{1 \pm e^{i\phi}}{2} \qquad (2.127)$$

and $|z_+|^2 + |z_-|^2 = 1$.

Then the ancilla qubits will be modified to

$$\begin{aligned} &a z_+ |000\rangle \otimes |00\rangle + a z_- |001\rangle \otimes |11\rangle \\ &+ b z_+ |111\rangle \otimes |00\rangle + b z_- |110\rangle \otimes |11\rangle \ . \qquad (2.128) \end{aligned}$$

Now measuring the ancilla bits can either result in $a_0 = a_1 = 0$ and

$$(a |000\rangle \otimes |00\rangle + b |111\rangle \otimes |00\rangle) \frac{z_+}{|z_+|} \qquad (2.129)$$

or $a_0 = a_1 = 1$ and

$$(a |001\rangle \otimes |11\rangle + b |110\rangle \otimes |11\rangle) \frac{z_-}{|z_-|} \ . \qquad (2.130)$$

In the first case, we are done since overall phases are not resolvable by further measurements. In the second, we proceed as before and are left with our original state

$$|\Psi\rangle \otimes |00\rangle = a |0_L\rangle \otimes |00\rangle + b |1_L\rangle \otimes |00\rangle \ . \qquad (2.131)$$

So indeed, this code protects against arbitrary phases!

- Example in sqc:

In [29]:
```
import sqc
import numpy as np

Nbits=5

# Initial state
op0=sqc.operator(Nbits).H(0).Rz(0,0.5)
s0=op0*sqc.state(Nbits)
print("Initial physical state")
print(s0)

# Create logical state
op1=sqc.operator(Nbits).CNOT(0,1).CNOT(0,2)
s1=op1*s0
print("Logical state")
print(s1)

# R_phi error
phierr=0.9
op2=sqc.operator(Nbits).H(0).H(1).H(2).Rz(0,phierr).H(0).H(1).H(2)
s2=op2*s1
print("After phase error")
print(s2)

# Correction circuit
opc=sqc.operator(Nbits).CNOT(0,3).CNOT(1,3).CNOT(0,4).CNOT(2,4).M(3,0).M(4,1)

# If
opc=opc.IF(1).X(1).X(3).ENDIF()
opc=opc.IF(2).X(2).X(4).ENDIF()
opc=opc.IF(3).X(0).X(3).X(4).ENDIF()

s3=opc*s2
print("After correction circuit")
print(s3)
```



```
# For aesthetical reasons, correct for expected overall phase
zplus=(1+np.exp(1j*phierr))/2
zminus=(1-np.exp(1j*phierr))/2
if opc.cval == 0:
    s3.v = [ x*abs(zplus)/zplus for x in s3.v ]
else:
    s3.v = [ x*abs(zminus)/zminus for x in s3.v ]

print("Phase rotated (case %d)" % opc.cval)
print(s3)
```

```
Initial physical state
   0.707107              * |00000>
 + (0.620545+0.339005j) * |00001>
Logical state
   0.707107              * |00000>
 + (0.620545+0.339005j) * |00111>
After phase error
   (0.573326+0.276948j) * |00000>
 + (0.133781-0.276948j) * |00001>
 + (0.25018-0.178907j)  * |00110>
 + (0.370365+0.517912j) * |00111>
After correction circuit
   (0.307567-0.636712j) * |00000>
 + (0.575172-0.411312j) * |00111>
Phase rotated (case 3)
   0.707107              * |00000>
 + (0.620545+0.339005j) * |00111>
```

In [ ]:

In [ ]:

2.3.3 The Shor code

- To protect against arbitrary one-qubit noise channel, we need to protect against the action of arbitrary Kraus operator E_k.

- Can always write

$$E_k = c_{k0}\mathbb{1} + c_{kX}X + c_{kY}Y + c_{kZ}Z \tag{2.132}$$

with Pauli matrices X, Y, Z, and complex c_{k0}, c_{kX}, c_{kY}, and c_{kZ}.

- Note that $Y = iXZ$ and therefore protecting against bit flips (X), sign flips (Z), and combined bit and sign flips (XZ) is sufficient to protect against an arbitrary one-qubit noise channel.

- A simple realization is the 9-qubit Shor code, combining the 3-qubit bit and phase flip codes:

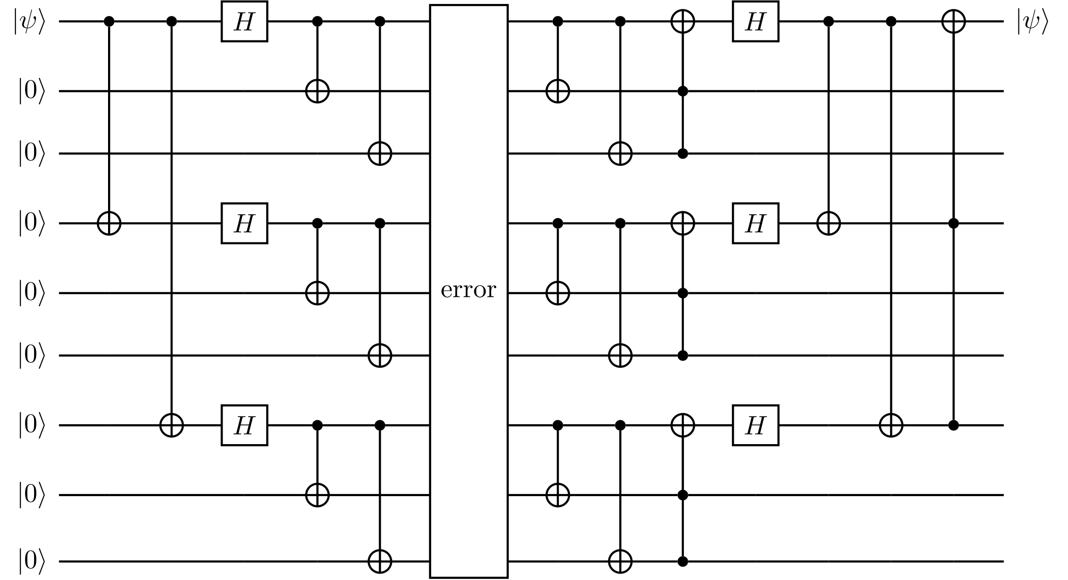
$$|0_L\rangle \equiv \frac{1}{2^{3/2}}\Big(|000\rangle + |111\rangle\Big) \otimes \Big(|000\rangle + |111\rangle\Big) \otimes \Big(|000\rangle + |111\rangle\Big), \tag{2.133}$$

$$|1_L\rangle \equiv \frac{1}{2^{3/2}}\Big(|000\rangle - |111\rangle\Big) \otimes \Big(|000\rangle - |111\rangle\Big) \otimes \Big(|000\rangle - |111\rangle\Big). \tag{2.134}$$

This code is identical to applying the three-qubit bit flip code and the three-qubit phase flip code in succession.

- By design, this therefore protects against arbitrary one-qubit noise channels. Show this explicitly in problem set 10.

- Example circuit:



- We can define gates acting on the logical bits $|0_L\rangle$ and $|1_L\rangle$. The operators

$$X_L \equiv \overline{X} \equiv \prod_{i=1}^{9} Z_i \,, \qquad Z_L \equiv \overline{Z} \equiv \prod_{i=1}^{9} X_i \tag{2.135}$$

with X_i and Z_i acting on physical qubits $i \in \{1, 2, \ldots, 9\}$, for example, satisfy

$$X_L |0_L\rangle = |1_L\rangle \; , \qquad X_L |1_L\rangle = |0_L\rangle \; , \tag{2.136}$$

$$Z_L |0_L\rangle = |0_L\rangle \; , \qquad Z_L |1_L\rangle = - |1_L\rangle \; . \tag{2.137}$$

Note that these have indeed not been switched. Such gates can be used in the **error** gate of the Shor code.

- Example in sqc:

In [5]:
```
import sqc
import numpy as np
from exercises import C2NOT

Nbits=9

# Initial state
op0=sqc.operator(Nbits).H(0).Rz(0,0.5)
s0=op0*sqc.state(Nbits)
print("Initial physical state")
print(s0)
```

```
Initial physical state
   0.707107            * |000000000>
 + (0.620545+0.339005j) * |000000001>
```

In [62]:
```
def Spread(a,b,c,op):
    return op.CNOT(a,b).CNOT(a,c)

def Correct(a,b,c,op):
    return C2NOT(b,c,a,op.CNOT(a,b).CNOT(a,c))

def Reset(bits,op):
    for a in bits:
        op=op.M(a,0).IF(1).X(a).ENDIF()
    return op

# Create logical state
op1=Spread(0,1,2,
        Spread(3,4,5,
            Spread(6,7,8,
               Spread(0,3,6,sqc.operator(Nbits)).H(0).H(3).H(6))))

# Correction circuit
opc=Correct(0,3,6,Correct(0,1,2,Correct(3,4,5,Correct(6,7,8,sqc.operator(Nbits)))).H(
```



```
    0).H(3).H(6))

# Reset ancilla bits circuit
opr=Reset(range(1,9),sqc.operator(Nbits))

# Print logical state
s1=op1*s0
print("Logical state")
print(s1)
```

```
Logical state
   (0.469396+0.119856j)  * |000000000>
 + (0.0306044-0.119856j) * |000000111>
 + (0.0306044-0.119856j) * |000111000>
 + (0.469396+0.119856j)  * |000111111>
 + (0.0306044-0.119856j) * |111000000>
 + (0.469396+0.119856j)  * |111000111>
 + (0.469396+0.119856j)  * |111111000>
 + (0.0306044-0.119856j) * |111111111>
```

In [63]:
```
# General one-qubit error
phierr=0.9

for errbit in range(9):
    print("Errbit", errbit)

    op2=sqc.operator(Nbits).Rz(errbit,phierr).X(errbit).H(errbit)
    s2=op2*s1
    print("After phase error")
    print(s2)

    s3=opc*s2
    print("After correction circuit")
    print(s3)

    s4=opr*s3
    print("After ancilla reset")
```

```python
    print(s4)

    before=s4[1]/s4[0]
    after=s0[1]/s0[0]
    print(before/after)
```

```
Errbit 0
After phase error
   (0.331913+0.0847513j)   * |000000000>
 + (-0.331913-0.0847513j)  * |000000001>
 + (0.0798399-0.0357306j)  * |000000110>
 + (0.0798399-0.0357306j)  * |000000111>
 + (0.0216406-0.0847513j)  * |000111000>
 + (-0.0216406+0.0847513j) * |000111001>
 + (0.139932+0.312678j)    * |000111110>
 + (0.139932+0.312678j)    * |000111111>
 + (0.0216406-0.0847513j)  * |111000000>
 + (-0.0216406+0.0847513j) * |111000001>
 + (0.139932+0.312678j)    * |111000110>
 + (0.139932+0.312678j)    * |111000111>
 + (0.331913+0.0847513j)   * |111111000>
 + (-0.331913-0.0847513j)  * |111111001>
 + (0.0798399-0.0357306j)  * |111111110>
 + (0.0798399-0.0357306j)  * |111111111>
After correction circuit
```

In [ ]:

2.3.4 Stabilizer formalism

- The bit flip, phase flip, and Shor codes are special cases of **stabilizer codes**.
- Formalism is based on properties of the Pauli group with elements

$$G_1 \equiv \{\pm\mathbb{1}, \pm i\mathbb{1}, \pm X, \pm iX, \pm Y, \pm iY, \pm Z, \pm iZ\} \tag{2.138}$$

  acting on a single qubit and

$$G_n \equiv \bigotimes_{i=1}^{n} G_1 \tag{2.139}$$

  acting on n qubits. Note that any two elements of G_n either commute or anticommute.
- Let S be a subgroup of G_n. We define a n-qubit vector space

$$V_S \equiv \left\{ |\psi\rangle \,\middle|\, g\,|\psi\rangle = |\psi\rangle\, \forall g \in S \right\} \tag{2.140}$$

  and we say S is the **stabilizer** of V_S.
- We can show that all elements of S for which $\dim V_S \neq 0$ commute with each other and are not equal to $-\mathbb{1}, \pm i\mathbb{1}$. We will from now on exclude trivial stabilizers S with $\dim V_S = 0$ from the discussion.
- If we can write every element of S as a product of independent operators in $\{K_i \in S\}$, we say the K_i **generate** S. Independence in this context means that no K_i can be written as a product of other generators. Such a set is not unique.
- It suffices to check that $K_i\,|\psi\rangle = |\psi\rangle$ for each K_i to establish that $|\psi\rangle \in V_S$.
- If S has $n-k$ generators, we have $\dim V_S = 2^k$ and we can select k logical qubits from V_S. To assign the k logical 0 and 1 bits, we pick operators $\overline{Z}_1, \ldots, \overline{Z}_k \in G_n$ that commute with all elements of G_n and use the eigenvalues of $\overline{Z}_i$ to define the logical bits. Resulting codes are called $[n,k]$ **stabilizer codes**.
- In the case of $k=1$, $\dim V_S = 2$ and the logical $\overline{Z} = \overline{Z}_1$ is selected to have eigenvectors $|0_L\rangle$ and $|1_L\rangle$ with eigenvalues 1 and -1.
- If an error $E \in G_n$ is applied to a logical state $|\psi_L\rangle$, we have

$$K_j E\,|\psi_L\rangle = (-1)^{m_j} E\,|\psi_L\rangle \tag{2.141}$$

  with $m_j = 0$ if $[K_j, E] = 0$ and $m_j = 1$ if $\{K_j, E\} = 0$. Similarly, if no error occurred, we find

$$K_j\,|\psi_L\rangle = |\psi_L\rangle \tag{2.142}$$

by construction. Therefore measuring the $n-k$ eigenvalues $\beta_1, \dots, \beta_{n-k}$ (syndromes) of $K_1, \dots, K_{n-k}$ gives us insight into possible errors. If one of the β_l is -1, an error occurred. If only values of $+1$ are measured, we can only be sure that no error occurred if E anticommutes with at least one K_j.

- In addition, however, if two errors E_i and E_j yield the same syndromes β_l, we cannot correct for both.
- The set of correctable errors can be formalized by defining the **normalizer**

$$N(S) \equiv \{E \in G_n | EgE^\dagger \in S, \forall g \in S\} \,. \tag{2.143}$$

We can then show that the set of error gates $\{E_j\} \subset G_n$ is correctable if $E_j^\dagger E_k \notin N(S) - S$ for all j and k.

- Example (3-qubit bit flip code):

$$S \equiv \{\mathbb{1}, Z_1 Z_2, Z_2 Z_3, Z_1 Z_3\} \tag{2.144}$$

with Z_l acting on qubit $l \in \{1, 2, 3\}$. A possible set of generators is

$$K_1 \equiv Z_1 Z_2 \,, \qquad K_2 \equiv Z_1 Z_3 \tag{2.145}$$

and a logical Z can be selected as

$$\overline{Z} = Z_1 Z_2 Z_3 \,. \tag{2.146}$$

It is straightforward to check that $[\overline{Z}, K_i] = 0$ and that

$$V_S = \text{span}\{|000\rangle \,, |111\rangle\} \,. \tag{2.147}$$

Furthermore

$$\overline{Z} \,|000\rangle = |000\rangle \equiv |0_L\rangle \;, \tag{2.148}$$

$$\overline{Z} \,|111\rangle = - \,|111\rangle \equiv - \,|1_L\rangle \;. \tag{2.149}$$

We can show that

$$\{\mathbb{1}, X_1, X_2, X_3\} \tag{2.150}$$

is a correctable set as discussed above. The syndromes for X_1, e.g., are

$$\beta_1 = \beta_2 = -1 \,. \tag{2.151}$$

Similarly, this code can correct $\{\mathbb{1}, Y_1, Y_2, Y_3\}$, however not $\{\mathbb{1}, Z_1, Z_2, Z_3\}$ (since they commute with all syndromes).

– Example (9-qubit Shor code):

  A possible set of generators is

$$
\begin{aligned}
K_1 &= Z_1 Z_2\,, & K_2 &= Z_2 Z_3\,, & (2.152)\\
K_3 &= Z_4 Z_5\,, & K_4 &= Z_5 Z_6\,, & (2.153)\\
K_5 &= Z_7 Z_8\,, & K_6 &= Z_8 Z_9\,, & (2.154)\\
K_7 &= X_1 X_2 X_3 X_4 X_5 X_6\,, & K_8 &= X_4 X_5 X_6 X_7 X_8 X_9 & (2.155)
\end{aligned}
$$

  and

$$\overline{Z} = \prod_{i=1}^{9} X_i\,. \tag{2.156}$$

  This corresponds to the Shor code. It is straightforward to check that this formally corrects for arbitrary single qubit errors.

– Consider a $[n, 1]$ stabilizer code, i.e., n physical qubits and $n-1$ stabilizers $K_1, \ldots, K_{n-1}$. As shown above, to correct for a general one-qubit error, we need to detect and correct X, Z, and XZ errors on the n physical qubits. Therefore, we need to detect at least $3n$ syndromes. At the same time, we will have $2^{n-1}-1$ possible syndromes indicated by measurements of eigenvalues β_i of K_i ($\beta_1 = \ldots = \beta_{n-1} = 1$ indicates no error). Therefore we need

$$2^{n-1} - 1 \geq 3n \tag{2.157}$$

  which can be satisfied for $n \geq 5$.

– Example 3 (5-qubit code, minimal size to correct for general one-qubit errors):

  A possible set of generators is

$$
\begin{aligned}
K_1 &= X_1 Z_2 Z_3 X_4\,, & K_2 &= X_2 Z_3 Z_4 X_5\,, & (2.158)\\
K_3 &= X_1 X_3 Z_4 Z_5\,, & K_4 &= Z_1 X_2 X_4 Z_5 & (2.159)
\end{aligned}
$$

  and

$$\overline{Z} = \prod_{i=1}^{5} Z_i\,. \tag{2.160}$$

  We will discuss this code in detail in problem set 11. For now, let us explicitly give the syndromes β_l for the error set $\{\mathbb{1}, X_l, Z_l, X_l Z_l | l \in \{1, 2, 3, 4, 5\}\}$:

E	β_1	β_2	β_3	β_4
X_1	1	1	1	-1
Z_3	1	1	-1	1
X_5	1	1	-1	-1
Z_5	1	-1	1	1
Z_2	1	-1	1	-1
X_4	1	-1	-1	1
X_5Z_5	1	-1	-1	-1
X_2	-1	1	1	1
Z_4	-1	1	1	-1
Z_1	-1	1	-1	1
X_1Z_1	-1	1	-1	-1
X_3	-1	-1	1	1
X_2Z_2	-1	-1	1	-1
X_3Z_3	-1	-1	-1	1
X_4Z_4	-1	-1	-1	-1

We see that the syndromes β_l uniquely match to the errors and therefore allow for their correction.

– In order to construct a logical state for a general stabilizer code, we can proceed as follows.
  We first remember that for any operator with $U^2 = \mathbb{1}$, we have

$$U\left|\pm\right\rangle = \pm\left|\pm\right\rangle \tag{2.161}$$

for

$$\left|\pm\right\rangle \equiv (\mathbb{1} \pm U)\left|\psi\right\rangle \tag{2.162}$$

for a general state $\left|\psi\right\rangle$ with $|\left|\pm\right\rangle| \neq 0$.
Therefore we can create logical $\left|0_L\right\rangle$ and $\left|1_L\right\rangle$ by normalizing the vectors

$$\left|\tilde{0}_L\right\rangle \equiv (\mathbb{1} + \overline{Z})\prod_{i=1}^{n-k}(\mathbb{1} + K_i)\left|\psi\right\rangle , \tag{2.163}$$

$$\left|\tilde{1}_L\right\rangle \equiv (\mathbb{1} - \overline{Z})\prod_{i=1}^{n-k}(\mathbb{1} + K_i)\left|\psi\right\rangle \tag{2.164}$$

for any $\left|\psi\right\rangle$ resulting in non-vanishing $\left|\tilde{0}_L\right\rangle$ and $\left|\tilde{1}_L\right\rangle$.

– Example (three-qubit bit-flip code):
  First note that

$$\left|\psi'\right\rangle = (\mathbb{1} + Z_1Z_2)(\mathbb{1} + Z_2Z_3)\sum_{n=0}^{7}\left|n\right\rangle = 4(\left|000\right\rangle + \left|111\right\rangle) . \tag{2.165}$$

Therefore

$$\left|\tilde{0}_L\right\rangle = (\mathbb{1} + \overline{Z})\left|\psi'\right\rangle = (\mathbb{1} + Z_1 Z_2 Z_3)\left|\psi'\right\rangle = 8\left|000\right\rangle , \tag{2.166}$$

$$\left|\tilde{1}_L\right\rangle = (\mathbb{1} - \overline{Z})\left|\psi'\right\rangle = (\mathbb{1} - Z_1 Z_2 Z_3)\left|\psi'\right\rangle = 8\left|111\right\rangle \tag{2.167}$$

and thus

$$\left|0_L\right\rangle = \left|000\right\rangle , \qquad \left|1_L\right\rangle = \left|111\right\rangle . \tag{2.168}$$

– Example (9-qubit Shor code):
We first note that

$$\left|\psi'\right\rangle = \prod_{i=1}^{6}(\mathbb{1} + K_i) \sum_{n=0}^{2^9-1} c_n \left|n\right\rangle \tag{2.169}$$

$$= 2^6 (c_0 \left|000000000\right\rangle + c_7 \left|000000111\right\rangle + c_{56} \left|000111000\right\rangle + c_{63} \left|000111111\right\rangle + c_{448} \left|111000000\right\rangle + c_{455} \left|111000111\right\rangle + c_{504} \left|111111000\right\rangle + c_{511} \left|111111111\right\rangle) . \tag{2.170}$$

Then (skip next, directly define double prime)

$$(\mathbb{1} + K_7)\left|\psi'\right\rangle = 2^6 [(c_0 + c_{504})(\left|000000000\right\rangle + \left|111111000\right\rangle) + (c_7 + c_{511})(\left|000000111\right\rangle + \left|111111111\right\rangle) + (c_{56} + c_{448})(\left|000111000\right\rangle + \left|111000000\right\rangle) + (c_{63} + c_{455})(\left|000111111\right\rangle + \left|111000111\right\rangle)] . \tag{2.171}$$

and

$$\left|\psi''\right\rangle \equiv (\mathbb{1} + K_8)(\mathbb{1} + K_7)\left|\psi'\right\rangle = \tag{2.172}$$

$$2^6 [\underbrace{(c_0 + c_{504} + c_{63} + c_{455})}_{\equiv c_A}(\left|000000000\right\rangle + \left|111111000\right\rangle + \left|000111111\right\rangle + \left|111000111\right\rangle) + \underbrace{(c_7 + c_{511} + c_{56} + c_{448})}_{\equiv c_B}(\left|000000111\right\rangle + \left|111111111\right\rangle + \left|000111000\right\rangle + \left|111000000\right\rangle)] . \tag{2.173}$$

Finally

$$\left|\tilde{0}_L\right\rangle = (\mathbb{1} + \overline{Z})\left|\psi''\right\rangle = \tag{2.174}$$

$$2^6 (c_A + c_B)[(\left|000000000\right\rangle + \left|111111000\right\rangle + \left|000111111\right\rangle + \left|111000111\right\rangle) + (\left|000000111\right\rangle + \left|111111111\right\rangle + \left|000111000\right\rangle + \left|111000000\right\rangle)] , \tag{2.175}$$

$$\left|\tilde{1}_L\right\rangle = (\mathbb{1} - \overline{Z})\left|\psi''\right\rangle = \tag{2.176}$$

$$2^6 (c_A - c_B)[(\left|000000000\right\rangle + \left|111111000\right\rangle + \left|000111111\right\rangle + \left|111000111\right\rangle) - (\left|000000111\right\rangle + \left|111111111\right\rangle + \left|000111000\right\rangle + \left|111000000\right\rangle)] \tag{2.177}$$

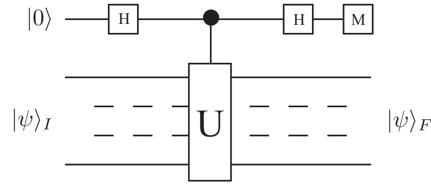
and therefore

$$|0_L\rangle = \frac{1}{2^{3/2}}\Big(|000\rangle + |111\rangle\Big) \otimes \Big(|000\rangle + |111\rangle\Big) \otimes \Big(|000\rangle + |111\rangle\Big), \tag{2.178}$$

$$|1_L\rangle = \frac{1}{2^{3/2}}\Big(|000\rangle - |111\rangle\Big) \otimes \Big(|000\rangle - |111\rangle\Big) \otimes \Big(|000\rangle - |111\rangle\Big). \tag{2.179}$$

- As discussed above, we may assume that the system is in either a +1 or a -1 eigenstate of each generator K_l corresponding to the syndromes β_l.

  We can measure β_l, e.g., using the circuit

  with $|\psi\rangle_I$ being the encoded qubit and $U = K_l$ using a single ancilla qubit.

  Before the measurement, the system is in state

  $$\frac{1}{2}(\mathbb{1} + U)\,|\psi\rangle_I \otimes |0\rangle + \frac{1}{2}(\mathbb{1} - U)\,|\psi\rangle_I \otimes |1\rangle\ . \tag{2.180}$$

  If the system is in a +1 eigenstate of U, the second term vanishes and the system is in

  $$|\psi\rangle_I \otimes |0\rangle \tag{2.181}$$

  and if it is in a -1 eigenstate, we find

  $$|\psi\rangle_I \otimes |1\rangle\ . \tag{2.182}$$

  Therefore, measuring the ancilla bit a, we learn the syndrome $\beta_l = (-1)^a$ while leaving $|\psi\rangle_F = |\psi\rangle_I$.

- Using these ingredients, we will create the circuit for the 5-qubit error correcting code in problem set 11.

2.3.5 Gottesman Knill Theorem

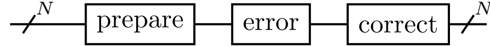
- Using the stabilizer formalism, one can prove the following remarkable theorem by Gottesman and Knill:

- Any circuit that only includes a CNOT, H, and R_ϕ with $\phi = \pi/4$ can be simulated on a classical computer in polynomial time!
- Any such circuit, therefore cannot yield an exponential speedup over a classical algorithm!
- Note that this covers a large set of entangled systems of the Bell type!
- Furthermore, note that this only restricts the angle ϕ compared to a fully universal set.

Comment: FlexNow anmeldung, problem set 10, compact notation for problem 1 can be done on one page.

2.3.6 Fault tolerance – general idea

- We have so far considered circuits of the form



where N physical qubits encode a single logical qubit. We considered an error to occur with probability p on any of the N physical qubits within the **error** gate. We have then shown that we can reduce the error rate to $O(p^2)$.

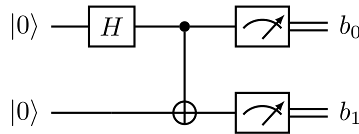
- To reduce the error rate on the entire circuit to $O(p^2)$, we introduce the concept of **fault tolerant** (FT) gate operations. We define FT to mean that a failure of a FT gate operation (including the case of a pre-existing error on a single physical input qubit) may only result in an error on a single physical qubit per encoded logical qubit.

- If we had FT preparation, FT logical gate, and FT correction circuits, we can produce arbitrary circuits which at the end have a single qubit error per logical qubit with O(p) and two or more qubit errors per logical qubit with O(p^2).

- Then using the redundancy in the encoded states (majority voting), we can FT measure the final logical qubits with overall error rate $O(p^2)$.

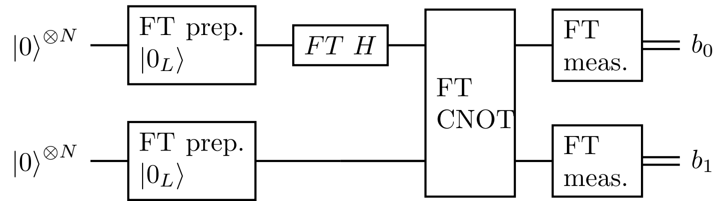
- Example (Bell state):

  Original circuit:



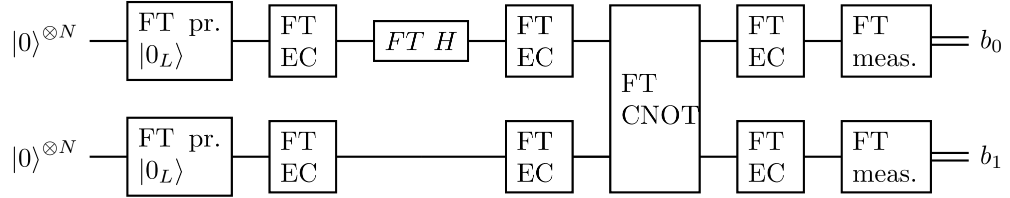
Assuming an error to occur with probability p on any of the qubits at any point in the circuit, the resulting measurements will be incorrect with probability O(p).

FT circuit with N physical qubits per logical qubit:



Assuming p to be an upper bound of the failure probability of a single FT operation and n the number of FT operations, the circuit failure is bound by n^2p^2. Note that n here scales with the number of total failure locations in the entire circuit!

FT circuit with N physical qubits per logical qubit with error-correction (EC):



This circuit only fails if two physical qubit errors accumulate before the error correction steps. We can bound the failure rate by counting the number c of possible 2-error combinations that can occur during the application of an FT EC, FT gate, and another FT EC combination. The FT EC gates, therefore localize the failure rate compared to the previous circuit.

– Finally, N logical qubits can again be encoded, e.g., using a stabilizer code, reducing the overall error rate to $O(p^4)$. For n such levels, we would require N^n physical qubits and reduce the error rate to $O(p^{2n})$. The resulting codes are called **concatenated codes**.

2.3.7 Threshold theorem

– Let us assume that errors on individual physical qubits occur independently with probability p per given fixed time step.

– Furthermore, assume logical gate operations, to leading order, fail with a probability of at most $p_L^{(1)} = cp^2$ for a FT circuit as discussed in the previous section. c here is an upper bound for the number of possible 2-error combinations that can occur during the application of an error correction circuit, the logical gate operation, and another error correction circuit.

– Concatenating codes to two levels then bounds the failure probability of a second-level logical gate operation by $p_L^{(2)} = c(p_L^{(1)})^2 = c^3p^4$. And for g concatenated levels

$$p_L^{(g)} = \frac{(cp)^{2^g}}{c}\,. \tag{2.183}$$

– From this it follows that we can make the failure rate of the g-level-encoded circuit arbitrarily small as long as $cp \ll 1$ and we go to appropriate order in g.

- By studying optimal implementations of FT operations, one typically finds that once $p < p_{\rm th}$ with $p_{\rm th} \approx 10^{-6} - 10^{-4}$, stable quantum computation is possible.

2.3.8 Fault tolerant operations

- When constructing logical gate operations U_L for stabilizer codes with stabilizer group S, we need $U_L \left|\psi\right\rangle_L \in V_S$ for $\left|\psi\right\rangle_L \in V_S$. Since

$$U_L K_i U_L^\dagger U_L \left|\psi\right\rangle_L = U_L K_i \left|\psi\right\rangle_L = U_L \left|\psi\right\rangle_L \tag{2.184}$$

for a generator K_i, we need that $U_L K_i U_L^\dagger \in S$. Any such gate U_L encodes a logical unitary gate operation. Logical Hadamard $\overline{H}$ and $\overline{X}$, e.g., are then defined by

$$\overline{HZH} = \overline{X}\,, \qquad \overline{XZ} = -\overline{ZX}\,. \tag{2.185}$$

For 5-qubit code, e.g.,

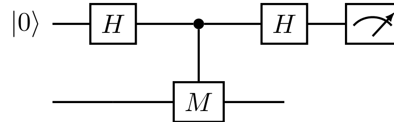
$$\overline{X} = X_1 X_2 X_3 X_4 X_5 \tag{2.186}$$

is the logical X. (A simple product does not work, e.g., for the Hadamard in the 5-qubit code.)

- A logical gate that we can write as a product of individual gates acting on only one physical qubit per logical qubit is called **transversal** and is by construction FT. discuss.

- FT constructions for other logical gates are possible but will not be discussed here. Often they rely on transversal gates involving additional ancilla bits, which are then measured to project to the gate of interest.

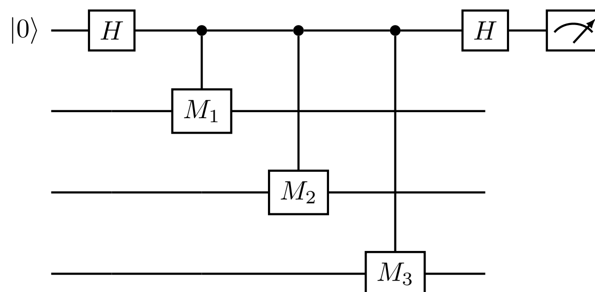
- FT syndrome measurement, logical state preparation, and measurement of classical bits can both be performed using a FT version of



discussed in the last lecture. If we set $M = K_i$, we can measure the syndrome β_i. If we then repeat the circuit for all generators of a stabilizer code, the resulting state will be a logical state. By then FT measuring, $\overline{Z}$, and if needed applying a FT $\overline{X}$ gate, we can create logical states in a FT manner. For $M = \overline{Z}$, the circuit yields a FT measurement circuit of a classical bit.

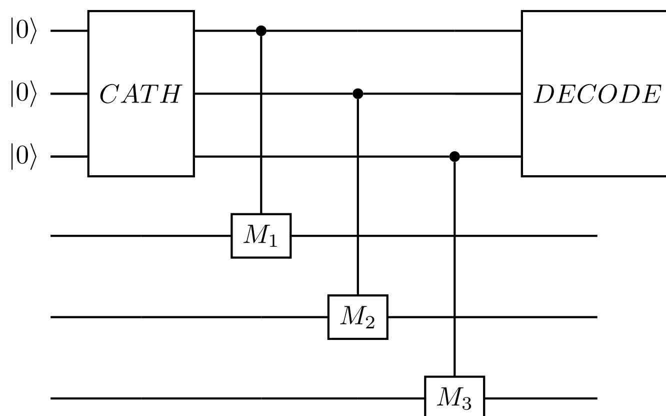
- Combined with a FT syndrome measurement, we need a FT application of error gates E_i to create the FT EC circuit. These are typically transversal, so straightforward to construct.

- Let us construct a FT measurement circuit. Let us assume that we have a transversal implementation of the logical-M gate. Then the above circuit looks like



assuming three physical bits per logical qubit for concreteness. This is not FT because a single error on the ancilla is propagated to multiple physical qubits.

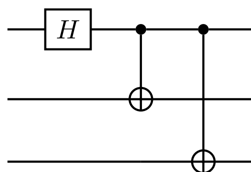
A better circuit uses one ancilla bit per controlled M_i



The CAT H circuit needs to prepare the **cat** state

$$|000\rangle + |111\rangle \tag{2.187}$$

in a reliable way. To do this, we first use a combination of H and CNOT gates



to create the state assuming no error. We then use an additional ancilla qubit, to measure all possible error syndromes such as Z_1Z_2 for the preparation of the cat state. If any such measurement fails, we reset the ancillas and start the CAT H gate again. If an error occurs after partial verification on a single ancilla qubit, it will only affect a single physical qubit of the logical qubit and FT is preserved.

Finally, the DECODE gate applies the H and CNOT gates of the CAT H circuit in reverse before measuring the top ancilla. If an error occurs during this step, the measurement will be wrong. We therefore repeat the entire circuit 3 times and take a majority vote to eliminate O(p) errors.

Chapter 3

Scientific quantum computing

- Use quantum computer to simulate other quantum systems
- Some problems are exponentially difficult with classical algorithms (real-time evolution of Minkowski field theory, simulation at real chemical potential) but polynomially complex on a quantum computer.
- For now map problem to quantum computer and simulate on classical computer. Once fault-tolerant quantum computing is available, we can transfer the circuits to a real quantum computer and scale up the problem size.

3.1 One-dimensional spin-chain

- Consider the Hamiltonian

$$H_{\text{open}} = -\frac{1}{2}\sum_{i=1}^{N-1} Z_i Z_{i+1} + \frac{g}{2}\sum_{i=1}^{N} X_i \tag{3.1}$$

  of the one-dimensional transverse-field Ising model with open boundary conditions. For a magnetic field $g < 1$, the system is ferromagnetically ordered at zero temperature and a paramagnet for $g > 1$.

- We can also study the case with periodic boundary conditions

$$H_{\text{periodic}} = -\frac{1}{2}\sum_{i=1}^{N} Z_i Z_{i+1} + \frac{g}{2}\sum_{i=1}^{N} X_i \tag{3.2}$$

  with $Z_{N+1} = Z_1$.

- On a quantum computer, we can simulate a unitary matrix such as the transfer matrix

$$T = \exp(itH)\,. \tag{3.3}$$

- We will employ a general Trotterization technique to create a circuit for T. First remember the Baker-Campbell-Hausdorff formula

$$\log(\exp(tA)\exp(tB)) = t(A+B) + \frac{1}{2}t^2[A,B] + O(t^3)\,. \tag{3.4}$$

  With this, we can show that

$$\exp(t(A+B)) = \left(\exp(tA/N)\exp(tB/N)\right)^N + O(t/N) \tag{3.5}$$

  for any positive integer N. We can also derive an improved relation

$$\exp(t(A+B)) = \left(\exp(\frac{t}{2N}A)\exp(\frac{t}{N}B)\exp(\frac{t}{2N}A)\right)^N + O(t^2/N^2) \tag{3.6}$$

$$= \exp(\frac{t}{2N}A)\left(\exp(\frac{t}{N}B)\exp(\frac{t}{N}A)\right)^N \exp(-\frac{t}{2N}A) + O(t^2/N^2)\,. \tag{3.7}$$

- With this, we can approximate T by products of matrices

$$T_0^{(j)} = \exp(it\frac{g}{2}X_j) \tag{3.8}$$

  and

$$T_1^{(i)} = \exp(-it\frac{1}{2}Z_iZ_{i+1})\,. \tag{3.9}$$

  Both cases are of the form

$$\exp(i\theta M) = \cos(\theta)\mathbb{1} + i\sin(\theta)M \tag{3.10}$$

  with $M^2 = \mathbb{1}$.

- We then can express the respective sums as products of elementary gates using

$$XR_z(\theta)XR_z(\theta)HR_z(-2\theta)H = \cos(\theta)\mathbb{1} + i\sin(\theta)X \tag{3.11}$$

  and

$$CNOT(1,0)X_0R_{z,0}(\theta)X_0R_{z,0}(-\theta)CNOT(1,0) = \cos(\theta)\mathbb{1} + i\sin(\theta)Z_0Z_1 \tag{3.12}$$

  with CNOT(c,t) having control qubit c and target qubit t. One-qubit gates act on the qubit denoted in the subscript.

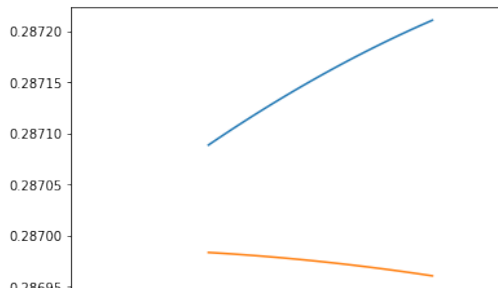
In [1]:
```
import sqc
import numpy as np
from exercises import CRz,qft,twoSiteTIMsz
import matplotlib.pyplot as plt
```

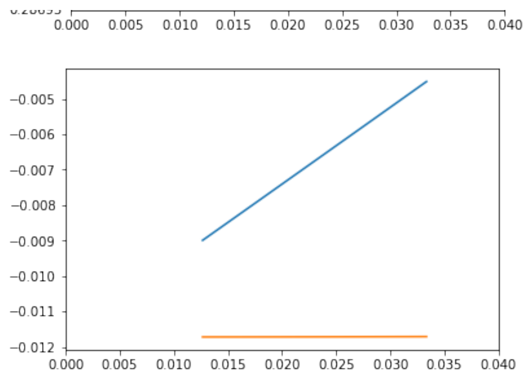
In [2]:
```
def CT(c,sbits,Ntrot,dt,op,params):
    imp=params[2]
    if not imp:
        for i in range(Ntrot):
            op=CT0(c,sbits,dt/Ntrot,op,params)
            op=CT1(c,sbits,dt/Ntrot,op,params)
    else:
        op=CT0(c,sbits,dt/Ntrot/2.,op,params)
        for i in range(Ntrot-1):
            op=CT1(c,sbits,dt/Ntrot,op,params)
            op=CT0(c,sbits,dt/Ntrot,op,params)
        op=CT1(c,sbits,dt/Ntrot,op,params)
        op=CT0(c,sbits,dt/Ntrot/2.,op,params)
    return op
```


In [3]:
```
def CT0(c,sbits,dt,op,params): # prod_j exp(i (dt g / 2) X_j)
    g=params[0]
    for s in sbits:
        a=dt*g/2.
        # X.Rp[a].X.Rp[a].H.Rp[-2 a].H
        op=op.H(s)
        op=CRz(c,s,-2.*a,op)
        op=op.H(s)
        op=CRz(c,s,a,op)
        op=op.X(s)
        op=CRz(c,s,a,op)
        op=op.X(s)
    return op

def CT1(c,sbits,dt,op,params): # CNOT01.XA.RA[t].XA.RA[-t].CNOT01
    periodic=params[1]
    nmax=len(sbits)
    if not periodic:
        nmax=nmax - 1

    for i in range(nmax):
        a=sbits[i]
        b=sbits[(i+1)%len(sbits)]
        l=-dt/2.
        op=op.CNOT(b,a)
        op=CRz(c,a,-l,op)
        op=op.X(a)
        op=CRz(c,a,l,op)
        op=op.X(a)
        op=op.CNOT(b,a)

    return op
```

In [7]:
```
# extrapolation with and without improved trotter
xvals=[]
yvalsr=[]
```

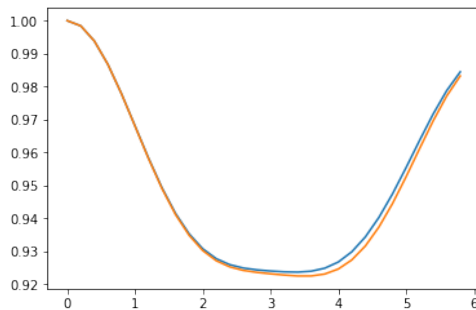
```
yvalsrQ=[]
yvalsi=[]
yvalsiQ=[]

for N in range(30,80):
    s1=CT(0,[1,2,3,4],N,1.5,sqc.operator(5).X(0),[0.5,False,False])*sqc.state(5)
    s1Q=CT(0,[1,2,3,4],N,1.5,sqc.operator(5).X(0),[0.5,False,True])*sqc.state(5)
    xvals.append(1/N)
    yvalsr.append(s1[3].real)
    yvalsrQ.append(s1Q[3].real)
    yvalsi.append(s1[3].imag)
    yvalsiQ.append(s1Q[3].imag)

plt.xlim(0,0.04)
plt.plot(xvals,yvalsr)
plt.plot(xvals,yvalsrQ)
plt.show()

plt.xlim(0,0.04)
plt.plot(xvals,yvalsi)
plt.plot(xvals,yvalsiQ)
plt.show()
```





In [5]:

```
# time-dependent plot
xvals=[]
yvals=[]
yvals2=[]
for tstep in range(30):
    t=0.2*tstep
    xvals.append(t)

    s1=CT(0,[1,2,3,4],10,t,sqc.operator(5).X(0),[0.2,False,True])*sqc.state(5)
    yvals.append(abs(s1[1])**2.)

    s1=CT(0,[1,2,3,4],20,t,sqc.operator(5).X(0),[0.2,False,True])*sqc.state(5)
    yvals2.append(abs(s1[1])**2.)

plt.plot(xvals,yvals)
plt.plot(xvals,yvals2)
plt.show()
```



In [35]:

```
# ground state for g=0
def bits(y,N):
    return [ (y//2**j) % 2 for j in range(N) ]
```

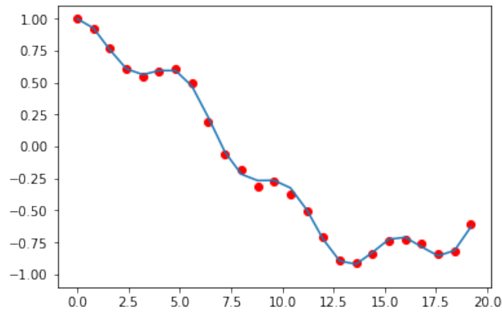


```
def SZ(res,nsites):
    ntot=0.0
    dtot=0.0
    for x in res:
        n=res[x]
        dtot+=n
        bs=[ -1 if b == 1 else 1 for b in bits(x,nsites)]
        ntot+=n*sum(bs)
    return ntot/dtot/nsites

def avgZ(nsites,g,pbc):
    op=sqc.operator(nsites+1).X(0)
    s0=op*sqc.state(nsites+1)
    print(s0)

    xvals=[]
    yvals=[]
    yvals2=[]

    dt=0.8
    Tdt=CT(0,range(1,nsites+1),10,dt,sqc.operator(1+nsites),[g,pbc,True])
    s1=s0
    for tstep in range(25):
        t=dt*tstep
        xvals.append(t)

        res=SZ(sqc.sample(s1, n=1000, mask=range(1,nsites+1)),nsites)
        yvals.append(res)

        if nsites == 2 and pbc == False:
            yvals2.append(twoSiteTIMsz(t,g))
        elif nsites == 2 and pbc == True:
            yvals2.append(twoSiteTIMsz(t*2,g/2))

        s1=Tdt*s1
```

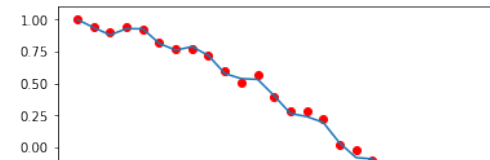
```
    plt.ylim(-1.1,1.1)
    plt.plot(xvals,yvals,'ro')
    if len(yvals2) != 0:
        plt.plot(xvals,yvals2)
    plt.show()
```

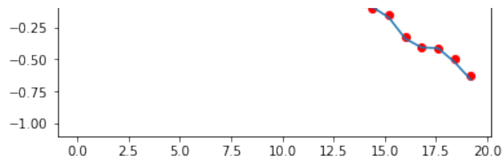
In [36]:
```
avgZ(2,0.5,False)
avgZ(2,0.5,True)
```

```
1 * |001>
```



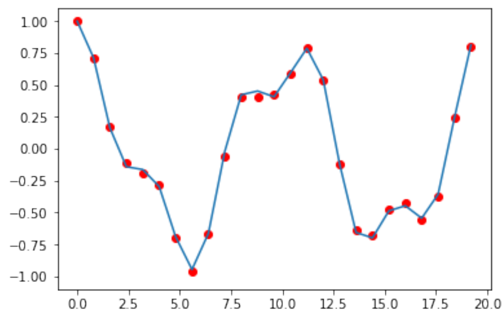
```
1 * |001>
```



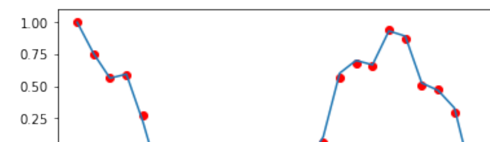


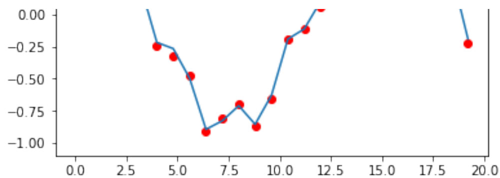
In [37]:
```
avgZ(2,1,False)
avgZ(2,1,True)
```

```
1 * |001>
```



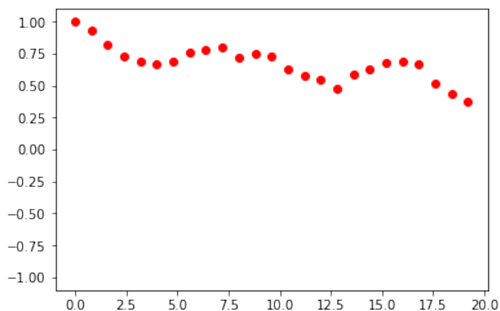
```
1 * |001>
```



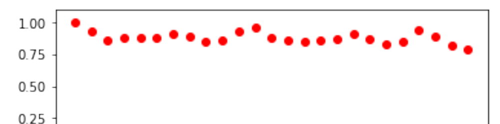


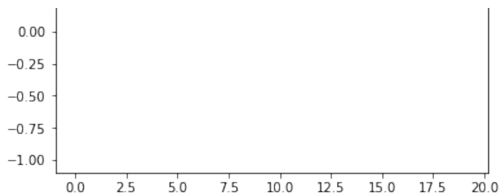
In [38]:
```
avgZ(4,0.5,False)
avgZ(4,0.5,True)
```

1 * |00001>



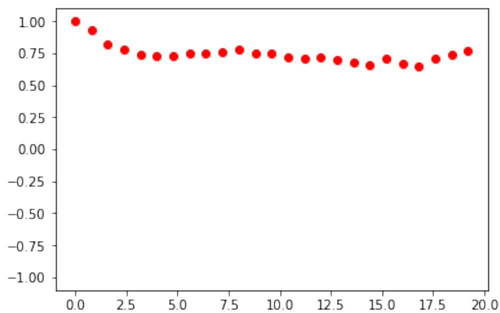
1 * |00001>



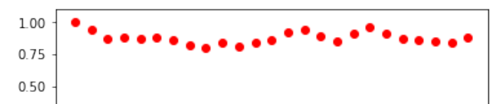


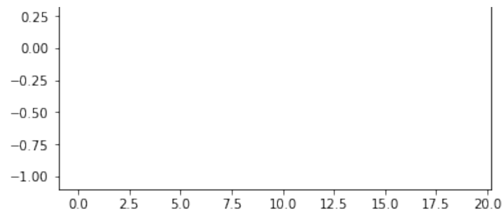
In [39]:
```
avgZ(6,0.5,False)
avgZ(6,0.5,True)
```

1 * |0000001>



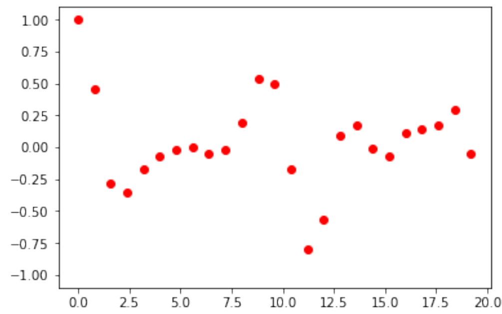
1 * |0000001>



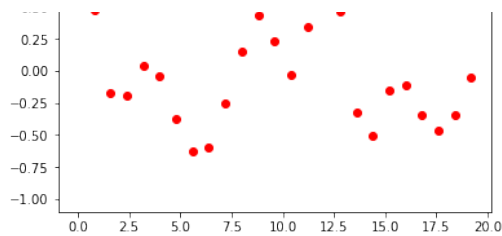


In [40]:
```
avgZ(4,1.5,False)
avgZ(4,1.5,True)
```

```
1 * |00001>
```



```
1 * |00001>
```



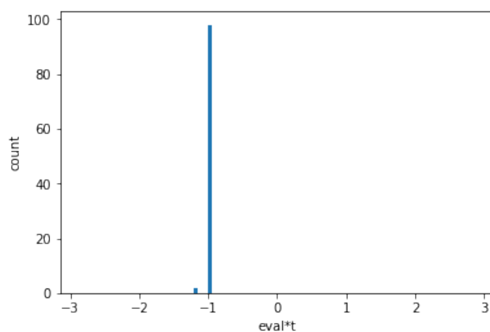
In [92]:
```
def phaseEstimate(op,xbits,cuj):
    N=len(xbits)
    for i in reversed(range(N)):
        op=op.H(xbits[i])
        op=cuj(xbits[i],2**i,op)
    op=qft(op,mask=xbits,inverse=True)
    return op

def period(x):
    if x > np.pi:
        return x - 2*np.pi
    return x

def measure(Nxbits,Nsites,Nmeasure,cuj,prep=None):
    Nbits=Nxbits+Nsites

    sbits=list(range(0,Nsites))

    xbits=list(range(Nsites,Nbits))

    st0=sqc.state(Nbits,basis=["|%g>|%s>" % ( period(2.*np.pi*(i//2**Nsites) / 2**Nxbits),bin(i%2**
    if prep == None:
        st0=sqc.operator(Nbits)*st0
    else:
        st0=prep*st0
```

```
    print("Initial = 0\n",st0)

    st1=phaseEstimate(sqc.operator(Nbits),xbits,cuj)*st0

    if Nmeasure == 0:
        print("State after phaseEstimate\n",st1)
    else:
        res=sqc.sample(st1,Nmeasure,mask=xbits)

        plt.bar([ period(2.*np.pi*x / 2**Nxbits) for x in res.keys() ],res.values(),width=0.06)
        plt.xlabel('eval*t')
        plt.xlim(-np.pi,np.pi)
        plt.ylabel('count')
        plt.show()
```

In [81]:

```
# two-site model at g=0 has evals at +-1/2
measure(5,2,100,lambda c,n,op: CT(c,range(2),40,2*n,op,[0.0,False,True]))
```

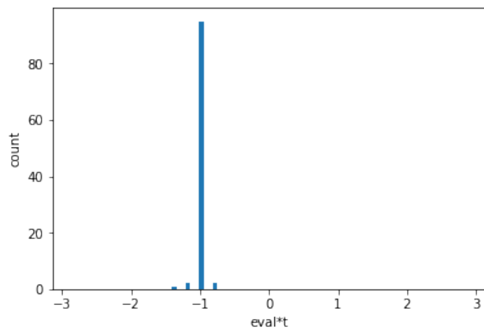
```
Initial = 0
    1 * |0>|0b0>
```



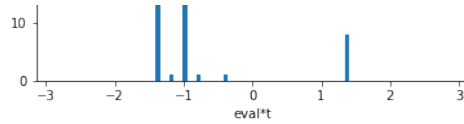
In [84]:

```
# two-site model at g=0 has evals at +-1/2
measure(5,2,100,lambda c,n,op: CT(c,range(2),40,2*n,op,[0.0,False,True]))
measure(5,2,100,lambda c,n,op: CT(c,range(2),40,2*n,op,[0.5,False,True]))
measure(5,2,100,lambda c,n,op: CT(c,range(2),40,2*n,op,[0.7,False,True]))
```

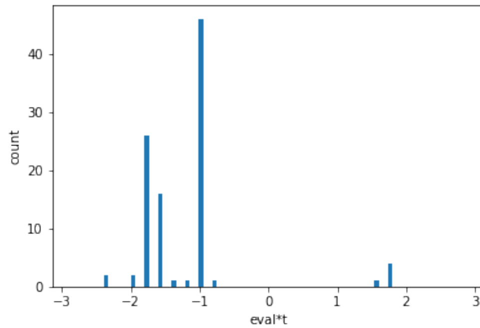
```
Initial = 0
    1 * |0>|0b0>
```



```
Initial = 0
    1 * |0>|0b0>
```




```
Initial = 0
    1 * |0>|0b0>
```

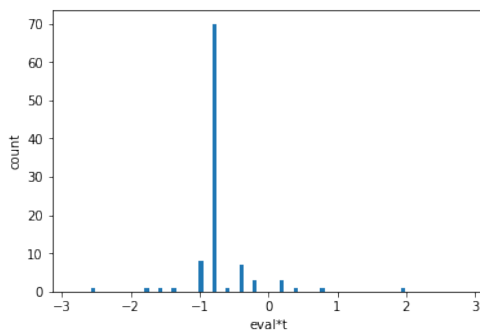


In [97]:
```
measure(5,4,100,lambda c,n,op: CT(c,range(4),40,0.5*n,op,[0.5,False,True]))

# Adiabatic-type method with a single time step
pr=CT(8,range(4),40,1.0,sqc.operator(9).X(8),[0.25,False,True]).X(8)
measure(5,4,100,lambda c,n,op: CT(c,range(4),40,0.5*n,op,[0.5,False,True]),pr)

# Adiabatic-type method with two steps
pr=CT(8,range(4),40,1.0,sqc.operator(9).X(8),[0.1666,False,True])
pr=CT(8,range(4),40,1.0,pr,[0.33333,False,True])
pr=pr.X(8)
measure(5,4,100,lambda c,n,op: CT(c,range(4),40,0.5*n,op,[0.5,False,True]),pr)
```



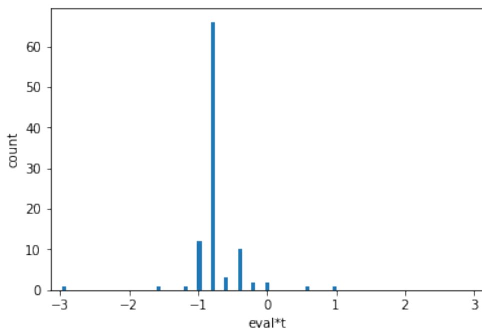
```
# We can also print out the state vector to get a better idea of the corresponding eigenstates:
#measure(5,4,0,lambda c,n,op: CT(c,range(4),40,0.5*n,op,[0.5,False,True]))
```

```
Initial = 0
    1 * |0>|0b0>
```

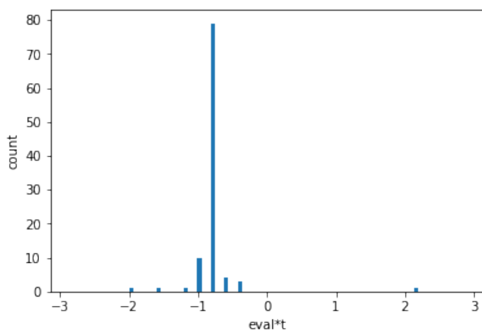



```
Initial = 0
    (0.0557817-0.973315j)      * |0>|0b0>
 + (0.0991756+0.0623829j)      * |0>|0b1>
 + (0.0495826+0.0896233j)      * |0>|0b10>
 + (-0.0111469+0.00894216j)    * |0>|0b11>
 + (0.0495826+0.0896233j)      * |0>|0b100>
 + (-0.0123218)                * |0>|0b101>
 + (-0.0135016+0.0021285j)     * |0>|0b110>
 + (-0.000580179-0.00163434j)  * |0>|0b111>
 + (0.0991756+0.0623829j)      * |0>|0b1000>
 + (-0.0123224+0.00681333j)    * |0>|0b1001>
 + (-0.0123218)                * |0>|0b1010>
 + (-0.000292665-0.0016942j)   * |0>|0b1011>
 + (-0.0111469+0.00894216j)    * |0>|0b1100>
```

```
+ (-0.000292665-0.0016942j)  * |0>|0b1101>
+ (-0.000580179-0.00163434j) * |0>|0b1110>
+ (0.000192028-0.000106915j) * |0>|0b1111>
```




```
Initial = 0
    (-0.938018-0.0679972j)    * |0>|0b0>
 + (0.158204-0.123527j)      * |0>|0b1>
 + (0.115664+0.00404565j)    * |0>|0b10>
 + (0.00301014+0.0430079j)   * |0>|0b11>
 + (0.115664+0.00404565j)    * |0>|0b100>
 + (-0.019009+0.0155275j)    * |0>|0b101>
 + (-0.0244576+0.0207519j)   * |0>|0b110>
 + (-0.00776577-0.00504017j) * |0>|0b111>
 + (0.158204-0.123527j)      * |0>|0b1000>
 + (-0.00734637+0.0422007j)  * |0>|0b1001>
 + (-0.019009+0.0155275j)    * |0>|0b1010>
 + (-0.00656182-0.00646598j) * |0>|0b1011>
 + (0.00301014+0.0430079j)   * |0>|0b1100>
 + (-0.00656182-0.00646598j) * |0>|0b1101>
 + (-0.00776577-0.00504017j) * |0>|0b1110>
 + (0.000954531-0.00236738j) * |0>|0b1111>
```

3.2 One-dimensional free particle

- Consider the Hamiltonian

$$H = \frac{\hat{p}^2}{2m} \tag{3.13}$$

with

$$\hat{p} = \frac{2\pi}{2^N} \mathrm{FT}^\dagger \hat{x} \mathrm{FT} \tag{3.14}$$

and Quantum Fourier Transform matrix FT for a system with $L = 2^N$ sites encoded in N qubits.

- We map out the Hilbert space in position space using

$$\hat{x} = \sum_{i=0}^{N-1} w_i \frac{\mathbb{1} - Z_i}{2} \tag{3.15}$$

with N qubits and

$$w_i = \begin{cases} 2^i \text{ for } i < N-1 \\ -2^i \text{ for } i = N-1 \end{cases} . \tag{3.16}$$

In this way

$$\hat{x} \left| x \right\rangle = x \left| x \right\rangle \tag{3.17}$$

with $x \in \{-2^{N-1}, \ldots, 2^{N-1} - 1\}$.

- We have

$$T = e^{itH} = \mathrm{FT}^\dagger e^{i \frac{t}{2m} \left(\frac{2\pi}{2^N}\right)^2 \hat{x}^2} \mathrm{FT} \,. \tag{3.18}$$

Furthermore

$$\hat{x}^2 = \sum_{i,j=0}^{N-1} \frac{w_i w_j}{4} (\mathbb{1} - Z_i - Z_j + Z_i Z_j) \tag{3.19}$$

$$= \frac{1}{4} + \frac{1}{2} \sum_{i=0}^{N-1} w_i Z_i + \sum_{i,j=0}^{N-1} \frac{w_i w_j}{4} Z_i Z_j \tag{3.20}$$

$$= \frac{1}{12}(2 + 4^N) + \frac{1}{2} \sum_{i=0}^{N-1} w_i Z_i + \sum_{i,j=0; i \neq j}^{N-1} \frac{w_i w_j}{4} Z_i Z_j \tag{3.21}$$

$$\tag{3.22}$$

using

$$\sum_{i=0}^{N-1} w_i = -1\,. \tag{3.23}$$

Therefore

$$e^{ia\hat{x}^2} = e^{i\frac{a(2+4^N)}{12}} \left[\prod_{j=0}^{N-1} e^{i\frac{aw_j}{2}Z_j}\right] \left[\prod_{l,j=0;l\neq j}^{N-1} e^{i\frac{aw_l w_j}{4}Z_l Z_j}\right] \tag{3.24}$$

since all Z_i commute.

- Similarly to the gates derived in the previous section, we find

$$e^{i\theta Z_j} = \cos(\theta) + i\sin(\theta)Z_j = X_j R_{z,j}(\theta) X_j R_{z,j}(-\theta) \tag{3.25}$$

$$= e^{i\theta} R_{z,j}(-2\theta)\,. \tag{3.26}$$

- Furthermore, it is instructive to consider

$$e^{ia\hat{x}} = e^{-\frac{1}{2}ia} \left[\prod_{j=0}^{N-1} e^{-\frac{1}{2}iaw_j Z_j}\right] \tag{3.27}$$

$$= e^{-\frac{1}{2}ia} \left[\prod_{j=0}^{N-1} e^{-\frac{1}{2}iaw_j} R_{z,j}(aw_j)\right] \tag{3.28}$$

$$= \prod_{j=0}^{N-1} R_{z,j}(aw_j)\,. \tag{3.29}$$

- Therefore

$$e^{i\hat{p}} = \mathrm{FT}^\dagger \left[\prod_{j=0}^{N-1} R_{z,j}\left(\frac{2\pi}{2^N} w_j\right)\right] \mathrm{FT} \tag{3.30}$$

$$= \mathrm{FT}^\dagger \left[\prod_{j=0}^{N-1} R_{z,j}\left(\frac{2\pi}{2^{N-j}}\right)\right] \mathrm{FT} \tag{3.31}$$

since $e^{2\pi i} = 1$. Note that this is just the addition through FT circuit of Sec. 1.2.6 (note the conventions for most-significant bits), i.e., as expected $e^{i\hat{p}}$ generates translations

$$e^{i\hat{p}} |x\rangle = |(x+1) \bmod L\rangle \tag{3.32}$$

with

$$\hat{x}\,|x\rangle = x\,|x\rangle\ . \tag{3.33}$$

- We will consider a simple wave packet of the form

$$|\psi(n,p)\rangle = \sum_{x=0}^{2^n-1} e^{-ipx}\,|x\rangle \tag{3.34}$$

$$= \left[\prod_{i=0}^{n-1} R_{z,i}(-p2^i)H_i\right]|0\rangle \tag{3.35}$$

  to study propagation of spatially-smeared moving particles.

- Now implement numerically (**one-dim-free-particle.ipynb**):
  - Free-particle transfer matrix
  - Propagation of localized particle (uncertainty principle)
  - Propagation of spatially-smeared particle

In [5]:
```
import sqc
import numpy as np
from exercises import CRz,qft,cqft
import matplotlib.pyplot as plt
import matplotlib.animation as animation
```

In [6]:
```
def period(x,L):
    if x >= L/2:
        return x - L
    return x

def w(j,N):
    assert(j<N and j>=0)
    if j == N-1:
        return -2**j
    return 2**j

def CexpIaX(c,a,xbits,op):
    N=len(xbits)
    for j in range(len(xbits)):
        op=CRz(c,xbits[j],a*w(j,N),op)
    return op

def CexpIaP(c,a,xbits,op):
    N=len(xbits)
    op=qft(op,mask=xbits,inverse=False)
    op=CexpIaX(c,a*2.*np.pi / 2**N,xbits,op)
    op=qft(op,mask=xbits,inverse=True)
    return op

nxbits=5
nabits=1
nbits=nxbits+nabits
s0=sqc.state(nbits,basis=["|%d>|%d>" % (x//2**nxbits,period(x%2**nxbits,2**nxbits))
                          for x in range(2**nbits)])
```



```
s1=sqc.operator(nbits).X(1).X(2).X(nxbits)*s0
print(s1)
s2=CexpIaP(nxbits,11,range(nxbits),sqc.operator(nbits))*s1
print(s2)
```

```
1 * |1>|6>
1 * |1>|-15>
```

In [7]:
```
def CexpIaX2(c,a,xbits,op):
    N=len(xbits)
    l=a/12.*(2.+4.**N)-a/2.
    op=CRz(c,xbits[0],l,CRz(c,xbits[0],l,op.X(xbits[0])).X(xbits[0]))
    for j in range(len(xbits)):
        op=CRz(c,xbits[j],-a*w(j,N),op)
        for i in range(len(xbits)):
            if i != j:
                t=a/4.*w(j,N)*w(i,N)
                ga=xbits[i]
                gb=xbits[j]
                op=op.CNOT(gb,ga)
                op=op.X(ga)
                op=CRz(c,ga,t,op)
                op=op.X(ga)
                op=CRz(c,ga,-t,op)
                op=op.CNOT(gb,ga)
    return op

def CexpIaP2(c,a,xbits,op):
    N=len(xbits)
    op=qft(op,mask=xbits,inverse=False)
    op=CexpIaX2(c,a*(2.*np.pi / 2**N)**2.,xbits,op)
    op=qft(op,mask=xbits,inverse=True)
    return op
```

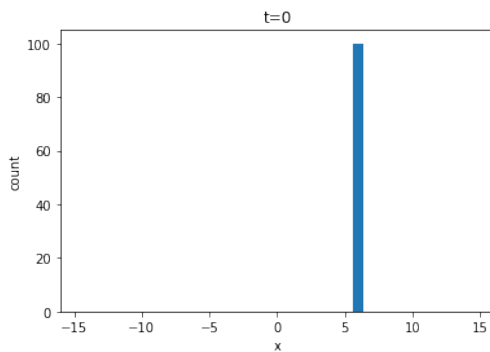
In [8]:
```
s3=CexpIaX2(nxbits,1,range(nxbits),sqc.operator(nbits))*s1
print(np.log(s3[6+2**nxbits])/1j+12.*np.pi)
```

```
(36-7.389922007661198e-16j)
```

In [9]:
```
def timeEvolutionPlot(dt,steps,s1):
    Tdt=CexpIaP2(nxbits,dt,range(nxbits),sqc.operator(nbits))
    print(len(Tdt.m))
    sn=s1
    for tstep in range(steps):
        res=sqc.sample(sn, n=100, mask=range(nxbits))
        sn=Tdt*sn

        xvals=[ period(x,2**nxbits) for x in res.keys() ]
        yvals=res.values()

        plt.xlabel('x')
        plt.xlim(-2**nxbits/2,2**nxbits/2)
        plt.ylabel('count')
        plt.title('t=%g' % (tstep*dt))
        plt.bar(xvals,yvals,width=0.75)
        plt.show()
```

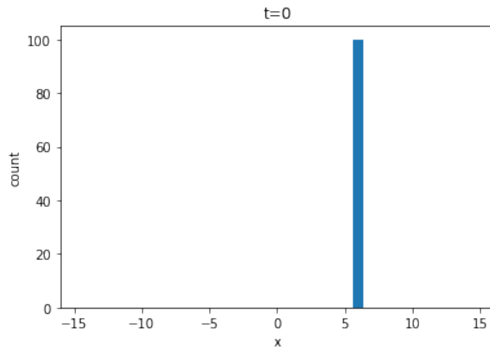

In [10]:
```
print(s1)
timeEvolutionPlot(0.1,30,s1)
```

```
   1 * |1>|6>
439
```



In [11]:
```
# Time-periodicity: all eigenvalues are multiple of (2pi/L)**2 -> once smallest eval has 2pi-period
# entire system has 2pi-periodicity -> temporal loop
Tcrit=2*np.pi / (2*np.pi/32)**2.
timeEvolutionPlot(Tcrit/20,21,s1)
```

```
439
```

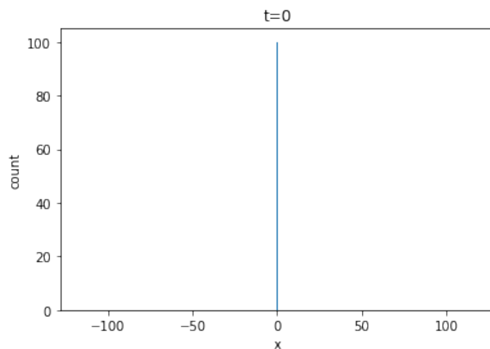


In [12]:
```
# Now discuss circuit to create smeared packet
#def Rot(i,t,op):
#    return op.Rz(i,-np.pi/2).H(i).Rz(i,-2*t).H(i).Rz(i,t).X(i).Rz(i,t).X(i).Rz(i,np.pi/2)

#s1=CexpIaP(nxbits,0.5,range(nxbits),sqc.operator(nbits).X(0).X(nxbits))*s0

nxbits=8
nabits=1
nbits=nxbits+nabits
s0=sqc.state(nbits,basis=["|%d>|%d>" % (x//2**nxbits,period(x%2**nxbits,2**nxbits))
                              for x in range(2**nbits)])
```



```
def createSource(xbits,p,op):
    n=len(xbits)
    for i in range(n):
        op=op.H(xbits[i]).Rz(xbits[i],-p*2**i)
    return op

s1=createSource(range(0),2*np.pi/2**nxbits*64,sqc.operator(nbits).X(nxbits))*s0
print(s1)

timeEvolutionPlot(5,4,s1)
#s2=expIaP2(0.3,range(nxbits),sqc.operator(nxbits))*s1
#s3=qft(sqc.operator(nxbits),mask=range(nxbits),inverse=False)*s2
#timeEvolutionPlot(0.1,30,s3)
```

```
   1 * |1>|0>
1156
```



In [13]:
```
s1=createSource(range(4),2*np.pi/2**nxbits*64,sqc.operator(nbits).X(nxbits))*s0
print(s1)

timeEvolutionPlot(5,4,s1)
```

```
   0.25    * |1>|0>
 + -0.25j  * |1>|1>
 + (-0.25) * |1>|2>
 + 0.25j   * |1>|3>
 + 0.25    * |1>|4>
 + -0.25j  * |1>|5>
 + (-0.25) * |1>|6>
 + 0.25j   * |1>|7>
 + 0.25    * |1>|8>
 + -0.25j  * |1>|9>
 + (-0.25) * |1>|10>
 + 0.25j   * |1>|11>
 + 0.25    * |1>|12>
 + -0.25j  * |1>|13>
 + (-0.25) * |1>|14>
 + 0.25j   * |1>|15>
1156
```

3.3 One-dimensional particle in time-independent potential

- Consider the Hamiltonian

$$H = \frac{\hat{p}^2}{2m} + V(\hat{x})\,. \tag{3.36}$$

- We construct the transfer matrix again by trotterizing the two

$$T_0(t) = e^{i\frac{\hat{p}^2}{2m}t} \tag{3.37}$$

  and

$$T_1(t) = e^{iV(\hat{x})t}\,. \tag{3.38}$$

- First test case: potential wall with

$$V(\hat{x}) = \begin{cases} V_0 & \text{for } x_0 \leq x \leq x_1\,, \\ 0 & \text{else} \end{cases}\,. \tag{3.39}$$

  For appropriate choices of x_0 and x_1 this can simply be implemented by a controlled R_z.

  Now implement numerically (**one-dim-particle-time-independent-potential.ipynb**):

  - Transfer matrix
  - Propagation of wave packet and scattering for $V \gg 0$, $V > 0$ and $V < 0$

- Next example: Harmonic oscillator

$$V(x) = \frac{1}{2}m\omega^2 x^2 \tag{3.40}$$

  Now implement numerically (**one-dim-particle-time-independent-potential.ipynb**):

  - Transfer matrix
  - Propagation of wave packet, eigenvalues and states

In [1]:
```
import sqc
import numpy as np
from exercises import CRz,qft,C2NOT
import matplotlib.pyplot as plt
```

In [2]:
```
# Only implement improved trotter
def CT(c,xbits,Ntrot,dt,op,params):
    op=CT0(c,xbits,dt/Ntrot/2.,op,params)
    for i in range(Ntrot-1):
        op=CT1(c,xbits,dt/Ntrot,op,params)
        op=CT0(c,xbits,dt/Ntrot,op,params)
    op=CT1(c,xbits,dt/Ntrot,op,params)
    op=CT0(c,xbits,dt/Ntrot/2.,op,params)
    return op

# Helper
def period(x,L):
    if x >= L/2:
        return x - L
    return x

def w(j,N):
    assert(j<N and j>=0)
    if j == N-1:
        return -2**j
    return 2**j

# exp(I a x)
def CexpIaX(c,a,xbits,op):
    N=len(xbits)
    for j in range(len(xbits)):
        op=CRz(c,xbits[j],a*w(j,N),op)
    return op

# exp(I a p)
```



```
def CexpIaP(c,a,xbits,op):
    N=len(xbits)
    op=qft(op,mask=xbits,inverse=False)
    op=CexpIaX(c,a*2.*np.pi / 2**N,xbits,op)
    op=qft(op,mask=xbits,inverse=True)
    return op

# exp(I a x^2)
def CexpIaX2(c,a,xbits,op):
    N=len(xbits)
    l=a/12.*(2.+4.**N)-a/2.
    op=CRz(c,xbits[0],l,CRz(c,xbits[0],l,op.X(xbits[0])).X(xbits[0]))
    for j in range(len(xbits)):
        op=CRz(c,xbits[j],-a*w(j,N),op)
        for i in range(len(xbits)):
            if i != j:
                t=a/4.*w(j,N)*w(i,N)
                ga=xbits[i]
                gb=xbits[j]
                op=op.CNOT(gb,ga)
                op=op.X(ga)
                op=CRz(c,ga,t,op)
                op=op.X(ga)
                op=CRz(c,ga,-t,op)
                op=op.CNOT(gb,ga)
    return op

# exp(I a p^2)
def CexpIaP2(c,a,xbits,op):
    N=len(xbits)
    op=qft(op,mask=xbits,inverse=False)
    op=CexpIaX2(c,a*(2.*np.pi / 2**N)**2.,xbits,op)
    op=qft(op,mask=xbits,inverse=True)
    return op

# Simple potential
def C2Rz(c0,c1,t,phi,o):
```

```
    o=CRz(c0,t,phi/2.,o)
    o=C2NOT(c0,c1,t,o)
    o=CRz(c0,t,-phi/2.,o)
    o=CRz(c0,c1,phi/2.,o)
    o=C2NOT(c0,c1,t,o)
    return o
```

In [3]:
```
# Kinetic term
def CT0(c,xbits,dt,op,params):
    mass=params[0]
    return CexpIaP2(c,dt/2.0/mass,xbits,op)

# Potential term
def CT1(c,xbits,dt,op,params):
    V0=params[1]
    c0=xbits[-1]
    c1=xbits[-2]
    return C2Rz(c,c0,c1,V0*dt,op.X(c0)).X(c0)
```

In [4]:
```
nabits=1
nxbits=5
nbits=nabits+nxbits
s0=sqc.state(nbits,basis=["|%d>|%d>" % (x//2**nxbits,period(x%2**nxbits,2**nxbits))
                          for x in range(2**nbits)])
s1=CT1(nxbits,range(nxbits),1.,sqc.operator(nbits).H(0).H(1).H(2).H(3).H(4).X(nxbits),[1,1])*s0
print(s1)
```

```
   0.176777              * |1>|0>
 + 0.176777              * |1>|1>
 + 0.176777              * |1>|2>
 + 0.176777              * |1>|3>
 + 0.176777              * |1>|4>
 + 0.176777              * |1>|5>
 + 0.176777              * |1>|6>
 + 0.176777              * |1>|7>
 + (0.0955129+0.148752j) * |1>|8>
```



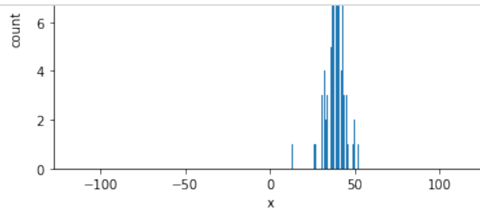
```
 + (0.0955129+0.148752j) * |1>|9>
 + (0.0955129+0.148752j) * |1>|10>
 + (0.0955129+0.148752j) * |1>|11>
 + (0.0955129+0.148752j) * |1>|12>
 + (0.0955129+0.148752j) * |1>|13>
 + (0.0955129+0.148752j) * |1>|14>
 + (0.0955129+0.148752j) * |1>|15>
 + 0.176777              * |1>|-16>
 + 0.176777              * |1>|-15>
 + 0.176777              * |1>|-14>
 + 0.176777              * |1>|-13>
 + 0.176777              * |1>|-12>
 + 0.176777              * |1>|-11>
 + 0.176777              * |1>|-10>
 + 0.176777              * |1>|-9>
 + 0.176777              * |1>|-8>
 + 0.176777              * |1>|-7>
 + 0.176777              * |1>|-6>
 + 0.176777              * |1>|-5>
 + 0.176777              * |1>|-4>
 + 0.176777              * |1>|-3>
 + 0.176777              * |1>|-2>
 + 0.176777              * |1>|-1>
```

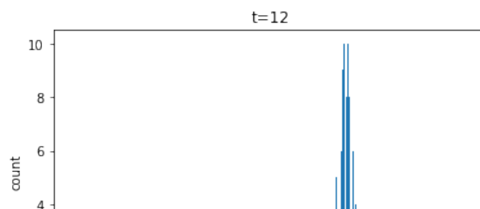
In [5]:
```
def timeEvolutionPlot(dt,steps,s1,Ntrot,params):
    Tdt=CT(nxbits,range(nxbits),Ntrot,dt,sqc.operator(nbits),params)
    print(len(Tdt.m))
    sn=s1
    for tstep in range(steps):
        res=sqc.sample(sn, n=100, mask=range(nxbits))
        sn=Tdt*sn

        xvals=[ period(x,2**nxbits) for x in res.keys() ]
        yvals=res.values()

        plt.xlabel('x')
        plt.xlim(-2**nxbits/2,2**nxbits/2)
```

```
        plt.ylabel('count')
        plt.title('t=%g' % (tstep*dt))
        plt.bar(xvals,yvals,width=0.75)
        plt.show()

def createSource(xbits,p,op):
    n=len(xbits)
    for i in range(n):
        op=op.H(xbits[i]).Rz(xbits[i],-p*2**i)
    return op


nabits=1
nxbits=8
nbits=nabits+nxbits
s0=sqc.state(nbits,basis=["|%d>|%d>" % (x//2**nxbits,period(x%2**nxbits,2**nxbits))
                          for x in range(2**nbits)])

s1=createSource(range(4),2*np.pi/2**nxbits*64,sqc.operator(nbits).X(nxbits))*s0
print(s1)
```

```
   0.25    * |1>|0>
 + -0.25j  * |1>|1>
 + (-0.25) * |1>|2>
 + 0.25j   * |1>|3>
 + 0.25    * |1>|4>
 + -0.25j  * |1>|5>
 + (-0.25) * |1>|6>
 + 0.25j   * |1>|7>
 + 0.25    * |1>|8>
 + -0.25j  * |1>|9>
 + (-0.25) * |1>|10>
 + 0.25j   * |1>|11>
 + 0.25    * |1>|12>
 + -0.25j  * |1>|13>
 + (-0.25) * |1>|14>
 + 0.25j   * |1>|15>
```

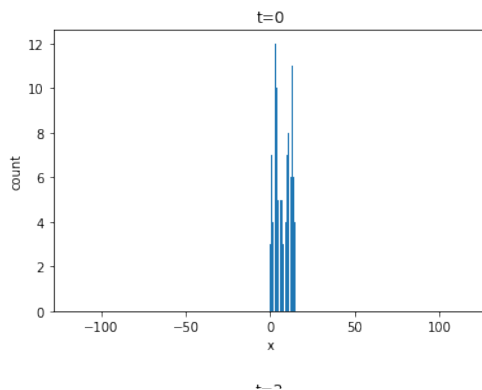

In [6]:
```
timeEvolutionPlot(2,30,s1,10,[0.5,3.5]) # Hard wall
```





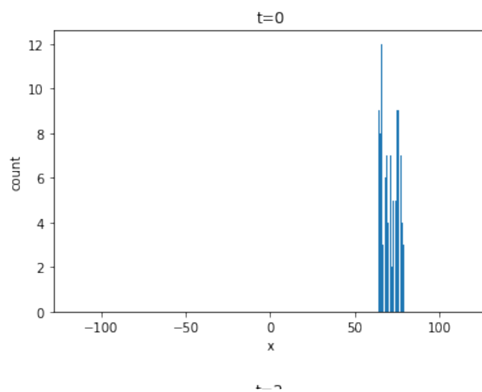
In [7]:
```
timeEvolutionPlot(2,30,s1,10,[0.5,2.0])
```

14066



In [8]:
```
timeEvolutionPlot(2,30,sqc.operator(nbits).X(6)*s1,15,[0.5,-3.0])
```

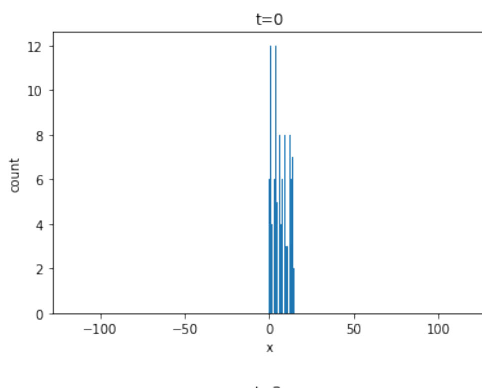
20521



In [9]:
```
# Next: harmonic oscillator, first propagation, then eigenspectrum
def CT1(c,xbits,dt,op,params):
    mass=params[0]
    omega=params[1]
    return CexpIaX2(c,dt*mass/2.*omega**2.,xbits,op)
```

In [10]:
```
timeEvolutionPlot(2,30,s1,10,[0.5,0.2])
```

```
21076
```



In [31]:
```
def phaseEstimate(op,xbits,cuj):
    N=len(xbits)
    for i in reversed(range(N)):
        op=op.H(xbits[i])
        op=cuj(xbits[i],2**i,op)
    op=qft(op,mask=xbits,inverse=True)
    return op

def periodPi(x):
    if x > np.pi:
        return x - 2*np.pi
    return x

def measure(Nxbits,Nsites,Nmeasure,cuj,prep=None,x0=-np.pi,x1=np.pi):
```



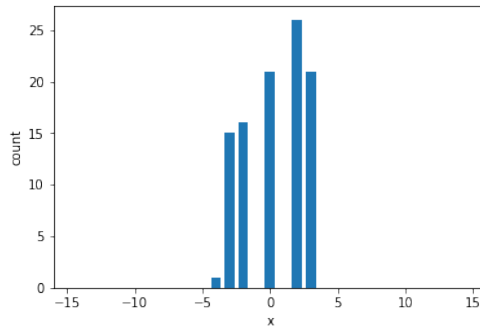
```
    Nbits=Nxbits+Nsites

    sbits=list(range(0,Nsites))

    xbits=list(range(Nsites,Nbits))

    st0=sqc.state(Nbits,basis=["|%g>|%d>" %
        ( periodPi(2.*np.pi*(i//2**Nsites) / 2**Nxbits),
          period(i%2**Nsites,2**Nsites)) for i in range(2**Nbits)])
    if prep == None:
        st0=sqc.operator(Nbits)*st0
    else:
        st0=prep*st0
    print("Initial = 0\n",st0)

    st1=phaseEstimate(sqc.operator(Nbits),xbits,cuj)*st0

    if Nmeasure == 0:
        print("State after phaseEstimate\n",st1)
    else:
        res,resstates=sqc.sample(st1,Nmeasure,mask=xbits,save_states=True)

        plt.bar([ periodPi(2.*np.pi*x / 2**Nxbits) for x in res.keys() ],
                res.values(),width=5.0/2.0**Nxbits)
        plt.xlabel('eval*t')
        plt.xlim(x0,x1)
        plt.ylabel('count')
        plt.show()

        # show two most sampled states
        evs=list(reversed(sorted(list(res.values()))))[0:2]

        for e in evs:
            imax=list(res.values()).index(e)
            ev=resstates[list(res.keys())[imax]]
            print(ev)
```

```
            xres=sqc.sample(ev,Nmeasure,mask=sbits)
            xvals=[ period(x,2**Nsites) for x in xres.keys() ]
            yvals=xres.values()

            plt.xlabel('x')
            plt.xlim(-2**Nsites/2,2**Nsites/2)
            plt.ylabel('count')
            plt.bar(xvals,yvals,width=0.75)
            plt.show()
```

In [32]:

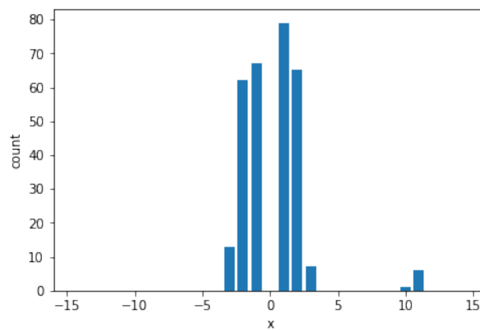
```
np.random.seed(13)
measure(7,5,100,lambda c,n,op: CT(c,range(5),10,0.2*n,op,[0.5,1]))
```

```
 + (-0.190757+0.329582j)      * |0.539961>|-3>
 + (-0.22261+0.443095j)       * |0.539961>|-2>
 + (-0.0177286-0.019724j)     * |0.539961>|-1>
```



In [33]:

```
measure(9,5,300,lambda c,n,op: CT(c,range(5),10,0.2*n,op,[0.5,1]),
        prep=sqc.operator(14).H(0).H(1),x0=0,x1=1)
```




```
   (0.613292+0.157869j)       * |0.110447>|0>
 + (0.470934+0.117019j)       * |0.110447>|1>
 + (0.223008+0.0549725j)      * |0.110447>|2>
```

In [ ]:

3.4 One-dimensional real scalar quantum field theory

- Consider a one-dimensional real scalar quantum field theory with periodic boundary conditions on a spatial lattice with n_s positions defined by the Lagrangian

$$L = \sum_{x=0}^{n_s-1} \frac{1}{2}\phi(x,t)\left[\overleftarrow{\partial}_t \overrightarrow{\partial}_t + \Delta^2 - m^2\right]\phi(x,t) \tag{3.41}$$

with $x \in \{0, \dots, n_s - 1\}$, $t, m \in \mathbb{R}$, and

$$\Delta^2\phi(x,t) \equiv \phi(x+1,t) + \phi(x-1,t) - 2\phi(x,t) \tag{3.42}$$

and $\phi(-1,t) = \phi(n_s-1,t)$, $\phi(n_s,t) = \phi(0,t)$.

- The canonical momentum for the field is then given by the appropriate functional derivative

$$\pi(x,t) = \partial_{\partial_t\phi(x,t)}L = \partial_t\phi(x,t) \tag{3.43}$$

and the Euler-Lagrange equation

$$\partial_{\phi(x,t)}L = \partial_t\pi(x,t) \tag{3.44}$$

with is

$$0 = \partial_t^2\phi(x,t) - \partial_{\phi(x,t)}\sum_{x=0}^{n_s-1}\frac{1}{2}\phi(x,t)\left[\phi(x+1,t)+\phi(x-1,t)-(2+m^2)\phi(x,t)\right] \tag{3.45}$$

$$= \partial_t^2\phi(x,t) - \left[\phi(x+1,t)+\phi(x-1,t)-(2+m^2)\phi(x,t)\right] \tag{3.46}$$

$$= (\partial_t^2 - \Delta^2 + m^2)\phi(x,t)\,. \tag{3.47}$$

- Next, we consider free-field solutions by first performing a discrete Fourier transform

$$\phi(x,t) = \frac{1}{\sqrt{2\pi n_s}}\int dE \sum_{k=0}^{n_s-1} e^{2\pi i x k/n_s} e^{itE}\tilde{\phi}(k,E) \tag{3.48}$$

yielding

$$\left(-E^2 + \left(2\sin\left(\frac{\pi k}{n_s}\right)\right)^2 + m^2\right)\tilde{\phi}(k,E) = 0\,. \tag{3.49}$$

- Defining

$$w_k = \sqrt{m^2 + \left(2\sin\left(\frac{\pi k}{n_s}\right)\right)^2} \tag{3.50}$$

we can then write the free field solution

$$f_k(x,t) \equiv \frac{1}{\sqrt{2w_k n_s}} e^{-iw_k t + i2\pi xk/n_s} \tag{3.51}$$

with

$$(\partial_t^2 - \Delta^2 + m^2) f_k(x,t) = 0 \tag{3.52}$$

and

$$\sum_{x=0}^{n_s-1} f_k^*(x,t) f_{k'}(x,t) = \frac{1}{2w_k}\delta_{k,k'}\,, \tag{3.53}$$

$$\sum_{x=0}^{n_s-1} f_k(x,t) f_{k'}(x,t) = \frac{1}{2w_k}\delta_{k,-k'} e^{-2iw_k t}\,. \tag{3.54}$$

- To quantize the theory, we first make the ansatz for the real field

$$\phi(x,t) = \sum_{k=0}^{n_s-1} \left[f_k(x,t)\hat{a}_k + f_k^*(x,t)\hat{a}_k^\dagger \right] \tag{3.55}$$

and operators $\hat{a}_k$ such that

$$\pi(x,t) = -i \sum_{k=0}^{n_s-1} w_k \left[f_k(x,t)\hat{a}_k - f_k^*(x,t)\hat{a}_k^\dagger \right]\,. \tag{3.56}$$

Then

$$\hat{a}_k = \sum_{x=0}^{n_s-1} f_k^*(x,t)(w_k \phi(x,t) + i\pi(x,t))\,. \tag{3.57}$$

We quantize the fields canonically, i.e., require

$$[\phi(x,t), \pi(y,t)] = i\delta_{x,y}\,, \quad [\phi(x,t), \phi(y,t)] = [\pi(x,t), \pi(y,t)] = 0 \tag{3.58}$$

yielding

$$[\hat{a}_k, \hat{a}_{k'}] = \sum_{x,y=0}^{n_s-1} f_k^*(x,t) f_{k'}^*(y,t) [w_k\phi(x,t) + i\pi(x,t), w_{k'}\phi(y,t) + i\pi(y,t)] \tag{3.59}$$

$$= \sum_{x,y=0}^{n_s-1} f_k^*(x,t) f_{k'}^*(y,t) i \left[w_{k'}[\pi(x,t), \phi(y,t)] + w_k[\phi(x,t), \pi(y,t)] \right] \tag{3.60}$$

$$= \sum_{x=0}^{n_s-1} f_k^*(x,t) f_{k'}^*(x,t) \left[w_{k'} - w_k \right] = 0 \tag{3.61}$$

and also

$$[\hat{a}_k, \hat{a}_{k'}^\dagger] = \sum_{x,y=0}^{n_s-1} f_k^*(x,t) f_{k'}(y,t) i \left[w_{k'}[\pi(x,t), \phi(y,t)] - w_k[\phi(x,t), \pi(y,t)] \right] \tag{3.62}$$

$$= \sum_{x=0}^{n_s-1} f_k^*(x,t) f_{k'}(x,t) \left[w_{k'} + w_k \right] = \delta_{k,k'} \, . \tag{3.63}$$

- Then Hamiltonian is given by a Legendre transformation of the Lagrangian
- This looks like a Harmonic oscillator for each mode number k. Note: we have a vacuum energy offset $E_0 = \frac{1}{2} \sum_{k=0}^{n_s-1} w_k$.
- In a quantum field theory, particles can be created and destroyed. Relevant degrees of freedom for non-interacting theory: occupation number

$$\hat{n}_k \equiv \hat{a}_k^\dagger \hat{a}_k \tag{3.64}$$

of particles with mode number k.

- Possible implementation on quantum computer (suppress mode number)

$$\hat{a}^\dagger = \sqrt{\hat{x}} e^{i\hat{p}} \, , \qquad \hat{a} = e^{-i\hat{p}} \sqrt{\hat{x}} \tag{3.65}$$

with n_o qubits for

$$\hat{x} = \sum_{i=0}^{n_o-1} 2^i \frac{\mathbb{1} - Z_i}{2} \tag{3.66}$$

and the corresponding $\hat{p}$ defined in the previous section. We identify the eigenstates of $\hat{n}$ with the eigenstates $|0\rangle, \ldots, |2^{n_o} - 1\rangle$ of $\hat{x}$.

This has

$$[\hat{a}, \hat{a}^\dagger] |n\rangle = (e^{-i\hat{p}} \hat{x} e^{i\hat{p}} - \hat{x}) |n\rangle = \begin{cases} 1 \text{ for } n < 2^{n_o} - 1 \\ 1 - 2^{n_o} \text{ else} \end{cases} |n\rangle \tag{3.67}$$

and

$$\hat{n} |n\rangle = n |n\rangle \, , \tag{3.68}$$

$$\hat{a}^\dagger |n\rangle = \sqrt{n + 1 \bmod 2^{n_o}} \, |n + 1 \bmod 2^{n_o}\rangle \, , \tag{3.69}$$

$$\hat{a} |n\rangle = \sqrt{n} \, |n - 1 \bmod 2^{n_o}\rangle \, . \tag{3.70}$$

This approximates the full theory well for sufficiently small energies and therefore occupation numbers.

Note again that $\hat{x}$ and $\hat{p}$ here are operators defined in previous sections and have nothing to do with position and momentum in the quantum field theory.

- Implementation of $e^{i\hat{p}}$ for two qubits is simple CNOT and NOT gates:

$$e^{i\hat{p}} = NOT(0)CNOT(0,1) \tag{3.71}$$

  with control qubit first and target qubit second parameter in CNOT.

  Draw circuit, exercise with occupation numbers from 0 to 3.

- With this, the derivation of the free transfer matrix is again straightforward but will not be shown explicitly here.
- We need n_o qubits per site, so a total of $N = n_o n_s$ qubits.
- Finally, we add an interaction term to the Lagrangian such as a $\phi(x,t)^4$ term. Then need to expand in $\hat{a}_k$ to find transfer matrix.