a

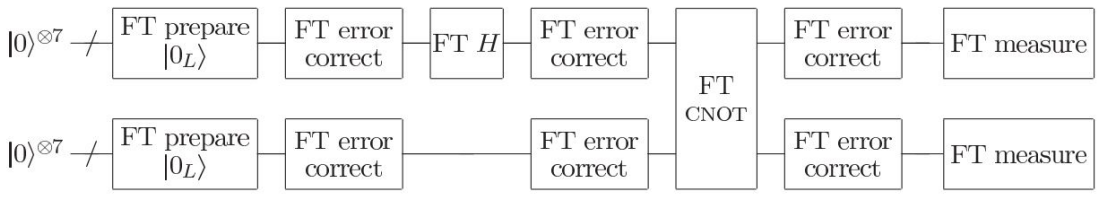
February 8, 2025

[ ]:
```
pip install -U git+https://github.com/MIROptics/ECC2025.git
```

[2]:
```
import numpy as np
import matplotlib.pyplot as plt
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister
from qiskit_aer import AerSimulator
from qiskit_ibm_runtime.fake_provider import FakeRochesterV2
from qiskit import transpile

from ECC2025.testing import test_9a, test_9b, test_9c
```

```
/tmp/ipykernel_55229/3360980799.py:8: DeprecationWarning: Sampler has been
deprecated as of Aer 0.15, please use SamplerV2 instead.
  from ECC2025.testing import test_9a, test_9b, test_9c
```

Los computadores cuánticos son muy susceptibles a error debido. Interacciones no deseadas con el ambiente causan **decoherecia**, donde la información codificada en los qubits se fuga hacia el ambiante. Debido a esto, los dispositivos cuánticos nunca son perfectos y las operaciones cuánticas se implementan con una precision limitada. Para algoritmos cuánticos que requieren miles de puertas cuánticas, esta carencia de presicion se acumula provocando que nuestros resultados se alejen de los esperados. Los **códigos de corrección de errores** asoman como una alternativa para reducir el impacto de estos errores y escalar computadores cuánticos. Estos se basan en codificar estados cuánticos en qubits lógicos, los cuales son conjuntos de qubits físicos. A través de medidas adecuadas sobre esto qubits lógicos podemos detectar la presencia de un error para posteriormente corregirlo. Esta capa de correción se hará después de cada operación en un circuito. La siguiente figura muestra como sería un circuito cuántico para preparar un estado de Bell de 2 qubit lógicos compuestos de 7 qubits físicos:



Actualmente, muchas empresas buscan gente con formación en computación cuántica, y en particular en corrección de errores. Acá les dejamos una lista de algunas ofertas de trabajo para gente con conocimiento en corrección de errores.

Riverlane

Parityqc

Alice and Bob

Ionq

Psiquantum

Rigetti

Quantinuum

QuEra Computing

En este desafio implementaremos el **código de corrección de errores de 9 qubits** propuesto por Peter Shor en 1995.

Empezaremos con el código de 3-qubits para mitigar errores de bit-flip. Consideremos el estado de 1 qubit

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle.$$

Decimos que este estado sufrió un error de bit-flip si el estado es negado, es decir,

$$X|\psi\rangle = \beta|0\rangle + \alpha|1\rangle.$$

Para proteger el estado $|\psi\rangle$ contra este error se realiza la siguiente codificación en 3 qubits,

$$|\psi'\rangle = \alpha|000\rangle + \beta|111\rangle.$$

Para detectar un bit-flip debemos medir los observables IZZ y ZZI, con $Z = |0\rangle\langle 0| - |1\rangle\langle 1|$. A esto se le conoce como **medidas estabilizadoras**. Los resultado de las medidas estabilizadoras nos permiter detectar errores de bit-flip:

1) $\langle IZZ\rangle = 1$ y $\langle ZZI\rangle = 1$, no hay error.

2) $\langle IZZ\rangle = 1$ y $\langle ZZI\rangle = -1$, el tercer qubit tuvo bit flip.

3) $\langle IZZ\rangle = -1$ y $\langle ZZI\rangle = 1$, el primer qubit tuvo bit flip.

4) $\langle IZZ\rangle = -1$ y $\langle ZZI\rangle = -1$, el segundo qubit tuvo bit flip.

Esta medida puede ser implementada sin alterar el estado $|\psi'\rangle$ con 2 qubits auxiliares, uno por cada medida estabilizadora ($clbit_0$ para IZZ y $clbit_1$ para ZZI). Posteriormente podemos corregir los errors aplicando puertas X dependiendo del resultado de la medida estabilizadora:

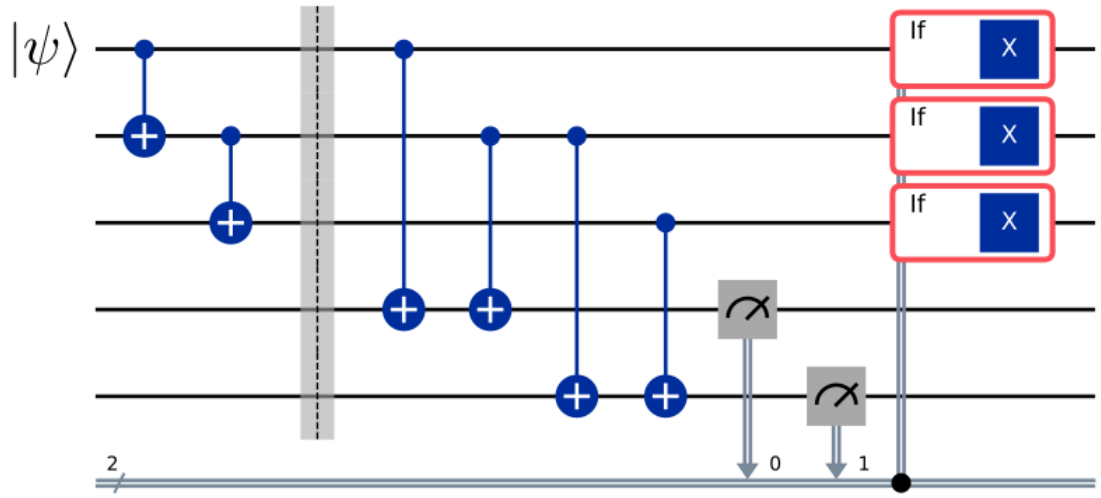
1) Si obtenemos $|00\rangle$, no se aplica ninguna corrección.

2) Si obtenemos $|01\rangle$, se aplica una X en el tercer qubit.

3) Si obtenemos $|10\rangle$, se aplica una X en el primer qubit

4) Si obtenemos $|11\rangle$, se aplica una X en el segundo qubit.

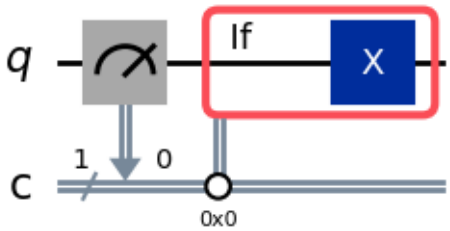
A continuación el circuito que implementa este código:



Desafio: Complete la siguiente función llamada `flip_code` para que implemente el circuito anterior. La función tiene el argumento `error`, el cual es un circuito para introducir el bit flip. Este debe estar en la posición de la barrera. La corrección debe realizarse a través de la función `QuantumCircuit.if_test`, la cual permite implementar puerta controladas clasicamente. A continuación un ejemplo:

[3]:
```
qr = QuantumRegister(1, 'q')
cr = ClassicalRegister(1, 'c')
qc = QuantumCircuit(qr,cr)
qc.measure(qr,cr)

with qc.if_test( (cr,0) ):
        qc.x(0)

qc.draw('mpl')
```

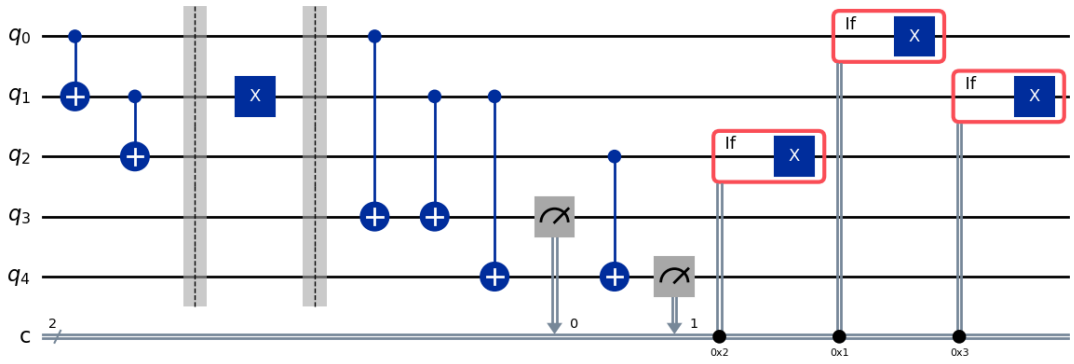
[3]:



[4]:
```
def flip_code( error=None ):

    qr = QuantumRegister(5, 'q')
    cr = ClassicalRegister(2, 'c')
```

```
    qc = QuantumCircuit( qr, cr )

    if error is None:
        error = QuantumCircuit( qr, cr )

    ### Acá va la codificación
    qc.cx(0,1)
    qc.cx(1,2)
    ###
    qc.barrier()
    qc.compose( error, inplace=True )
    qc.barrier()
    ### Acá va la medida estabilizadora
    qc.cx(0,3)
    qc.cx(1,3)
    qc.cx(1,4)
    qc.cx(2,4)
    qc.measure([3,4],[0,1])
    ### Acá va el control clásico
    with qc.if_test( (cr,2) ):
        qc.x(2)
    with qc.if_test( (cr,1) ):
        qc.x(0)
    with qc.if_test( (cr,3) ):
        qc.x(1)

    return qc

# Esto es un bit flip
qr = QuantumRegister(5, 'q')
cr = ClassicalRegister(2, 'c')
error = QuantumCircuit( qr, cr )
error.x(1)

qc_flip = flip_code(error)
qc_flip.draw('mpl',fold=-1)
```

[4]:



[5]:
```
test_9a( qc_flip )
```

```
Felicidades, tu código corrige amplitud
```

Con un código similar podemos corregir el error de phase flip, el cual se define como un cambio de signo en el estado,

$$Z|\psi\rangle = \alpha|0\rangle - \beta|1\rangle.$$

En este caso, la codificación es

$$|\psi''\rangle = \alpha|++\rangle + \beta|--\rangle,$$

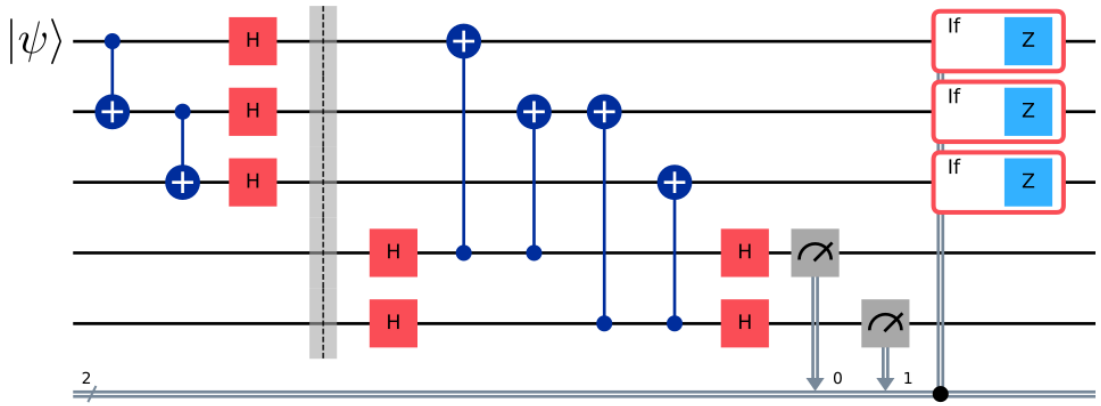
donde $|\pm\rangle = (|0\rangle \pm |1\rangle)/\sqrt{2}$. La medida estabilizadora es $IXX\text{y}XXI$, y la corrección se realiza con puertas Z controladas clasicamente, similar al caso anterior:

1) $\langle IXX\rangle = 1\text{y}\langle XXI\rangle = 1$, no hay error.
2) $\langle IXX\rangle = 1\text{y}\langle XXI\rangle = -1$, el tercer qubit tuvo phase flip.
3) $\langle IXX\rangle = -1\text{y}\langle XXI\rangle = 1$, el primer qubit tuvo phase flip.
4) $\langle IXX\rangle = -1\text{y}\langle XXI\rangle = -1$, el segundo qubit tuvo phase flip.

La correcciones vienen dadas por:

1) Si obtenemos $|00\rangle$, no se aplica ninguna corrección.
2) Si obtenemos $|01\rangle$, se aplica una Z en el tercer qubit.
3) Si obtenemos $|10\rangle$, se aplica una Z en el primer qubit
4) Si obtenemos $|11\rangle$, se aplica una Z en el segundo qubit.

Esto se implementa con el siguiente circuito:

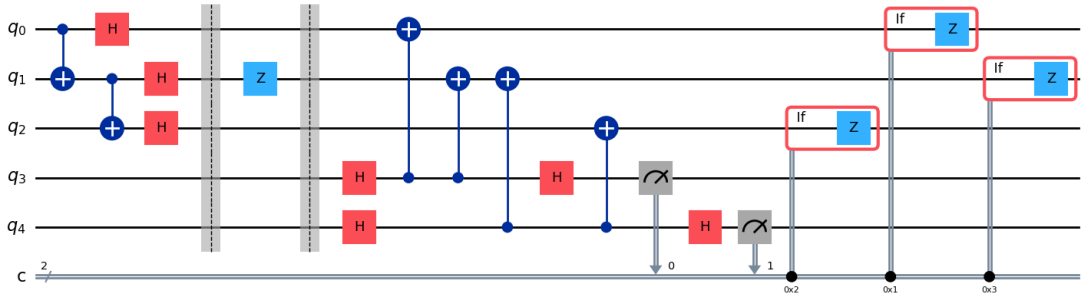


Desafio: Complete la siguiente función llamada `phase_code` para que implemente el circuito anterior. De forma análoga, el argumento `error` es un circuito que introduce un phase plip que debe estar en la posición de la barrera de la figura.

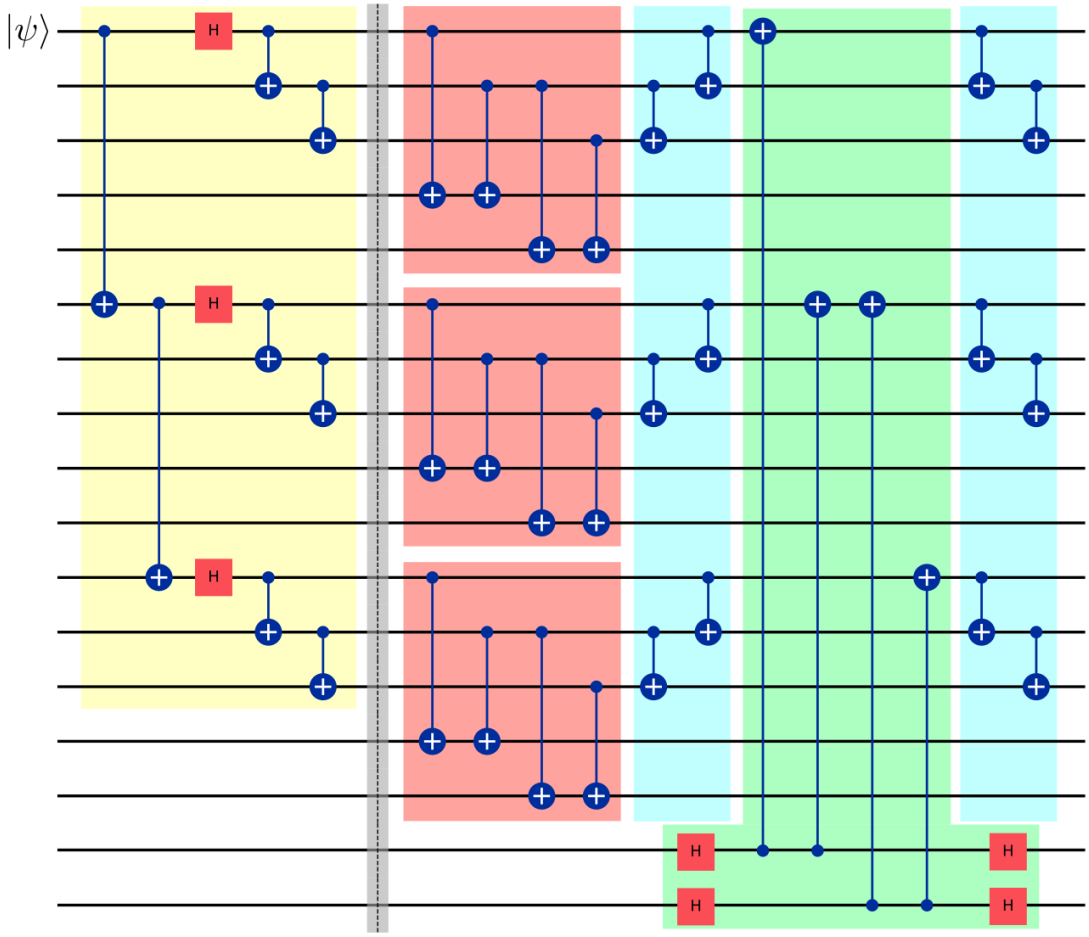
[6]:
```
def phase_code( error=None ):

    qr = QuantumRegister(5, 'q')
    cr = ClassicalRegister(2, 'c')
    qc = QuantumCircuit( qr, cr )

    if error is None:
        error = QuantumCircuit( qr, cr )

    ### Acá va la codificación
    qc.cx(0,1)
    qc.cx(1,2)
    qc.h([0,1,2])
    ###
    qc.barrier()
    qc.compose( error, inplace=True )
    qc.barrier()
    ### Acá va la medida estabilizadora
    qc.h([3,4])
    qc.cx(3,0)
    qc.cx(3,1)
    qc.cx(4,1)
    qc.cx(4,2)
    qc.h([3,4])
    qc.measure([3,4],[0,1])
    ### Acá va el control clásico
    with qc.if_test( (cr,2) ):
        qc.z(2)
    with qc.if_test( (cr,1) ):
        qc.z(0)
    with qc.if_test( (cr,3) ):
```

```
        qc.z(1)

    return qc

## Esto es un phase flip
qr = QuantumRegister(5, 'q')
cr = ClassicalRegister(2, 'c')
error = QuantumCircuit( qr, cr )
error.z(1)

qc_phase = phase_code(error)
qc_phase.draw('mpl', fold=-1)
```

[6]:



El código de Shor combina los código de bit y phase flip en un único circuito. Si omitimos las medidas intermedias y las puertas controladas clasicamente, este estada representado por el siguiente circuito:



La sección en amarillo representa la codificación del estado $|\psi\rangle$en 9 qubits. Las secciones en rojo representan 3 bloques de código bit flip, mientras que la sección en verde es un código phase flip entre esos bloques. Las secciones en azul son un decoding-encoding, necesario para aplicar el código phase flip sin romper los códigos bit flip. En total 8 qubits auxiliares son utilizados para implementar las medidas estabilizadoras$\{ZZIIIIIII, IZZIIIIII, IIIZZIIII, IIIIZZIII, IIIIIIZZI, IIIIIIIZZ, XXXXXXIII, III$
Debido a que todo error de 1 qubit puede descomponerse en una combinación de bit flip y phase flip, el código de Shor puede corregir un error arbitrario de 1 qubit.

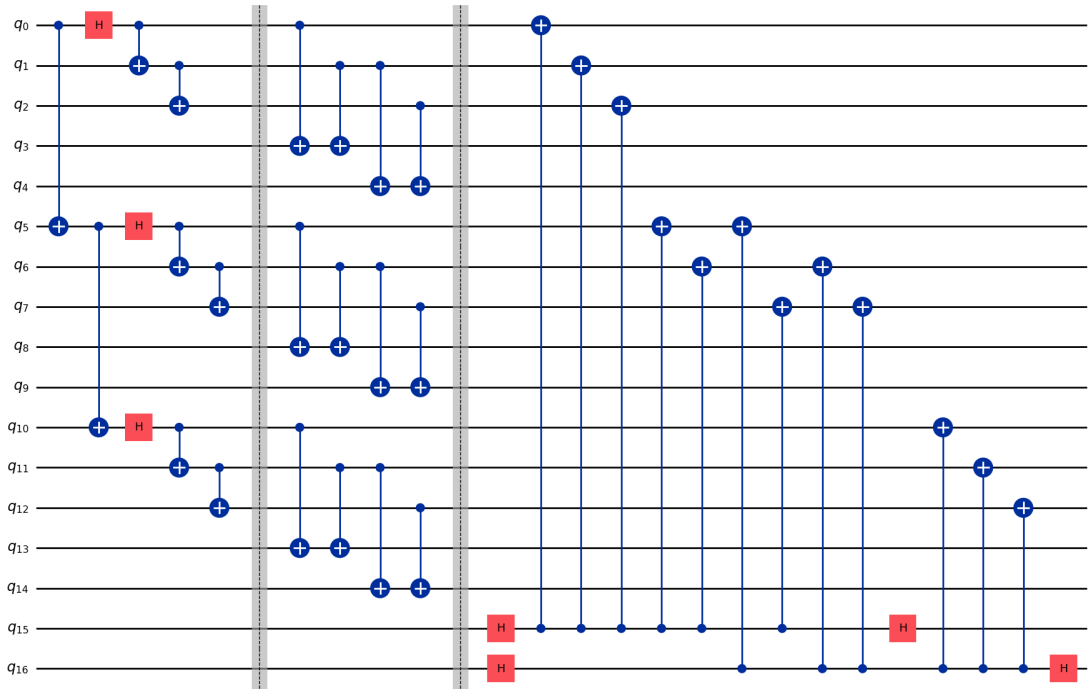
Notemos que en total se requieren 36 puertas CNOTs. Sin embargo, si combinamos las secciones en azul y verde podemos obtener un circuito equivalente que solo requiere 32 puertas CNOTs.

Desafio: Complete la siguiente función llamada `shor_code` para que implemente el circuito anterior, pero optimizado a 32 puertas CNOTs.

[7]:
```
def shor_code():

    qr = QuantumRegister(17, 'q')
    qc = QuantumCircuit( qr )

    ### Acá va tu circuito
```

```
    qc.cx(0,5)
    qc.cx(5,10)
    qc.h([0,5,10])

    qc.cx(0,1)
    qc.cx(1,2)
    qc.cx(5,6)
    qc.cx(6,7)
    qc.cx(10,11)
    qc.cx(11,12)

    qc.barrier()

    for j in [0,5,10]:
        qc.cx(0+j,3+j)
        qc.cx(1+j,3+j)
        qc.cx(1+j,4+j)
        qc.cx(2+j,4+j)

    qc.barrier()
    ###

    qc.h([-2,-1])

    qc.cx(-2, 0)
    qc.cx(-2, 1)
    qc.cx(-2, 2)

    qc.cx(-2, 5)
    qc.cx(-2, 6)
    qc.cx(-2, 7)

    qc.cx(-1, 5)
    qc.cx(-1, 6)
    qc.cx(-1, 7)

    qc.cx(-1, 10)
    qc.cx(-1, 11)
    qc.cx(-1, 12)

    qc.h([-2,-1])


    return qc

qc_shor_code= shor_code()
qc_shor_code.draw('mpl', fold=-1)
```

[7]:



[8]:
```
test_9b( qc_shor_code )
```

```
Felicidades, tu circuito tiene 32 cx
```

Una observación importante es que el código de Shor no es infalible, pues a pesar de que nos protege de errores arbitrarios de 1 qubits, estos solo pueden pasar una vez. Si por ejemplo, el código de Shor no es capaz de detectar 2 bit flips entre qubits en un mismo bloque. Supongamos que p es la probabilidad de que un error ocurra en cada qubit independientemente. Entonces, la probabilidad de que el código de Shor funcione es

$$Pr(\text{no error}) + Pr(1 \text{ error}) = (1-p)^9 + 9p(1-p)^8.$$

Sin el código, el circuito no es afectado por error con probabilidad $1-p$. De este modo, el código ayuda si

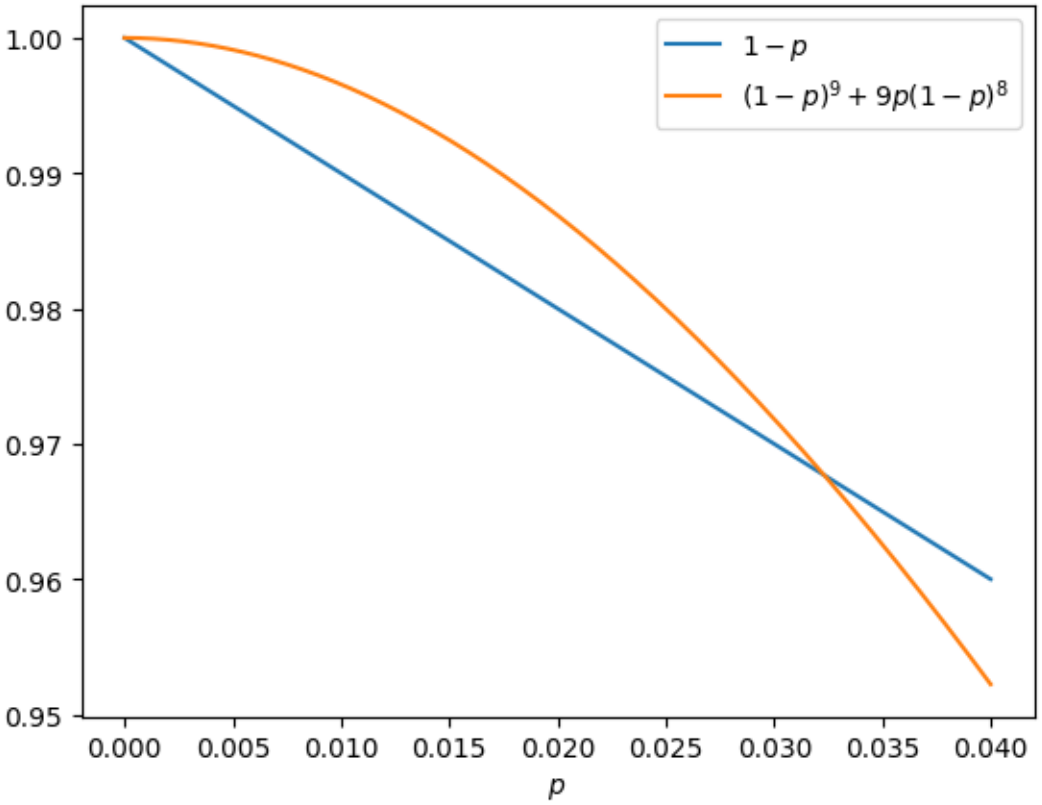
$$(1-p)^9 + 9p(1-p)^8 > 1-p.$$

Estas dos funciones son graficadas en la siguiente celda.

[9]:
```
p = np.linspace(0,0.04,1000)
prob_vanilla = 1 - p
prob_shor    = (1-p)**9 +9*p*(1-p)**8
plt.plot( p, prob_vanilla)
plt.plot( p, prob_shor)
```

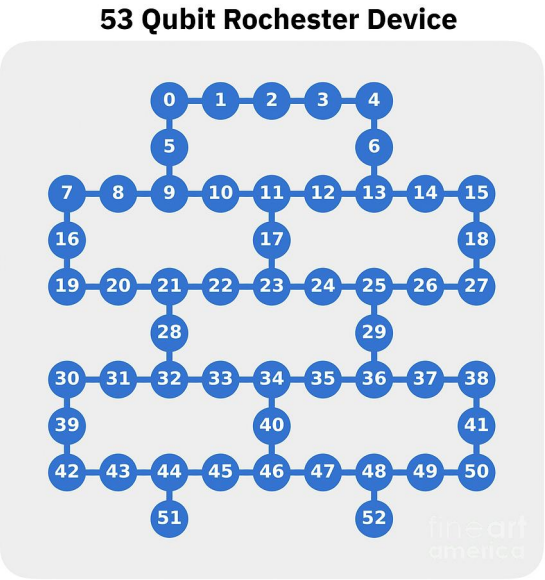
```
plt.legend([r'$1 - p$', r'$(1-p)^9 + 9p(1-p)^8$'])
plt.xlabel(r'$p$')
```

[9]: `Text(0.5, 0, '$p$')`



Podemos ve que hay una region para $p \leq 0.30$ donde el código de Shor aumenta la probabilidad de que el circuito se ejecute correctamente. Sin embargo, para probabilidades de error mayor, el código de Shor funciona peor que el circuito sin corrección. Para estar en la región debemos reducir la probabilidad de error, y esto se logra optimizando el diseño y funcionamiento de los dipositivos cuánticos. Actualmente nos encontramos en la transición de la derecha de la intersección a la izquierda, y se espera que en los próximos años tengamos qubit lógicos resistentes a errores arbitrarios.

Ahora mapearemos el código de Shor a un dispositivo cuántico para ver el qubit lógico. Consid-



eresmos el siguiende dispositivo cuántico de 53 qubits:

[10]:
```
from qiskit.transpiler.passes import RemoveBarriers

real_backend = FakeRochesterV2()
aer = AerSimulator()
coupling_map = list( real_backend.coupling_map )
basis_gates = [ 'h', 'u', 'cx', 'swap' ]

def count_gates( shor_code, layout ):
    qc_shor = shor_code()
    qc_shor = RemoveBarriers()(qc_shor)
    qc_transpiled = transpile( qc_shor, aer, basis_gates=basis_gates,
                               coupling_map=real_backend.coupling_map,
                               optimization_level=0 , seed_transpiler=0,
                               initial_layout=layout )
    return qc_transpiled.count_ops()
    # qc_transpiled.draw('mpl', fold=-1)
```

Si el circuito mapeado al dispositivo cuántico contiene puertas `cx` entre qubits físicamente no conectado, es necesario introducir puertas `swaps`. Estas permiten intercambiar el estado entre dos qubit, y requieren 3 puertas `cx` para ser implementadas. Veamos el número total de `cx` con el un mapeo trivíal, donde los qubit virtuales se mapean al los qubits físicos en la misma posición:

[11]:
```
layout = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]
count_gates( shor_code, layout )
```

[11]:
```
OrderedDict([('swap', 50), ('cx', 32), ('h', 7)])
```

Este número de `cx` no es óptimo y puede ser mejorado. Encontrar un mapeo óptimo es extremadamente dificil, pues el número de posibles mapeos de n_v qubits virtuales a n_f qubit físicos es gigantesco:

$$N_{var} = \frac{n_f!}{(n_f - n_v)!} = \frac{53!}{(53-17)!} = 1149182788022013258049536000.$$

Incluso si nos restringimos a n_v qubits físicos, el número sigue siendo grande

$$N_{var} = \frac{n_v!}{(n_v - n_v)!} = 17! = 355687428096000.$$

Por esto, la optimización de circuitos para dispositivos físicos es un problema muy relevante y ampliamente estudiado.

Desafio: Busque un mapeo de qubits virtuales, es decir los qubits de su circuito, a los qubits físicos del dispositivo cuántico anterior que utilize una cantidad de `swap` menor a 40. HINT: Intente que los qubits con más puertas `cx` esten juntos, y busque una distribución de qubit simétrica, sea horizontal, vertical o diagonal.

[12]:
```
layout = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16] ## Esta es solo un opción␣
↪de default

count_gates( shor_code, layout )
```

[12]:
```
OrderedDict([('swap', 50), ('cx', 32), ('h', 7)])
```

[13]:
```
test_9c( shor_code, layout )
```

```
Tu mapeo emplea muchas swaps
```

[ ]:
```

```