

Computational note

March 26, 2026

Computational note for Quantum Computation course (MAT3420), spring 2026

```
[1]: import sympy as sy
sy.init_printing(use_unicode=True)
```

1 Lecture 3

Exponential of transforms

```
[2]: θ = sy.Symbol('θ', real=True)
```

```
[3]: S = sy.Matrix([[0, sy.I * θ],
                  [-sy.I * θ, 0]])
S, sy.exp(sy.I * S)
```

```
[3]:  $\left(\begin{bmatrix} 0 & i\theta \\ -i\theta & 0 \end{bmatrix}, \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}\right)$ 
```

```
[4]: T = sy.Matrix([[0, θ],
                  [θ, 0]])
T, sy.exp(sy.I * T)
```

```
[4]:  $\left(\begin{bmatrix} 0 & \theta \\ \theta & 0 \end{bmatrix}, \begin{bmatrix} \frac{e^{i\theta} + e^{-i\theta}}{2} & \frac{e^{i\theta} - e^{-i\theta}}{2} \\ \frac{e^{i\theta} - e^{-i\theta}}{2} & \frac{e^{i\theta} + e^{-i\theta}}{2} \end{bmatrix}\right)$ 
```

```
[5]: sy.simplify(sy.exp(sy.I * T))
```

```
[5]:  $\begin{bmatrix} \cos(\theta) & i\sin(\theta) \\ i\sin(\theta) & \cos(\theta) \end{bmatrix}$ 
```

2 Lecture 4

Parametrize density operators on $\mathbb{C}^2$

```
[6]: # Pauli matrices
X = sy.Matrix([[0, 1],
```

```

[1, 0]])
Y = sy.Matrix([[0, -sy.I],
               [sy.I, 0]])
Z = sy.Matrix([[1, 0],
               [0, -1]])

```

```
[7]: x, y, z = sy.symbols('x y z', real=True)
```

```
[8]: # 'evenly mixed' state represented by scalar matrix
HlfI2 = sy.Matrix([[sy.Rational(1,2), 0],
                  [0, sy.Rational(1,2)]])
```

2×2 matrices which are selfadjoint and has trace 1 can be written as linear combination of I_2, X, Y, Z , with coefficient $\frac{1}{2}$ for I_2

```
[9]: T = HlfI2 + x * X + y * Y + z * Z
T
```

```
[9]: 
$$\begin{bmatrix} z + \frac{1}{2} & x - iy \\ x + iy & \frac{1}{2} - z \end{bmatrix}$$

```

given $T = T^\dagger$ and $\text{tr}(T) = 1$, we expect the positivity of T to be equivalent to the positivity of $T - T^2$

```
[10]: sy.simplify(T - T * T)
```

```
[10]: 
$$\begin{bmatrix} -x^2 - y^2 - z^2 + \frac{1}{4} & 0 \\ 0 & -x^2 - y^2 - z^2 + \frac{1}{4} \end{bmatrix}$$

```

T has determinant 0 when $x^2 + y^2 + z^2 = \frac{1}{4}$, meaning it has rank 1 on the sphere defined by this equation

```
[11]: T.det()
```

```
[11]: 
$$-x^2 - y^2 - z^2 + \frac{1}{4}$$

```

3 Exercise 1

```
[12]: from sympy.physics.quantum import TensorProduct
```

3.1 Exercise 1.1

Kronecker product and matrix product

```
[13]: A11, A12, A21, A22 = sy.symbols('A11 A12 A21 A22')
A = sy.Matrix([[A11, A12],
               [A21, A22]])
```

```
B11, B12, B21, B22 = sy.symbols('B11 B12 B21 B22')
B = sy.Matrix([[B11, B12],
               [B21, B22]])
A, B
```

```
[13]:  $\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$ 
```

```
[14]: TensorProduct(A, B)
```

```
[14]:  $\begin{bmatrix} A_{11}B_{11} & A_{11}B_{12} & A_{12}B_{11} & A_{12}B_{12} \\ A_{11}B_{21} & A_{11}B_{22} & A_{12}B_{21} & A_{12}B_{22} \\ A_{21}B_{11} & A_{21}B_{12} & A_{22}B_{11} & A_{22}B_{12} \\ A_{21}B_{21} & A_{21}B_{22} & A_{22}B_{21} & A_{22}B_{22} \end{bmatrix}$ 
```

```
[15]: v1, v2, w1, w2 = sy.symbols('v1 v2 w1 w2')
v = sy.Matrix([v1, v2])
w = sy.Matrix([w1, w2])
```

```
[16]: TensorProduct(v, w)
```

```
[16]:  $\begin{bmatrix} v_1w_1 \\ v_1w_2 \\ v_2w_1 \\ v_2w_2 \end{bmatrix}$ 
```

```
[17]: TensorProduct(A, B) * TensorProduct(v, w)
```

```
[17]:  $\begin{bmatrix} A_{11}B_{11}v_1w_1 + A_{11}B_{12}v_1w_2 + A_{12}B_{11}v_2w_1 + A_{12}B_{12}v_2w_2 \\ A_{11}B_{21}v_1w_1 + A_{11}B_{22}v_1w_2 + A_{12}B_{21}v_2w_1 + A_{12}B_{22}v_2w_2 \\ A_{21}B_{11}v_1w_1 + A_{21}B_{12}v_1w_2 + A_{22}B_{11}v_2w_1 + A_{22}B_{12}v_2w_2 \\ A_{21}B_{21}v_1w_1 + A_{21}B_{22}v_1w_2 + A_{22}B_{21}v_2w_1 + A_{22}B_{22}v_2w_2 \end{bmatrix}$ 
```

```
[18]: TensorProduct(A * v, B * w)
```

```
[18]:  $\begin{bmatrix} (A_{11}v_1 + A_{12}v_2)(B_{11}w_1 + B_{12}w_2) \\ (A_{11}v_1 + A_{12}v_2)(B_{21}w_1 + B_{22}w_2) \\ (A_{21}v_1 + A_{22}v_2)(B_{11}w_1 + B_{12}w_2) \\ (A_{21}v_1 + A_{22}v_2)(B_{21}w_1 + B_{22}w_2) \end{bmatrix}$ 
```

```
[19]: # naive comparison gives wrong answer
# because symbolic manipulation is not through enough
TensorProduct(A, B) * TensorProduct(v, w) == TensorProduct(A * v, B * w)
```

```
[19]: False
```

```
[20]: # unfortunately, wrong answer even with the simplify command
```

```
sy.simplify(TensorProduct(A, B) * TensorProduct(v, w)) == sy.  
↪simplify(TensorProduct(A * v, B * w))
```

[20]: False

```
[21]: # correct answer by taking the difference, then simplify  
sy.simplify(TensorProduct(A, B) * TensorProduct(v, w) - TensorProduct(A,   
↪* v, B * w))
```

[21]: $\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$

3.2 Exercise 1.2

tensor product of qubit states

```
[22]: zeroket = sy.Matrix([1, 0])  
oneket = sy.Matrix([0, 1])
```

```
[23]: # list the Kronecker product of computational basis vectors  
print("printing |00>, |01>, |10>, |11>")  
TensorProduct(zeroket,zeroket), TensorProduct(zeroket,oneket),   
↪TensorProduct(oneket,zeroket), TensorProduct(oneket,oneket)
```

printing |00>, |01>, |10>, |11>

[23]: $\begin{pmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \end{pmatrix}$

```
[24]: plusket = sy.Matrix([1/sy.sqrt(2), 1/sy.sqrt(2)])  
minusket = sy.Matrix([1/sy.sqrt(2), -1/sy.sqrt(2)])  
plusket, minusket
```

[24]: $\begin{pmatrix} \begin{bmatrix} \frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} \end{bmatrix}, \begin{bmatrix} \frac{\sqrt{2}}{2} \\ -\frac{\sqrt{2}}{2} \end{bmatrix} \end{pmatrix}$

since the Kronecker product of computational basis vectors (i.e., |00>, |01>, |10>, |11>) become standard basis vectors, when TensorProduct(plusket, plusket) gives 4 componets we can treat them as coefficients of |11> in terms of those vectors

```
[25]: TensorProduct(plusket,plusket)
```

[25]:

$$\begin{bmatrix} \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \end{bmatrix}$$

[26]: `TensorProduct(plusket,minusket)`

[26]:
$$\begin{bmatrix} \frac{1}{2} \\ \frac{1}{2} \\ -\frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \\ -\frac{1}{2} \end{bmatrix}$$

3.3 Exercise 1.3

[27]: `# Hadamard matrix`
`H = sy.Matrix([[1, 1],`
`[1, -1]]) * (1/sy.sqrt(2))`

[28]: `eyeket = sy.Matrix([1/sy.sqrt(2), sy.I/sy.sqrt(2)])`
`minuseyeket = sy.Matrix([1/sy.sqrt(2), -sy.I/sy.sqrt(2)])`
`eyeket, minuseyeket`

[28]:
$$\left(\begin{bmatrix} \frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}i}{2} \end{bmatrix}, \begin{bmatrix} \frac{\sqrt{2}}{2} \\ -\frac{\sqrt{2}i}{2} \end{bmatrix} \right)$$

the transformation of $|0\rangle$ by H gives $|+\rangle$

[29]: `H * zeroket`

[29]:
$$\begin{bmatrix} \frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} \end{bmatrix}$$

[30]: `H * oneket`

[30]:
$$\begin{bmatrix} \frac{\sqrt{2}}{2} \\ -\frac{\sqrt{2}}{2} \end{bmatrix}$$

[31]: `H * plusket`

[31]:
$$\begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

[32]: `H * minusket`

[32]:
$$\begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

```
[33]: H * eyeket
```

```
[33]: 
$$\begin{bmatrix} \frac{1}{2} + \frac{i}{2} \\ \frac{1}{2} - \frac{i}{2} \end{bmatrix}$$

```

```
[34]: # is it a scalar multiple of |i>? – Not quite
H * eyeket == eyeket * ((1 + sy.I)/sy.sqrt(2))
```

```
[34]: False
```

```
[35]: # is it a scalar multiple of |-i>? – True, but get false negative
H * eyeket == minuseyeket * ((1 + sy.I)/sy.sqrt(2))
```

```
[35]: False
```

```
[36]: # simplfying the difference gives the expected answer
sy.simplify(H * eyeket - minuseyeket * ((1 + sy.I)/sy.sqrt(2)))
```

```
[36]: 
$$\begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

```

3.4 Exercise 1.5

```
[37]: # list the eigenvalue, multiplicity, eigenvector(s)
H.eigenvecs()
```

```
[37]: 
$$\left[ \left( -1, 1, \begin{bmatrix} 1 - \sqrt{2} \\ 1 \end{bmatrix} \right), \left( 1, 1, \begin{bmatrix} 1 + \sqrt{2} \\ 1 \end{bmatrix} \right) \right]$$

```

```
[38]: # eigenvector for eigenvalue 1
v1 = sy.Matrix([1+sy.sqrt(2), 1])
```

```
[39]: # calculate the unit vector
v1norm = sy.sqrt((v1.T * v1)[0])
u1 = sy.simplify(v1 * (1/v1norm))
u1
```

```
[39]: 
$$\begin{bmatrix} \frac{\sqrt{\sqrt{2}+2}}{2\sqrt{\sqrt{2}+2}} \\ \frac{2}{\sqrt{2}} \end{bmatrix}$$

```

To get a vector halfway between $|0\rangle$ and $|+\rangle$, we should consider vector at angle $\frac{\pi}{8}$: $\cos(\frac{\pi}{8})|0\rangle + \sin(\frac{\pi}{8})|1\rangle$

```
[40]: u2 = sy.Matrix([sy.cos(sy.pi/8), sy.sin(sy.pi/8)])
u2
```

```
[40]:
```

$$\begin{bmatrix} \sqrt{\frac{\sqrt{2}}{4} + \frac{1}{2}} \\ \sqrt{\frac{1}{2} - \frac{\sqrt{2}}{4}} \end{bmatrix}$$

```
[41]: # compare them
sy.simplify(u1 - u2)
```

```
[41]:  $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$ 
```

4 Lecture 7

Elitzur-Vaidman bomb experiment

```
[42]: import numpy as np
rng = np.random.default_rng()
```

```
[43]: def col_vec(components):
    """Return a column vector (as a NumPy array) with the given
    components.

    In a matrix context, this should be treated as a column vector.
    """
    return np.array(components)

def col_vec_inn_prod(v1, v2):
    """Hilbert space inner product of column vectors."""
    return np.conjugate(v1).dot(v2)

zeroket = col_vec([1, 0])
oneket = col_vec([0, 1])
plusket = col_vec([1, 1]) * 1/np.sqrt(2)
minusket = col_vec([1, -1]) * 1/np.sqrt(2)
eyeket = col_vec([1, 1j]) * 1/np.sqrt(2)
zeroket, oneket, eyeket
```

```
[43]: (array([1, 0]),
array([0, 1]),
array([0.70710678+0.j, 0. +0.70710678j]))
```

```
[44]: col_vec_inn_prod(zeroket, eyeket), col_vec_inn_prod(oneket, eyeket),
col_vec_inn_prod(plusket, eyeket)
```

```
[44]: ((0.7071067811865475+0j),
       0.7071067811865475j,
       (0.4999999999999999+0.4999999999999999j))
```

measurement of qubit state $v = (z, w)$ in “computational basis” should give the case 0 with probability $|z|^2$, and the case 1 with probability $|w|^2$

```
[45]: def measure_qubit(v):
       """Measure a single qubit state vector.

       Input:
           v: qubit state, a length-2 NumPy array (complex amplitudes)

       Output (int):
           0 with probability  $|v_0|^2$ ,
           1 with probability  $|v_1|^2 = 1 - |v_0|^2$ .
       """
       # Compute the threshold value  $|v_0|^2 / ||v||^2$ 
       # Do not assume  $||v|| == 1$ 
       # hence threshold =  $|v_0|^2 / ||v||^2$ 
       norm_v_sq = np.real(np.dot(v, v))
       v1_sq = np.abs(v[1])**2
       threshold = v1_sq / norm_v_sq
       rand_num = rng.random()
       if rand_num < threshold:
           return 0
       else:
           return 1
```

```
[46]: # if we measure  $|0\rangle$ , expect the outcome 0 with probability 1
       [measure_qubit(zeroKet) for _ in range(1000)].count(0)
```

```
[46]: 1000
```

```
[47]: # if we measure  $|1\rangle$ , expect the outcome 1 with probability 1
       [measure_qubit(oneKet) for _ in range(1000)].count(0)
```

```
[47]: 0
```

```
[48]: # if we measure  $|+\rangle$ , expect the outcome 0 with probability 0.5
       [measure_qubit(plusKet) for _ in range(1000)].count(0)
```

```
[48]: 470
```

```
[49]: def bomb_machine(v):
       """Implements the Elitzur–Vaidman bomb test.
```

```

    Input:
        v: a qubit (column vector with 2 components)

    Output (qubit or str):
        - If measurement of v in the computational basis yields 0,
        returns  $|0\rangle$ .
        - If the measurement yields 1, returns "💣".
    """
    if measure_qubit(v) == 0:
        return zeroket
    else:
        return "💣"

def empty_suitcase(v):
    """An empty suitcase returns the input unchanged."""
    return v

```

```

[50]: # with  $|+\rangle$  as input, expect the explosion half of the time, otherwise  $|0\rangle$ 
[bomb_machine(plusket) for _ in range(10)]

```

```

[50]: [array([1, 0]),
array([1, 0]),
array([1, 0]),
array([1, 0]),
array([1, 0]),
'💣',
array([1, 0]),
'💣',
'💣',
array([1, 0])]

```

```

[51]: def EV_algo( $\epsilon$ , suitcase):
    """Determine whether the EV-suitcase contains a bomb.

    Input:
         $\epsilon$  (float): control parameter, assumed to be small
        suitcase: EV-bomb function or empty_suitcase function

    Output:
        str reporting if the bomb exploded, otherwise the guess about
        the content of the suitcase.
    """
    # work out the iteration count N and the rotation matrix

```

```

N = int(np.floor(np.pi / (2 * ε)))
R_ε = np.array([[np.cos(ε), -np.sin(ε)],
                 [np.sin(ε), np.cos(ε)]])
# initial vector is |0>
w_vec = zeroket
for _ in range(N):
    v = suitcase(w_vec)
    if isinstance(v, str) and (v == "💣"):
        return "we died"
    # otherwise continue with the loop
    w_vec = R_ε @ v
# if the suitcase did not explode, do measurement to decide our guess
    last_qubit_meas = measure_qubit(w_vec)
    if last_qubit_meas == 0:
        return "guess bomb"
    else:
        return "guess not bomb"

```

when the suitcase contains the EV-bomb, we expect explosion with probability $\sim \frac{\pi}{2}\epsilon$, and if we survive, get the correct guess with probability $\sim 1 - \epsilon^2$

```
[52]: ε = 0.1
      np.pi * ε / 2, 1 - ε**2
```

```
[52]: (0.15707963267949, 0.99)
```

with $\epsilon = 0.1$, we expect to die 15% of the times, and otherwise expect the correct guess 99% of the times

```
[53]: test_results = [EV_algo(0.1, bomb_machine) for _ in range(100)]
      test_results.count("guess bomb"), test_results.count("guess not bomb"),
      ↪ test_results.count("we died")
```

```
[53]: (84, 2, 14)
```

```
[54]: ε = 0.01
      np.pi * ε / 2, 1 - ε**2
```

```
[54]: (0.015707963267949, 0.9999)
```

with $\epsilon = 0.01$, we expect to die 1.5% of the times, and otherwise expect the correct guess 99.99% of the times

```
[55]: test_results = [EV_algo(0.01, bomb_machine) for _ in range(100)]
      test_results.count("guess bomb"), test_results.count("guess not bomb"),
      ↪ test_results.count("we died")
```

```
[55]: (99, 0, 1)
```

when the suitcase does not contain the EV-bomb, we survive for sure, and get the correct guess with probability (at least) $\sim 1 - \epsilon^2$

```
[56]: test_results = [EV_algo(0.1, empty_suitcase) for _ in range(100)]
      test_results.count("guess bomb"), test_results.count("guess not bomb"),
      ↪test_results.count("we died")
```

```
[56]: (1, 99, 0)
```

```
[57]: test_results = [EV_algo(0.01, empty_suitcase) for _ in range(100)]
      test_results.count("guess bomb"), test_results.count("guess not bomb"),
      ↪test_results.count("we died")
```

```
[57]: (0, 100, 0)
```

5 Lecture 8

Deutsch's algorithm

```
[58]: # we work with symbolic computation, so use sympy
      zeroket = sy.Matrix([1, 0])
      oneket = sy.Matrix([0, 1])
      plusket = sy.Matrix([1/sy.sqrt(2), 1/sy.sqrt(2)])
      minusket = sy.Matrix([1/sy.sqrt(2), -1/sy.sqrt(2)])
```

```
[59]: zerozeroket = TensorProduct(zeroket, zeroket)
      zerooneket = TensorProduct(zeroket, oneket)
      onezeroket = TensorProduct(oneket, zeroket)
      oneoneket = TensorProduct(oneket, oneket)
```

Helper functions for two-qubit computational basis; see Exercise 1.2 for the convention

```
[60]: def comp_basis_coeff(v, ij):
      """Return the coefficient of a two-qubit state in the computational_
      ↪basis.

      Input:
        v: SymPy vector with 4 components
        ij: pair of integers, one of (0, 0), (0, 1), (1, 0), (1, 1)

      Output:
        Coefficient of v for |ij>.
      """
      if ij == (0,0):
          return v[0]
      elif ij == (0,1):
```

```

        return v[1]
    elif ij == (1,0):
        return v[2]
    return v[3]

def comp_basis_for_index(ij):
    """Return the computational basis vector  $|ij\rangle$  for the given index.

    Input:
        ij: pair of integers, one of (0, 0), (0, 1), (1, 0), (1, 1)

    Output:
        The corresponding computational basis vector  $|ij\rangle$ .
    """
    if ij == (0,0):
        return zerozeroket
    elif ij == (0,1):
        return zerooneket
    elif ij == (1,0):
        return onezeroket
    return oneoneket

```

```

[61]: a, b, c, d = sy.symbols('a00 a01 a10 a11')
test_vec = a * zerozeroket + b * zerooneket + c * onezeroket + d * oneoneket
test_vec

```

```

[61]: 
$$\begin{bmatrix} a_{00} \\ a_{01} \\ a_{10} \\ a_{11} \end{bmatrix}$$


```

```

[62]: index_list = [(0,0), (0,1), (1,0), (1,1)]
[comp_basis_coeff(test_vec, index) for index in index_list]

```

```

[62]: [a00, a01, a10, a11]

```

```

[63]: # XOR operation is represented by the ^ operator
0 ^ 0, 0 ^ 1, 1 ^ 0, 1 ^ 1

```

```

[63]: (0, 1, 1, 0)

```

```

[64]: def U_f_gate(f, v):
    """Quantum gate for Deutsch's algorithm.

    Input:

```

f: boolean function; takes input 0 or 1, returns 0 or 1
v: SymPy vector with 4 components

Output:

The result of applying the linear transform

$|ij\rangle \rightarrow |ik\rangle$ with $k = \text{XOR}(f(i), j)$ to v .

```
"""
ij_list = [(0,0), (0,1), (1,0), (1,1)]
a, b, c, d = [comp_basis_coeff(v, index) for index in ij_list]
ik_list = [(p[0], f(p[0]) ^ p[1]) for p in ij_list]
va, vb, vc, vd = [comp_basis_for_index(ik_pair) for ik_pair in
ik_list]
return a * va + b * vb + c * vc + d * vd
```

with the constant function $f(i) = 0$, expect $U_f(v) = v$ from $\text{XOR}(0, j) = j$

```
[65]: def f0(i):
        return 0
    U_f_gate(f0, test_vec)
```

```
[65]: [a00]
        [a01]
        [a10]
        [a11]
```

with the constant function $f(i) = 1$, expect flip on the second bit from $\text{XOR}(1, j) = \text{NOT}(j)$

```
[66]: def f1(i):
        return 1
    U_f_gate(f1, test_vec)
```

```
[66]: [a01]
        [a00]
        [a11]
        [a10]
```

with the identity function $f(i) = i$, expect CNOT

```
[67]: def f_id(i):
        return i
    U_f_gate(f_id, test_vec)
```

```
[67]: [a00]
        [a01]
        [a11]
        [a10]
```

```
[68]: # initial vector for Deutsch's algorithm
plusminusket = TensorProduct(plusket, minusket)
plusminusket
```

$$[68]: \begin{bmatrix} 1 \\ \frac{1}{2} \\ -\frac{1}{2} \\ 1 \\ \frac{1}{2} \\ -\frac{1}{2} \end{bmatrix}$$

U_f gate with f_1 on the input vector $v = |+-\rangle$ gives the output vector $-v$

```
[69]: U_f_gate(f1, plusminusket)
```

$$[69]: \begin{bmatrix} 1 \\ -\frac{1}{2} \\ \frac{1}{2} \\ 1 \\ -\frac{1}{2} \\ \frac{1}{2} \end{bmatrix}$$

U_f gate with f_{id} on the input vector $|+-\rangle$ gives the output vector $|--\rangle$

```
[70]: U_f_gate(f_id, plusminusket), TensorProduct(minusket,minusket)
```

$$[70]: \left(\begin{bmatrix} 1 \\ \frac{1}{2} \\ -\frac{1}{2} \\ 1 \\ \frac{1}{2} \\ -\frac{1}{2} \end{bmatrix}, \begin{bmatrix} 1 \\ \frac{1}{2} \\ -\frac{1}{2} \\ 1 \\ \frac{1}{2} \\ -\frac{1}{2} \end{bmatrix} \right)$$

```
[71]: E_plus = TensorProduct(plusket * plusket.T, sy.eye(2))
E_minus = TensorProduct(minusket * minusket.T, sy.eye(2))
E_plus, E_minus
```

$$[71]: \left(\begin{bmatrix} \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} \end{bmatrix}, \begin{bmatrix} \frac{1}{2} & 0 & -\frac{1}{2} & 0 \\ 0 & \frac{1}{2} & 0 & -\frac{1}{2} \\ -\frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & -\frac{1}{2} & 0 & \frac{1}{2} \end{bmatrix} \right)$$

```
[72]: # apply U_f gate for the constant function f1 to the initial input
v_prime = U_f_gate(f1, plusminusket)
# then estimate the probability of observing the + case
p_plus = v_prime.T * E_plus * v_prime
p_plus
```

$$[72]: [1]$$

```
[73]: # apply U_f gate for the identity function f_id to the initial input
v_prime = U_f_gate(f_id, plusminusket)
# then estimate the probability of observing the + case
p_plus = v_prime.T * E_plus * v_prime
p_plus
```

```
[73]: [0]
```

6 Lecture 10

The BB84 key distribution protocol

```
[74]: import numpy as np
```

```
[75]: # from Lecture 7
rng = np.random.default_rng()
def col_vec(components):
    """Return a column vector (as a NumPy array) with the given_
    _components.

    In a matrix context, this should be treated as a column vector.
    """
    return np.array(components)

def col_vec_inn_prod(v1, v2):
    """Hilbert space inner product of column vectors."""
    return np.conjugate(v1).dot(v2)

zeroket = col_vec([1, 0])
oneket = col_vec([0, 1])
plusket = col_vec([1, 1]) * 1/np.sqrt(2)
minusket = col_vec([1, -1]) * 1/np.sqrt(2)
eyeket = col_vec([1, 1j]) * 1/np.sqrt(2)
```

6.1 preparing strings and states

control parameter is an integer n , that Alice and Bob share;

```
[76]: m = 5
n = 20
```

Alice chooses two strings $x = x_1 \cdots x_n$ and $y = y_1 \cdots y_n$ of 0's and 1's. This choice is random;

```
[77]: x = [rng.integers(2) for _ in range(n)]
y = [rng.integers(2) for _ in range(n)]
```

```
x, y
```

```
[77]: ([0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1],  
       [1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1])
```

she then prepares an n -qubits $v_1, \dots, v_n$ representing x according to the following rule: - if $y_j = 0$, then $v_j = |0\rangle$ or $|1\rangle$ depending on $x_j = 0$ or not; - if $y_j = 1$, then $v_j = |+\rangle$ or $|-\rangle$ depending on $x_j = 0$ or not;

```
[78]: def v_enc(x_val, y_val):  
    if y_val == 0:  
        if x_val == 0:  
            return zeroket  
        else:  
            return oneket  
    else:  
        if x_val == 0:  
            return plusket  
        return minusket
```

```
[79]: v = [v_enc(x[i], y[i]) for i in range(n)]  
v
```

```
[79]: [array([0.70710678, 0.70710678]),  
       array([1, 0]),  
       array([0, 1]),  
       array([0, 1]),  
       array([ 0.70710678, -0.70710678]),  
       array([0.70710678, 0.70710678]),  
       array([1, 0]),  
       array([0.70710678, 0.70710678]),  
       array([ 0.70710678, -0.70710678]),  
       array([0.70710678, 0.70710678]),  
       array([0.70710678, 0.70710678]),  
       array([ 0.70710678, -0.70710678]),  
       array([0.70710678, 0.70710678]),  
       array([ 0.70710678, -0.70710678]),  
       array([0, 1]),  
       array([0, 1]),  
       array([ 0.70710678, -0.70710678]),  
       array([1, 0]),  
       array([0, 1]),  
       array([ 0.70710678, -0.70710678])]
```

on Bob's side, he chooses a string $y' = y'_1 \cdots y'_n$ of 0's and 1's, again in a random way.

```
[80]: yp = [rng.integers(2) for _ in range(n)]
yp
```

```
[80]: [1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1]
```

6.2 transmit states and strings, without eavesdropping

Alice sends the state v to Bob;

Bob makes the measurement of the factors v_j by the following rule: - if $y'_j = 0$, the j -th qubit is measured in the computational basis, i.e., with the PVM $E_0 = |0\rangle\langle 0|$ and $E_1 = |1\rangle\langle 1|$; - if $y'_j = 1$, the j -th qubit is measured in the $(|+\rangle, |-\rangle)$ -basis, i.e., with $E_+ = |+\rangle\langle +|$ and $E_- = |-\rangle\langle -|$.

he records the result of measurement as the string $x' = x'_1 \cdots x'_n$ of 0's and 1's, where the case "+" corresponds to 0 and the case "-" corresponds to 1 in the case of $y'_j = 1$;

```
[81]: def measure_qubit(v, basis_choice):
    """Measure a qubit in a specified basis.

    Input:
        v: qubit state, a length-2 NumPy array (complex amplitudes)
        basis (int): 0 for computational basis, 1 for the |+⟩ / |−⟩ basis

    Output (int):
        0 or 1, according to the measurement of v in the specified basis.
        The case "+" corresponds to 0.
    """
    # to be on the safe side, do not assume ||v|| = 1
    # normalize the input vector
    norm_v = np.sqrt(np.real(col_vec_inn_prod(v, v)))
    v_normalized = v / norm_v
    if basis_choice == 0:
        # the case 0 happens with probability |v0|^2
        threshold = np.real(np.conjugate(v_normalized[0]) *
        ↪v_normalized[0])
        rand_num = rng.random()
        if rand_num < threshold:
            return 0
        else:
            return 1
    else:
        # the case "+" happens with probability |⟨+|v⟩|^2
        inn_prod_with_plus_ket = col_vec_inn_prod(plusket, v_normalized)
        threshold = np.abs(inn_prod_with_plus_ket)**2
        rand_num = rng.random()
        if rand_num < threshold:
```

```

        return 0
    else:
        return 1

```

```
[82]: xp = [measure_qubit(v[i], yp[i]) for i in range(n)]
xp
```

```
[82]: [0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1]
```

Alice and Bob exchange the strings y and y' ;

now they know the j 's satisfying $y_j = y'_j$, call them $j_1, \dots, j_k$; now Alice's string $x_{j_1} \dots x_{j_k}$ and Bob's string $x'_{j_1} \dots x'_{j_k}$ are candidates of their shared secret.

```
[83]: matching_indices = [i for i in range(n) if y[i] == yp[i]]
shared_secret_Alice = [x[i] for i in matching_indices]
shared_secret_Bob = [xp[i] for i in matching_indices]
matching_indices, shared_secret_Alice, shared_secret_Bob
```

```
[83]: ([0, 2, 6, 9, 10, 12, 13, 14, 15, 18, 19],
      [0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1],
      [0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1])
```

6.3 transmit states and strings, with eavesdropping

Alice sends the state v to Bob;

Eve looks into the state, but needs to make guess about measurement schemes

```
[84]: y_eve = [rng.integers(2) for _ in range(n)]
```

when she performs the measurement, the state changes according to the outcome

```
[85]: x_eve = [measure_qubit(v[i], y_eve[i]) for i in range(n)]
v = [v_enc(x_eve[i], y_eve[i]) for i in range(n)]
v
```

```
[85]: [array([0.70710678, 0.70710678]),
      array([1, 0]),
      array([ 0.70710678, -0.70710678]),
      array([0, 1]),
      array([0, 1]),
      array([0.70710678, 0.70710678]),
      array([1, 0]),
      array([0.70710678, 0.70710678]),
      array([0, 1]),
      array([0.70710678, 0.70710678]),
      array([1, 0]),
```

```

array([0, 1]),
array([0.70710678, 0.70710678]),
array([ 0.70710678, -0.70710678]),
array([ 0.70710678, -0.70710678]),
array([0, 1]),
array([ 0.70710678, -0.70710678]),
array([1, 0]),
array([0.70710678, 0.70710678]),
array([0, 1])]

```

Bob makes the measurement of the factors v_j by the following rule: - if $y'_j = 0$, the j -th qubit is measured in the computational basis, i.e., with the PVM $E_0 = |0\rangle\langle 0|$ and $E_1 = |1\rangle\langle 1|$; - if $y'_j = 1$, the j -th qubit is measured in the $(|+\rangle, |-\rangle)$ -basis, i.e., with $E_+ = |+\rangle\langle +|$ and $E_- = |-\rangle\langle -|$.

he records the result of measurement as the string $x' = x'_1 \cdots x'_n$ of 0's and 1's, where the case “+” corresponds to 0 and the case “-” corresponds to 1 in the case of $y'_j = 1$;

```
[86]: xp = [measure_qubit(v[i], yp[i]) for i in range(n)]
xp
```

```
[86]: [0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1]
```

Alice and Bob exchange the strings y and y' ;

now they know the j 's satisfying $y_j = y'_j$, call them $j_1, \dots, j_k$; now Alice's string $x_{j_1} \cdots x_{j_k}$ and Bob's string $x'_{j_1} \cdots x'_{j_k}$ are candidates of their shared secret.

```
[87]: matching_indices = [i for i in range(n) if y[i] == yp[i]]
shared_secret_Alice = [x[i] for i in matching_indices]
shared_secret_Bob = [xp[i] for i in matching_indices]
matching_indices, shared_secret_Alice, shared_secret_Bob
```

```
[87]: ([0, 2, 6, 9, 10, 12, 13, 14, 15, 18, 19],
      [0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1],
      [0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1])
```

6.4 verify if there was eavesdropping or not

Alice randomly chooses some $j_{a_1}, \dots, j_{a_m}$ among $j_1, \dots, j_k$; she sends the pairs (j_{a_s}, z_{a_s}) to Bob;

```
[88]: sample_indices = rng.choice(matching_indices, m, replace=False)
sample_secret_Alice = [x[i] for i in sample_indices]
sample_indices, sample_secret_Alice
```

```
[88]: (array([13, 12, 10, 0, 15]), [1, 0, 0, 0, 1])
```

Bob checks these z_{a_s} against the corresponding terms in his z' ; if he sees any discrepancy, he reports it to Alice. (Then they know that the shared secret is bad and should be discarded.)

```
[89]: sample_secret_Bob = [xp[i] for i in sample_indices]
      sample_secret_Bob, sample_secret_Alice == sample_secret_Bob
```

```
[89]: ([1, 0, 0, 0, 1], True)
```

otherwise they can keep the remaining terms of z and z' as valid shared secret which is not known to Eve.

```
[90]: remaining_secret_Alice = [x[i] for i in matching_indices if i not in
      ↪ sample_indices]
      remaining_secret_Bob = [xp[i] for i in matching_indices if i not in
      ↪ sample_indices]
      remaining_secret_Alice, remaining_secret_Bob
```

```
[90]: ([1, 0, 0, 1, 1, 1], [0, 0, 0, 0, 0, 1])
```

7 Lecture 11

7.1 superdense coding

```
[91]: import sympy as sy
```

```
[92]: from sympy.physics.quantum import TensorProduct
      zeroket = sy.Matrix([1, 0])
      oneket = sy.Matrix([0, 1])
      EPR_state = (TensorProduct(zeroket, zeroket) + TensorProduct(oneket,
      ↪ oneket)) * (1/sy.sqrt(2))
```

```
[93]: # keep track of the state
      w = EPR_state
```

message is a string of two bits

```
[94]: s = [0, 1]
```

if $s_1 = 1$, Alice applies the Pauli X transform on her part of w (the H_A factor)

```
[95]: X = sy.Matrix([[0, 1],
      ↪ [1, 0]])
      # indexing starts with 0
      if s[0] == 1:
          w = TensorProduct(X, sy.eye(2)) * w
```

if $s_2 = 1$, she then applies the Pauli Z transform on her part of the state

```
[96]: Z = sy.Matrix([[1, 0],
                    [0, -1]])
# indexing starts with 0
if s[1] == 1:
    w = TensorProduct(Z, sy.eye(2)) * w
```

then she sends her part of the state to Bob; this means Bob now has access to the overall state in $H_A \otimes H_B$

Bob does measurement for the projection-valued measure corresponding to the basis

$$w_{00} = w_{\text{EPR}}, \quad w_{10} = (X \otimes I_2)w_{\text{EPR}}, \quad w_{01} = (Z \otimes I_2)w_{\text{EPR}}, \quad w_{11} = (ZX \otimes I_2)w_{\text{EPR}}$$

```
[97]: w00 = EPR_state
w10 = TensorProduct(X, sy.eye(2)) * w00
w01 = TensorProduct(Z, sy.eye(2)) * w00
w11 = TensorProduct(Z * X, sy.eye(2)) * w00
```

the (index of) result of the measurement is guaranteed to be $s_1 s_2$, what Alice wanted to send

```
[98]: def inn_prod_abs_sq(w1, w2):
    """Return the squared absolute value of the inner product <w1, w2>.

    Input:
        w1, w2: SymPy column matrices

    Output:
        The scalar |<w1, w2>|^2 as a SymPy expression.
    """
    inn_prod = sy.conjugate(w1).T * w2
    abs_sq = sy.conjugate(inn_prod) * inn_prod
    return abs_sq[0]
```

```
[99]: # measurement probabilities are square of absolute value of inner product
inn_prod_abs_sq(w00, w), inn_prod_abs_sq(w10, w), inn_prod_abs_sq(w01, w),
inn_prod_abs_sq(w11, w)
```

```
[99]: (0, 0, 1, 0)
```

check consistency of the above scheme; we want Bob's basis to be orthogonal

```
[100]: inn_prod_abs_sq(w00, w10), inn_prod_abs_sq(w00, w01), \
inn_prod_abs_sq(w00, w11), inn_prod_abs_sq(w10, w01), \
inn_prod_abs_sq(w10, w11), inn_prod_abs_sq(w01, w11)
```

```
[100]: (0, 0, 0, 0, 0, 0)
```

7.2 quantum teleportation

Alice wants to send a qubit state $v \in H_{A1} = \mathbb{C}^2$ to Bob

```
[101]:  $\alpha, \beta = \text{sy.symbols}(\text{"}\alpha \beta\text{"})$   
 $v = \text{sy.Matrix}([\alpha, \beta])$ 
```

we take $v \otimes w_{\text{EPR}}$ in the composite system $H_{A1} \otimes H_{A2} \otimes H_B$;

```
[102]: tot_state = TensorProduct(v, EPR_state)
```

on her part of the system $H_{A1} \otimes H_{A2}$, Alice first performs the CNOT transform;

```
[103]: CNOT = sy.Matrix([[1, 0, 0, 0],  
                        [0, 1, 0, 0],  
                        [0, 0, 0, 1],  
                        [0, 0, 1, 0]])  
tot_state = TensorProduct(CNOT, sy.eye(2)) * tot_state
```

then she performs the Hadamard transform on the H_{A1} part;

```
[104]: H = sy.Matrix([[1, 1],  
                    [1, -1]]) * (1/sy.sqrt(2))  
tot_state = TensorProduct(H, sy.eye(4)) * tot_state
```

Alice measures her part $H_{A1} \otimes H_{A2}$ with PVM for the computational basis

```
[105]: zerozeroket = TensorProduct(zeroket, zeroket)  
onezeroket = TensorProduct(oneket, zeroket)  
zerooneket = TensorProduct(zeroket, oneket)  
oneoneket = TensorProduct(oneket, oneket)  
E00 = zerozeroket.reshape(4,1) * zerozeroket.reshape(1,4)  
E01 = zerooneket.reshape(4,1) * zerooneket.reshape(1,4)  
E10 = onezeroket.reshape(4,1) * onezeroket.reshape(1,4)  
E11 = oneoneket.reshape(4,1) * oneoneket.reshape(1,4)
```

```
[106]: def E(s):  
    """Convenience function to get E_ij for s = ij."""  
    if s == [0, 0]:  
        return E00  
    elif s == [0, 1]:  
        return E01  
    elif s == [1, 0]:  
        return E10  
    return E11
```

she sends the outcome $s_1 s_2$ to Bob;

```
[107]: # choose s
s = [1, 0]
# the factor of 2 is to get normalized vector
new_state = TensorProduct(E(s), sy.eye(2)) * tot_state * 2
```

Bob first looks at s_1 , and if it is 1, then he applies the Pauli Z transform to his part of the state

```
[108]: if s[0] == 1:
    new_state = TensorProduct(sy.eye(4), Z) * new_state
```

he next looks at s_2 , and if it is 1, then he applies the Pauli X transform;

```
[109]: if s[1] == 1:
    new_state = TensorProduct(sy.eye(4), X) * new_state
```

now Bob's state is guaranteed to be v .

```
[110]: new_state
```

```
[110]: 
$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \alpha \\ \beta \\ 0 \\ 0 \end{bmatrix}$$

```

8 Exercise 2

8.1 Exercise 2.1

```
[111]: import sympy as sy
from sympy.physics.quantum import TensorProduct
```

```
[112]: zeroket = sy.Matrix([1, 0])
oneket = sy.Matrix([0, 1])
zerozeroket = TensorProduct(zeroket, zeroket)
zerooneket = TensorProduct(zeroket, oneket)
onezeroket = TensorProduct(oneket, zeroket)
oneoneket = TensorProduct(oneket, oneket)
zerozeroket, zerooneket, onezeroket, oneoneket
```

```
[112]: 
$$\begin{pmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \end{pmatrix}$$

```

CNOT gate - if the first bit is 0, do nothing - if it is 1, flip the second bit

```
[113]: CNOT = sy.Matrix([[1, 0, 0, 0],
                        [0, 1, 0, 0],
                        [0, 0, 0, 1],
                        [0, 0, 1, 0]])
CNOT * zerooneket, CNOT * onezeroket
```

```
[113]:  $\begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$ 
```

reversed CNOT gate - if the second bit is 0, do nothing - if it is 1, flip the first bit

```
[114]: revCNOT = zerozeroket * zerozeroket.reshape(1,4)
revCNOT += onezeroket * onezeroket.reshape(1,4)
revCNOT += zerooneket * oneoneket.reshape(1,4)
revCNOT += oneoneket * zerooneket.reshape(1,4)
revCNOT
```

```
[114]:  $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$ 
```

matrix for the CNOT—reversed CNOT—CNOT circuit

```
[115]: compos = CNOT * revCNOT * CNOT
compos
```

```
[115]:  $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ 
```

we see that $|ij\rangle$ is transformed to $|ji\rangle$

```
[116]: compos * zerozeroket, compos * zerooneket, compos * onezeroket, compos * oneoneket
```

```
[116]:  $\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$ 
```

```
[117]: v1, v2, w1, w2 = sy.symbols("v1 v2 w1 w2")
v = sy.Matrix([v1, v2])
w = sy.Matrix([w1, w2])
vTw = TensorProduct(v, w)
```

```
vTw
```

```
[117]: 
$$\begin{bmatrix} v_1 w_1 \\ v_1 w_2 \\ v_2 w_1 \\ v_2 w_2 \end{bmatrix}$$

```

the circuit transforms $v \otimes w$ to $w \otimes v$

```
[118]: compos*vTw
```

```
[118]: 
$$\begin{bmatrix} v_1 w_1 \\ v_2 w_1 \\ v_1 w_2 \\ v_2 w_2 \end{bmatrix}$$

```

```
[119]: # record this matrix as "flip"
flip = compos
```

8.2 Exercise 2.2

```
[120]: # Hadamard matrix
H = sy.Matrix([[1, 1],
               [1, -1]]) * (1/sy.sqrt(2))
```

three-qubit CNOT gate with - first (qu)bit as control - second qubit is passed through - third (qu)bit is target

This can be constructed as $(\text{flip} \otimes I_2)(I_2 \otimes \text{CNOT})(\text{flip} \otimes I_2)$

```
[121]: I2 = sy.eye(2)
three_qubit_CNOT = TensorProduct(flip, I2) * TensorProduct(I2, CNOT) * \
    TensorProduct(flip, I2)
```

```
[122]: circ = TensorProduct(I2, three_qubit_CNOT) * \
    TensorProduct(three_qubit_CNOT, I2) * \
    TensorProduct(sy.eye(4), CNOT * TensorProduct(H, I2))
```

```
[123]: EPR_state = (TensorProduct(zero ket, zero ket) + TensorProduct(one ket, \
    one ket)) * (1/sy.sqrt(2))
```

```
[124]: def eq_check(i, j):
    qubit1 = zero ket if i == 0 else one ket
    qubit2 = zero ket if j == 0 else one ket
    input_state = TensorProduct(qubit1, TensorProduct(qubit2, \
    zero zero ket))
    circ_effect = circ * input_state
    EPR_at_34 = TensorProduct(sy.eye(4), EPR_state)
```

```
test_34_with_EPR = EPR_at_34.T * circ_effect
return test_34_with_EPR
```

```
[125]: eq_check(1,0)
```

```
[125]: 
$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

```

8.3 Exercise 2.3

```
[126]: # Pauli matrices
X = sy.Matrix([[0, 1],
               [1, 0]])
Y = sy.Matrix([[0, -sy.I],
               [sy.I, 0]])
Z = sy.Matrix([[1, 0],
               [0, -1]])
```

```
[127]: w10 = TensorProduct(X, I2) * EPR_state
w10
```

```
[127]: 
$$\begin{bmatrix} 0 \\ \frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} \\ 0 \end{bmatrix}$$

```

9 Lecture 12

The CHSH game: two cooperating players Alice and Bob plays the following game - Charlie (referee) gives them input bits x and y separately - they need to independently give back answer bits a and b - they win if $a + b = xy \bmod 2$ holds, lose otherwise

Alice and Bob prepares the EPR pair state before the game, and share the ($H_A = \mathbb{C}^2 = H_B$):

$$w = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \in H_A \otimes H_B$$

For Alice's part, if $x = 0$, she will measure her part in the computational basis to decide a . Otherwise, she will measure it in the $(|+\rangle, |-\rangle)$ -basis, and reply with $a = 0$ in the "+" case, $a = 1$ in the "-" case.

For Bob's part, he is going to work with the orthonormal basis

$$v = \cos \frac{\pi}{8} |0\rangle + \sin \frac{\pi}{8} |1\rangle, \quad v' = \cos \frac{5\pi}{8} |0\rangle + \sin \frac{5\pi}{8} |1\rangle$$

if $y = 0$, and otherwise with the orthonormal basis

$$w = \cos \frac{-\pi}{8} |0\rangle + \sin \frac{-\pi}{8} |1\rangle, \quad w' = \cos \frac{3\pi}{8} |0\rangle + \sin \frac{3\pi}{8} |1\rangle.$$

```
[128]: import numpy as np
rng = np.random.default_rng()
```

```
[129]: zeroket = np.array([1, 0])
oneket = np.array([0, 1])
plusket = np.array([1, 1]) / np.sqrt(2)
minusket = np.array([1, -1]) / np.sqrt(2)
EPR_stt = (np.kron(zeroket, zeroket) + np.kron(oneket, oneket)) / np.
    sqrt(2)
EPR_stt
```

```
[129]: array([0.70710678, 0.          , 0.          , 0.70710678])
```

```
[130]: bob_v = np.array([np.cos(np.pi / 8), np.sin(np.pi / 8)])
bob_vp = np.array([np.cos(5 * np.pi / 8), np.sin(5 * np.pi / 8)])
bob_w = np.array([np.cos(-np.pi / 8), np.sin(-np.pi / 8)])
bob_wp = np.array([np.cos(3 * np.pi / 8), np.sin(3 * np.pi / 8)])

def col_vec_inn_prod(v1, v2):
    """Hilbert space inner product of column vectors."""
    return np.conjugate(v1).dot(v2)

def measure_part_in_twoqubit(v, loc, basis_choice):
    """Measure one qubit of a two-qubit state in a specified basis.

    Input:
        v: two-qubit state, a length-4 NumPy array (complex components)
        loc (str): "A" for left qubit or "B" for right qubit
        basis_choice (int):
            0 → computational basis
            1 → |+⟩ / |−⟩ basis
            2 → Bob's first basis
            3 → Bob's second basis

    Output (o1: int, o2: np.array):
        o1: 0 or 1, according to the measurement of v in the specified_
        basis
        o2: updated (post-measurement) state vector
    """
    # to be on the safe side, do not assume ||v|| = 1
    # normalize the input vector
```

```

norm_v = np.sqrt(np.real(col_vec_inn_prod(v, v)))
v_normalized = v / norm_v
# decide basis
match basis_choice:
    case 0:
        basis_v0 = zeroket
        basis_v1 = oneket
    case 1:
        basis_v0 = plusket
        basis_v1 = minusket
    case 2:
        basis_v0 = bob_v
        basis_v1 = bob_vp
    case _:
        basis_v0 = bob_w
        basis_v1 = bob_wp
# decide PVM
sE0 = np.atleast_2d(basis_v0).T @ np.atleast_2d(basis_v0)
sE1 = np.atleast_2d(basis_v1).T @ np.atleast_2d(basis_v1)
if loc == "A":
    E0 = np.kron(sE0, np.eye(2))
    E1 = np.kron(sE1, np.eye(2))
else:
    E0 = np.kron(np.eye(2), sE0)
    E1 = np.kron(np.eye(2), sE1)
# the case 0 happens with probability <math>\langle v, E0(v) \rangle</math>
threshold = col_vec_inn_prod(v, E0 @ v)
rand_num = rng.random()
if rand_num < threshold:
    E0v = E0 @ v
    norm_E0v = np.sqrt(np.real(col_vec_inn_prod(E0v, E0v)))
    new_state = E0v / norm_E0v
    return (0, new_state)
else:
    E1v = E1 @ v
    norm_E1v = np.sqrt(np.real(col_vec_inn_prod(E1v, E1v)))
    new_state = E1v / norm_E1v
    return (1, new_state)

```

```
[131]: measure_part_in_twoqubit(EPR_stt, "A", 1)
```

```
[131]: (0, array([0.5, 0.5, 0.5, 0.5]))
```

```
[132]: measure_part_in_twoqubit(EPR_stt, "B", 2)
```

```
[132]: (0, array([0.85355339, 0.35355339, 0.35355339, 0.14644661]))
```

```
[133]: def play_CHSH():
        init_vec = EPR_stt
        # Charlie's question bits
        x, y = rng.integers(2), rng.integers(2)
        # Alice computes her answer first
        a, new_state = measure_part_in_twoqubit(init_vec, "A", x)
        # Bob computes his answer next
        b, _ = measure_part_in_twoqubit(new_state, "B", y + 2)
        # they win if a + b = x y mod 2
        if (a + b) % 2 == (x * y) % 2:
            return "win"
        else:
            return "lose"
```

```
[134]: [play_CHSH() for _ in range(100)].count("win")
```

```
[134]: 87
```

10 Lecture 13

```
[135]: import sympy as sy
        from sympy.physics.quantum import TensorProduct
        from functools import reduce
        import itertools as it
```

```
[136]: zeroket = sy.Matrix([1, 0])
        oneket = sy.Matrix([0, 1])
```

```
[137]: def comp_basis(x):
        """Return the computational basis vector |x>.

        Input:
            x: iterable of ints (0 or 1)

        Output:
            sympy.Matrix, 1-dimensional array representing |x>.
        """
        vec_seq = (zeroket if xi == 0 else oneket for xi in x)
        return reduce(TensorProduct, vec_seq)
```

```
[138]: # |01> and |10>
        comp_basis([0, 1]), comp_basis([1, 0])
```

```
[138]:
```

$$\begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

```
[139]: def dot_pr(x, y):
        """Return the dot product of (integer) sequences."""
        return sum(xi * yi for xi, yi in it.zip_longest(x, y))
```

```
[140]: dot_pr([0, 1], [1, 1])
```

```
[140]: 1
```

```
[141]: def v_y(y):
        """State vector from Proposition 13.1.

        Input:
            y: iterable of ints (0 or 1)

        Output:
            1D sympy.Matrix representing the normalized sum of
             $(-1)^{(x \cdot y)} |x\rangle$  over all bit strings  $x$  of the same length as  $y$ .
        """
        n = len(y)
        ind_list = it.product(range(2), repeat=n)
        res = sy.Matrix([0] * 2**n)
        for x in ind_list:
            res += comp_basis(x) * (1/sy.sqrt(2**n)) * (-1)**dot_pr(x, y)
        return res
```

```
[142]: v_y([0,0]), v_y([1,1])
```

```
[142]:
```

$$\begin{pmatrix} \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \end{pmatrix}, \begin{pmatrix} \frac{1}{2} \\ \frac{1}{2} \\ -\frac{1}{2} \\ -\frac{1}{2} \end{pmatrix}$$

```
[143]: def tens_pow_H(k):
        """Return the tensor product of copies of the Hadamard matrix."""
        H = sy.Matrix([[1, 1],
                        [1, -1]]) * (1/sy.sqrt(2))
        return reduce(TensorProduct, (H,) * k)
```

```
[144]: tens_pow_H(2)
```

```
[144]:
```

$$\begin{bmatrix} \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} & -\frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

10.1 Deutsch-Josza problem

```
[145]: def U(k, f):
        """Return the U_f gate for the Deutsch-Josza problem."""
        ind_list = it.product(range(2), repeat=k+1)
        res = sy.zeros(2**(k+1))
        for in_x in ind_list:
            out_x = in_x[:-1] + (f(in_x[:-1]) ^ in_x[-1],)
            res += comp_basis(out_x) * comp_basis(in_x).T
        return res
```

```
[146]: U(1, lambda x: 1)
```

```
[146]: 
$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

```

the vector v_y for $y = 1 \dots 1$

```
[147]: v_y([1] * 3)
```

```
[147]: 
$$\begin{bmatrix} \frac{\sqrt{2}}{4} \\ -\frac{\sqrt{2}}{4} \\ -\frac{\sqrt{2}}{4} \\ \frac{\sqrt{2}}{4} \\ -\frac{\sqrt{2}}{4} \\ \frac{\sqrt{2}}{4} \\ -\frac{\sqrt{2}}{4} \\ \frac{\sqrt{2}}{4} \\ -\frac{\sqrt{2}}{4} \end{bmatrix}$$

```

the constant case gives $\pm v_y$ for $y = 1 \dots 1$

```
[148]: U(2, lambda x: 1) * v_y([1] * 3)
```

```
[148]:
```

$$\begin{bmatrix} -\frac{\sqrt{2}}{4} \\ \frac{\sqrt{2}}{4} \\ -\frac{\sqrt{2}}{4} \\ \frac{\sqrt{2}}{4} \\ -\frac{\sqrt{2}}{4} \\ \frac{\sqrt{2}}{4} \\ -\frac{\sqrt{2}}{4} \\ \frac{\sqrt{2}}{4} \end{bmatrix}$$

the balanced case gives a vector orthogonal to the above

```
[149]: U(2, lambda x: x[0]) * v_y([1] * 3)
```

$$\begin{bmatrix} \frac{\sqrt{2}}{4} \\ -\frac{\sqrt{2}}{4} \\ \frac{\sqrt{2}}{4} \\ -\frac{\sqrt{2}}{4} \\ \frac{\sqrt{2}}{4} \\ -\frac{\sqrt{2}}{4} \\ \frac{\sqrt{2}}{4} \\ -\frac{\sqrt{2}}{4} \end{bmatrix}$$

```
[150]: # computational basis presentation
Hpow = tens_pow_H(3)
Hpow * U(2, lambda x: 0) * Hpow * comp_basis([1] * 3), \
Hpow * U(2, lambda x: 1) * Hpow * comp_basis([1] * 3), \
Hpow * U(2, lambda x: x[0]) * Hpow * comp_basis([1] * 3), \
Hpow * U(2, lambda x: (x[0] + x[1]) % 2) * Hpow * comp_basis([1] * 3)
```

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ -1 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

10.2 Bernstein-Vazirani problem

```
[151]: def V(k,f):
        """Return V_f gate for the Bernstein-Vazirani problem."""
        ind_list = it.product(range(2), repeat=k)
        res = sy.zeros(2**k)
        for x in ind_list:
            res += comp_basis(x) * comp_basis(x).T * (-1)**f(x)
        return res
```

```
[152]: V(2,lambda x: dot_pr(x, [0,1])), V(2,lambda x: dot_pr(x, [1,1]))
```

```
[152]:  $\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$ 
```

```
[153]: # check Proposition 13.3
s = [0,1]
V(2,lambda x: dot_pr(x, s)) * v_y([0,0]), v_y(s)
```

```
[153]:  $\begin{pmatrix} \frac{1}{2} \\ \frac{1}{2} \\ -\frac{1}{2} \\ \frac{1}{2} \end{pmatrix}, \begin{pmatrix} \frac{1}{2} \\ \frac{1}{2} \\ -\frac{1}{2} \\ \frac{1}{2} \end{pmatrix}$ 
```

```
[154]: s = [0,1]
tens_pow_H(2) * V(2,lambda x: dot_pr(x, s)) * v_y([0,0]), comp_basis(s)
```

```
[154]:  $\begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$ 
```

11 Lecture 14

Simon's problem: we work with Boolean functions $f: \{0,1\}^k \rightarrow \{0,1\}^k$ and a secret string $s \in \{0,1\}^k$ that contains 1. We know that $f(x) = f(y)$ if and only if either $x = y$ or y is the bitwise-XOR, and want to work out s from f .

```
[155]: import numpy as np
        rng = np.random.default_rng()
        import itertools as it
        from functools import reduce
```

```
[156]: # set secret string
k = 4 # 5 is the sensible maximum of naive implementation?
s = tuple([rng.integers(2) for _ in range(k)])
if 1 not in s:
    # try again
    s = tuple([rng.integers(2) for _ in range(k)])
    if 1 not in s:
        print("we could not generate a secret string.")
```

```
[157]: # function satisfying the above condition
def f_from_s(x):
    """Return x or y = x XOR s, chosen by a specific rule.

    Input:
        x: tuple of integers (0 or 1)
        s: tuple/list of integers (0 or 1) of length k
        k: int, length of x and s

    Rule:
        Compute y as the bitwise XOR of x with s.
        Find the first index i such that s[i] == 1.
        Return whichever of x or y has 0 at that index.

    Output:
        A tuple equal to either x or x XOR s.
    """
    ind_s_1 = [i for i in range(k) if s[i] == 1][0]

    if x[ind_s_1] == 0:
        return x
    y_l = [x[i] ^ s[i] for i in range(k)]
    return tuple(y_l)
```

```
[158]: s, f_from_s(s), f_from_s((0,) * k), f_from_s((1,) * k)
```

```
[158]: ((1, 1, 1, 1), (0, 0, 0, 0), (0, 0, 0, 0), (0, 0, 0, 0))
```

11.1 classical algorithm

The classical “birthday attack” algorithm works by picking input strings $x^1, x^2, \dots$ and querying $f(x^1), f(x^2), \dots$, recording the results. It stops when it sees a pair x^i, x^j such that $f(x^i) = f(x^j)$. Then the secret is $s = x^i \oplus x^j$, the bitwise XOR.

```
[159]: res_rec = {} # dictionary of f(x) as key and x as value
ind_list = it.product(range(2), repeat=k)
```

```

for x in ind_list:
    f_x = f_from_s(x)
    if f_x in res_rec.keys():
        prev_x = res_rec[f_x]
        found_secret = [x[i] ^ prev_x[i] for i in range(k)]
        print(f"found secret {found_secret} after {1 + len(res_rec)}_
↳queries")
        break
    # if f(x) is not found, record the results and continue with next x
    res_rec[f_x] = x

```

found secret [1, 1, 1, 1] after 9 queries

the expected run time is $\sqrt{2^k}$, where k is the length of secret

```
[160]: np.sqrt(2**k)
```

```
[160]: 4.0
```

11.2 Simon's quantum algorithm

the quantum algorithm looks at the state

$$v = (H^{\otimes k} \otimes I_{2^k}) U_f (H^{\otimes k} \otimes I_{2^k}) |0 \cdots 0\rangle$$

in the $2k$ -qubit space $\mathbb{C}^{2^{2k}} = \mathbb{C}^{2^k} \otimes \mathbb{C}^{2^k}$.

```
[161]: zeroket = np.array([1, 0])
       oneket = np.array([0, 1])
```

```
[162]: def comp_basis(x):
       # computational basis vector
       # input
       # x: array of int (0 or 1)
       # output
       # numpy.array, 1-dimensional array representing |x>
       vec_seq = (zeroket if xi == 0 else oneket for xi in x)
       return reduce(np.kron, vec_seq)
```

```
[163]: def tens_pow_H(k):
       # tensor product of copies of the Hadamard matrix
       H = np.array([[1, 1],
                     [1, -1]]) * (1/np.sqrt(2))
       return reduce(np.kron, [H] * k)
```

```
[164]: def termwise_XOR(x, y):
       # termwise XOR of tuples
       return tuple([xi ^ yi for xi, yi in it.zip_longest(x, y)])
```

```
def U(k, f):
    # U_f gate for the Simon algorithm
    ind_list = it.product(range(2), repeat=2*k)
    res = np.zeros((2**(2*k), 2**(2*k)), dtype=np.int8)
    for in_x in ind_list:
        out_x = in_x[0:k] + termwise_XOR(f(in_x[0:k]), in_x[k:2*k])
        res += np.atleast_2d(comp_basis(out_x)).T @ np.
        atleast_2d(comp_basis(in_x))
    return res
```

```
[165]: U(k, f_from_s).shape
```

```
[165]: (256, 256)
```

```
[166]: # first 20 x 20 portion of U_f
print(U(k, f_from_s)[0:20, 0:20])
```

```
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
```

```
[167]: # bottom right 20x20 portion
print(U(k, f_from_s)[-20:,-20:])
```

```
[0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

```

[0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]

```

```

[168]: H_pow_ampl = np.kron(tens_pow_H(k), np.identity(2**k))
       simon_gate = H_pow_ampl @ U(k,f_from_s) @ H_pow_ampl

```

```

[169]: v = simon_gate @ comp_basis([0] * (2 * k))

```

```

[170]: def proj(x):
       """Return the projection for the computational basis."""
       basis_vec = np.atleast_2d(comp_basis(x))
       return basis_vec.T @ basis_vec

       def meas_prob(x):
           """Return the probability of observing x in the first half of v."""
           return np.dot(v, np.kron(proj(x), np.identity(2**k)) @ v)

```

```

[171]: intervals = {}
       t = 0.0
       for x in it.product(range(2), repeat=k):
           prob_for_x = meas_prob(x)
           intervals[x] = (t, t + prob_for_x)
           t += prob_for_x
       print(f"{x} with probability {prob_for_x}")

```

```

(0, 0, 0, 0) with probability 0.12499999999999983
(0, 0, 0, 1) with probability 0.0
(0, 0, 1, 0) with probability 0.0
(0, 0, 1, 1) with probability 0.12499999999999983
(0, 1, 0, 0) with probability 0.0
(0, 1, 0, 1) with probability 0.12499999999999983
(0, 1, 1, 0) with probability 0.12499999999999983

```

```

(0, 1, 1, 1) with probability 0.0
(1, 0, 0, 0) with probability 0.0
(1, 0, 0, 1) with probability 0.12499999999999983
(1, 0, 1, 0) with probability 0.12499999999999983
(1, 0, 1, 1) with probability 0.0
(1, 1, 0, 0) with probability 0.12499999999999983
(1, 1, 0, 1) with probability 0.0
(1, 1, 1, 0) with probability 0.0
(1, 1, 1, 1) with probability 0.12499999999999983

```

```

[172]: x_samples = []
for _ in range(k): # when k is small, it's better to repeat a bit more
    rand_num = rng.random()
    for x in it.product(range(2), repeat=k):
        if intervals[x][0] < rand_num and rand_num < intervals[x][1]:
            x_samples.append(x)
            break

```

```

[173]: x_samples

```

```

[173]: [(0, 1, 1, 0), (1, 0, 1, 0), (0, 1, 0, 1), (0, 0, 0, 0)]

```

```

[174]: # find s by solving x*s = 0 mod 2 for the x in the above samples
def inn_prod(x, s):
    return sum(xi * si for xi, si in it.zip_longest(x, s))

def predicate(s):
    if s == (0,) * k:
        return False
    inn_prod_mod_2_list = [inn_prod(x, s) % 2 for x in x_samples]
    if 1 in inn_prod_mod_2_list:
        return False
    return True

for s_cand in it.product(range(2), repeat=k):
    if predicate(s_cand):
        print(f"{s_cand} is a candidate for the secret")

```

(1, 1, 1, 1) is a candidate for the secret

Variation: every time we get a new candidate for $x^i = x_1^i \dots x_k^i$, we set the system of linear equations

$$x_1^i s_1 + x_2^i s_2 + \dots + x_k^i s_k = 0 \pmod{2} \quad (i = 1, 2, \dots)$$

and continue until there is only one non-zero solution for s.

```
[175]: import sympy as sy
# prepare variables s_1, ..., s_k
s_var = [sy.Symbol(f"s_{j+1}") for j in range(k)]

x_samples = []
l_poly = []
for trial_cnt in range(2 * k):
    rand_num = rng.random()
    for x in it.product(range(2), repeat=k):
        if intervals[x][0] < rand_num and rand_num < intervals[x][1]:
            new_x = x
            break
    # ignore x^i = (0,...,0), otherwise set linear polynomial with
    # coefficients x^i_j
    if new_x == (0,) * k:
        continue
    else:
        f2 = sy.FF(2)
        i_th_lin_poly = sy.Poly(sum(f2(new_x[j]) * s_var[j] for j in
    range(k)))
        l_poly.append(i_th_lin_poly)
    # find solutions
    ans_set = sy.solve(l_poly)
    print(ans_set)
    if len(ans_set) == k - 1:
        print(f"got a unique solution after {trial_cnt + 1} trials")
        break
```

```
{s_1: -s_3}
{s_1: s_4, s_3: -s_4}
{s_1: s_4, s_2: -s_4, s_3: -s_4}
got a unique solution after 3 trials
```

```
[176]: # find free variables
free_vars = set()
for key in ans_set.keys():
    free_vars |= ans_set[key].free_symbols
# get the nonzero solution as a string
if len(free_vars) == 0:
    subs_dict = {}
else:
    subs_dict = {list(free_vars)[0]: 1}

ans_seq = ()
for the_var in s_var:
```

```

    if the_var in ans_set.keys():
        ans_seq += (ans_set[the_var].subs(subs_dict) % 2,)
    else:
        ans_seq += (1,)

ans_seq

```

[176]: (1, 1, 1, 1)

optimization 1: sparse matrices try bigger k with sparse matrix algorithms from scipy library

```

[177]: # set secret string
k = 7
s = tuple([rng.integers(2) for _ in range(k)])
if 1 not in s:
    # try again
    s = tuple([rng.integers(2) for _ in range(k)])
    if 1 not in s:
        print("we could not generate a secret string.")

```

[178]: s

[178]: (0, 0, 1, 0, 1, 1, 0)

```

[179]: # sparse matrix for large k
import scipy.sparse as spa

def ind_from_seq(x):
    """Convert the binary sequence of length k to matrix index 0...
    2**len(x)-1."""
    return sum(2**((len(x)-j-1) * x[j]) for j in range(len(x)))

def comp_basis_sp(x):
    """Return 2-d column vector representing the x-th basis."""
    mat_size = 2**len(x)
    res = spa.dok_matrix((mat_size,1))
    res[ind_from_seq(x),0] = 1
    return res.tocsr()

def U_sp(k, f):
    """Return U_f gate for Simon's algorithm."""
    mat_size = 2**(2*k)
    res = spa.dok_matrix((mat_size,mat_size))
    ind_list = it.product(range(2), repeat=2*k)
    for in_x in ind_list:

```

```

        out_x = in_x[0:k] + termwise_XOR(f(in_x[0:k]), in_x[k:2*k])
        res[ind_from_seq(out_x), ind_from_seq(in_x)] = 1
    return res.tocsr()

```

```
[180]: H_pow_ampl_sp = spa.kron(tens_pow_H(k), np.identity(2**k))
```

```
[181]: simon_gate_sp = H_pow_ampl_sp @ U_sp(k, f_from_s) @ H_pow_ampl_sp
```

```
[182]: v = simon_gate_sp.dot(comp_basis_sp([0] * (2 * k)))
```

```
[183]: # sparse version of helper commands
def proj(x):
    """Return projection for the computational basis."""
    basis_vec = comp_basis_sp(x)
    return basis_vec @ basis_vec.T

def meas_prob(x):
    """Return the probability of observing x in the first half of v."""
    return (v.T @ spa.kron(proj(x), np.identity(2**k)) @ v)[0,0]

```

```
[184]: intervals = {}
t = 0.0
for x in it.product(range(2), repeat=k):
    prob_for_x = meas_prob(x)
    intervals[x] = (t, t + prob_for_x)
    t += prob_for_x

```

```
[185]: x_samples = []
for _ in range(k): # when k is small, it's better to repeat a bit more
    rand_num = rng.random()
    for x in it.product(range(2), repeat=k):
        if intervals[x][0] < rand_num and rand_num < intervals[x][1]:
            x_samples.append(x)
            break

```

```
[186]: x_samples
```

```
[186]: [(1, 1, 0, 1, 1, 1, 0), (1, 0, 1, 1, 0, 1, 0), (0, 1, 0, 1, 0, 0, 1), (0, 1, 0, 1, 1, 1, 1), (1, 1, 0, 1, 1, 1, 1), (1,
```

```
[187]: for s_cand in it.product(range(2), repeat=k):
    if predicate(s_cand):
        print(f"{s_cand} is a candidate for the secret")

```

```

(0, 0, 1, 0, 1, 1, 0) is a candidate for the secret
(0, 1, 0, 1, 1, 1, 0) is a candidate for the secret
(0, 1, 1, 1, 0, 0, 0) is a candidate for the secret

```

optimization 2: fast Walsh-Hadamard transform we avoid creating matrices and do in-place manipulation as much as possible

```
[188]: # set secret string
k = 10
s = tuple([rng.integers(2) for _ in range(k)])
if 1 not in s:
    # try again
    s = tuple([rng.integers(2) for _ in range(k)])
    if 1 not in s:
        print("we could not generate a secret string.")

s
```

```
[188]: (1, 0, 1, 1, 0, 0, 0, 0, 1, 1)
```

```
[189]: def fwht(a):
    """Perform the in-place Fast Walsh-Hadamard Transform of array a."""
    h = 1
    n = len(a)
    while h < n:
        # perform FWHT
        for i in range(0, n, h * 2):
            for j in range(i, i + h):
                x = a[j]
                y = a[j + h]
                a[j] = x + y
                a[j + h] = x - y
        # normalize and increment h
        a *= 1/np.sqrt(2)
        h *= 2

def H_pow_ampl_fwht(x, k):
    """
    Apply ( $H^{\otimes k} \otimes I_{2^k}$ ) to x using FWHT

    Input:
        x: 1D array, length  $2^{(2k)}$ 
        k: int, number of qubits on which  $H^{\otimes k}$  acts

    Output:
        1D array  $y = (H^{\otimes k} \otimes I_{2^k}) x$ 
    """
    x = x.reshape(2**k, 2**k) # shape (2^k, 2^k)
```

```

# Apply FWHT along axis 0
for col in range(2**k):
    fwht(x[:, col])

return x.reshape(2*(2*k))

```

```

[190]: def apply_index_permutation(x, perm):
        """
        Apply basis index permutation y such that y[perm[i]] = x[i].

        Input:
            x: 1D array, length N
            perm: 1D array-like of length N; a permutation of range(N),

        Output:
            1D array y = P @ x, where P|i> = |perm[i]>.
        """
        perm = np.asarray(perm)
        y = np.empty_like(x)
        y[perm] = x # y[perm[i]] = x[i]
        return y

```

```

[191]: perm_rep_Uf = [0] * 2*(2*k)
        ind_list = it.product(range(2), repeat=2*k)
        for in_x in ind_list:
            out_x = in_x[0:k] + termwise_XOR(f_from_s(in_x[0:k]), in_x[k:2*k])
            perm_rep_Uf[ind_from_seq(in_x)] = ind_from_seq(out_x);

```

```

[192]: def comp_basis_big(x):
        """Return 2-d column vector representing the x-th basis."""
        res = np.array([0] * 2*len(x), dtype='float64')
        res[ind_from_seq(x)] = 1
        return res

```

```

[193]: start_vec = comp_basis_big([0] * (2 * k))
        first_H_eff_vec = H_pow_ampl_fwht(start_vec, k)
        permed_vec = apply_index_permutation(first_H_eff_vec, perm_rep_Uf)
        second_H_eff_vec = H_pow_ampl_fwht(permed_vec, k)

```

```

[194]: # adaptation of helper commands
        def pair_first_half(v, w):
            """
            Contract a 2k-qubit state v with a k-qubit state w over the first k_
            _qubits.
            (Components are assumed to be real.)

```

Input:

v: 1D array, length 2^{2k} (state on $2k$ qubits)
w: 1D array, length 2^k (state on first k qubits)
k: number of qubits in w and in the first subsystem

Output:

1D array of length 2^k representing the resulting state on the last k qubits.

"""

Reshape v as a bipartite state: (first k qubits, last k qubits)

v_resaped = v.reshape(2^k , 2^k)

Contract over the first subsystem: $u = (\langle w | \otimes I) | v \rangle$

This is $u = w.conj().T @ v_resaped$

return w.T @ v_resaped

```
def meas_prob(x,v):
```

"""Return the probability of observing x in the first half of v."""

w = comp_basis_big(x)

u = pair_first_half(v, w)

return np.dot(u, u)

```
[195]: intervals = {}  
t = 0.0  
for x in it.product(range(2), repeat=k):  
    prob_for_x = meas_prob(x,second_H_eff_vec)  
    intervals[x] = (t, t + prob_for_x)  
    t += prob_for_x
```

perform the measurement to get strings $x^1, x^2, \dots$, then solve the linear equations to get the secret string

```
[196]: import sympy as sy  
# prepare variables s_1, ..., s_k  
s_var = [sy.Symbol(f"s_{j+1}") for j in range(k)]  
  
x_samples = []  
l_poly = []  
for trial_cnt in range(2 * k):  
    rand_num = rng.random()  
    for x in it.product(range(2), repeat=k):  
        if intervals[x][0] < rand_num and rand_num < intervals[x][1]:  
            new_x = x  
            break
```

```

    # ignore  $x^i = (0, \dots, 0)$ , otherwise set linear polynomial with
    coefficients  $x^i_j$ 
    if new_x == (0,) * k:
        continue
    else:
        f2 = sy.FF(2)
        i_th_lin_poly = sy.Poly(sum(f2(new_x[j]) * s_var[j] for j in
    range(k)))
        l_poly.append(i_th_lin_poly)
    # find solutions
    ans_set = sy.solve(l_poly)
    print(ans_set)
    if len(ans_set) == k - 1:
        print(f"got a unique solution after {trial_cnt + 1} trials")
        break

```

```

{s_2: -s_3 - s_4 - s_6 - s_8}
{s_1: s_4 - s_5 - s_7, s_2: -s_3 - s_4 - s_6 - s_8}
{s_1: s_4 - s_5 - s_7, s_10: -2*s_4 + s_5 - s_6 + s_7 - s_9, s_2: -s_3 -
    s_4 -
s_6 - s_8}
{s_1: -s_8 - s_9, s_10: -s_5 - s_6 - s_7 + 2*s_8 + s_9, s_2: -s_3 - s_5 -
    s_6 -
s_7 + s_9, s_4: s_5 + s_7 - s_8 - s_9}
{s_1: -s_8 - s_9, s_10: -s_5 - s_6 - s_7 + 2*s_8 + s_9, s_2: s_5 - 2*s_8,
    s_3:
-2*s_5 - s_6 - s_7 + 2*s_8 + s_9, s_4: s_5 + s_7 - s_8 - s_9}
{s_1: -s_8 - s_9, s_10: -s_6 - s_7 + 2*s_8 + s_9, s_2: -2*s_8, s_3: -s_6 -
    s_7 +
2*s_8 + s_9, s_4: s_7 - s_8 - s_9, s_5: 0}
{s_1: -s_8 - s_9, s_10: -s_6 + s_8 - s_9, s_2: -2*s_8, s_3: -s_6 + s_8 -
    s_9,
s_4: s_9, s_5: 0, s_7: s_8 + 2*s_9}
{s_1: -s_9, s_10: -s_6 - s_9, s_2: 0, s_3: -s_6 - s_9, s_4: s_9, s_5: 0,
    s_7:
2*s_9, s_8: 0}
{s_1: -s_9, s_10: 0, s_2: 0, s_3: 0, s_4: s_9, s_5: 0, s_6: -s_9, s_7:
    2*s_9,
s_8: 0}
got a unique solution after 9 trials

```

```

[197]: # find free variables
free_vars = set()
for key in ans_set.keys():
    free_vars |= ans_set[key].free_symbols

```

```

# get the nonzero solution as a string
if len(free_vars) == 0:
    subs_dict = {}
else:
    subs_dict = {list(free_vars)[0]: 1}

ans_seq = ()
for the_var in s_var:
    if the_var in ans_set.keys():
        ans_seq += (ans_set[the_var].subs(subs_dict) % 2,)
    else:
        ans_seq += (1,)

ans_seq

```

[197]: (1, 0, 0, 1, 0, 1, 0, 0, 1, 0)

How about the classical algorithm?

```

[198]: # query count estimate
np.sqrt(2**k)

```

[198]: 32.0

```

[199]: res_rec = {} # dictionary of f(x) as key and x as value
ind_list = it.product(range(2), repeat=k)
for _ in range(2**k):
    x = tuple([rng.integers(2) for _ in range(k)])
    if x in res_rec.values():
        continue
    f_x = f_from_s(x)
    if f_x in res_rec.keys():
        prev_x = res_rec[f_x]
        found_secret = [x[i] ^ prev_x[i] for i in range(k)]
        print(f"found secret {found_secret} after {1 + len(res_rec)}_
↳queries")
        break
    # if f(x) is not found, record the results and continue with next x
    res_rec[f_x] = x

```

found secret [1, 0, 1, 1, 0, 0, 0, 0, 1, 1] after 24 queries

12 Lecture 15

12.1 Extended Euclidean division algorithm

find x and y satisfying

$$ax + by = \gcd(a, b)$$

construct three sequences r_i , s_i , and t_i in the following way: - $i = 0$: $r_0 = a$, $s_0 = 1$, $t_0 = 0$; - $i = 1$: $r_1 = b$, $s_1 = 0$, $t_1 = 1$; - the $(i + 1)$ -th step: + find q_i such that $r_{i+1} = r_{i-1} - q_i r_i$ holds for $0 \leq r_{i+1} < r_i$; + if $r_{i+1} = 0$, then terminate the algorithm with output $x = s_i$, $y = t_i$, and $\gcd(a, b) = r_i$; + otherwise set $s_{i+1} = s_{i-1} - q_i s_i$ and $t_{i+1} = t_{i-1} - q_i t_i$ and proceed to the next step.

The algorithm is set up to keep the condition

$$as_i + bt_i = r_i \quad (i = 0, 1, \dots)$$

while making r_i smaller at each step. At each step, r_{i+1} is the remainder of r_{i-1} divided by r_i , and q_i is the quotient. We get the correct answer because $\gcd(a, b)$ is the smallest positive integer solution k for $ax + by = k$ with (possibly negative) integers x and y .

```
[200]: def ext_Euclid_div(a, b, report=False):
        """Returns (gcd(a,b), x, y) such that a * x + b * y = gcd(a, b)."""
        # i = 0 and i = 1
        r = [a, b]
        s = [1, 0]
        t = [0, 1]
        for i in range(2, max(a,b)**2):
            q_i, rem = divmod(r[-2], r[-1])
            if rem == 0:
                if report:
                    print(f"found {r[-1]} = gcd({a}, {b}) after {i} steps")
                return (r[-1], s[-1], t[-1])
            # add next terms, then continue
            r.append(rem)
            s.append(s[-2] - q_i * s[-1])
            t.append(t[-2] - q_i * t[-1])
            if report:
                print(f"we now have r: {r[-1]}, s: {s[-1]}, t: {t[-1]}")
```

```
[201]: a, b = 40902, 24140 # from Knuth's book
gcd, x, y = ext_Euclid_div(a, b, report=True)
```

```
we now have r: 16762, s: 1, t: -1
we now have r: 7378, s: -1, t: 2
we now have r: 2006, s: 3, t: -5
we now have r: 1360, s: -10, t: 17
we now have r: 646, s: 13, t: -22
we now have r: 68, s: -36, t: 61
```

we now have $r: 34$, $s: 337$, $t: -571$
found $34 = \gcd(40902, 24140)$ after 9 steps

```
[202]: a * x + b * y, gcd == a * x + b * y
```

```
[202]: (34, True)
```

12.2 RSA cryptsystem

Preparation: first Alice chooses (big) prime numbers p and q

```
[203]: # helper function to compute exponential modulo k for big exponent
from functools import reduce
def pow_mod(base, expo, k):
    """Returns base**expo mod k."""
    if expo < 0:
        return None
    return reduce(lambda s, t: (s * t) % k, [base] * expo)
```

```
[204]: import numpy as np
rng = np.random.default_rng()

from math import sqrt

def is_prime(n: np.int64):
    """
    Miller-Rabin primality test.

    A return value of False means n is certainly not prime. A return_
    _value of
    True means n is very likely a prime.
    """
    # special cases for small n
    if n in [2, 3, 5, 7]:
        return True
    elif n < 10:
        return False

    s = 0
    d = n - 1
    while d % 2 == 0:
        d >>= 1
        s += 1
    assert(2**s * d == n - 1)
```

```

def trial_composite(a):
    if pow_mod(a, d, n) == 1:
        return False
    for i in range(s):
        if pow_mod(a, 2**i * d, n) == n-1:
            return False
    return True

for i in range(8): # number of trials
    a = rng.integers(2, n)
    if trial_composite(a):
        return False

return True

def getLargePrime(keysize=1024, report=False):
    """Generate a random prime number with specified keysize"""
    for try_count in range(keysize ** 2):
        num = rng.integers(2**(keysize-1), 2**(keysize))
        if is_prime(num):
            if report:
                print(f"found a prime number {num} after {try_count} trials.")
            return num

```

```

[205]: p = getLargePrime(10, report=True)
       q = getLargePrime(10, report=True)
       p, q

```

found a prime number 947 after 3 trials.
found a prime number 743 after 0 trials.

[205]: (947, 743)

she chooses another integer e that is coprime to the least common multiple $\text{lcm}(p-1, q-1)$

```

[206]: gcd, _, _ = ext_Euclid_div(p-1, q-1)
       c = int((p-1) * (q-1) / gcd) # lcm
       e_cands = [3, 5, 7, 11, 13, 17, 19]
       for e_cand in e_cands:
           gcd_e_c, _, _ = ext_Euclid_div(e_cand, c)
           if gcd_e_c == 1:
               e = e_cand
               break
       e = None
       c, e

```

[206]: (350966, 3)

she finds d satisfying $ed \equiv 1 \pmod{\text{lcm}(p-1, q-1)}$ (this is the *private key*)

```
[207]: _, d, _ = ext_Euclid_div(e, c)
        if d < 0:
            d = d % c
        d, (e * d) % c
```

[207]: (116989, 1)

she advertises $N = pq$ and e as the *public key*.

```
[208]: N = p * q
        print(f"Public key: N={N}, e={e}")
```

Public key: N=703621, e=3

Now, suppose that Bob wants to send a message represented by an integer x to Alice. He first checks that x does not share prime factors with N (if it does have one of them as a prime factor, he asks for a new public key)

```
[209]: x = 42
```

```
[210]: gcd, _, _ = ext_Euclid_div(x, N)
        gcd == 1 # should be true
```

[210]: True

he computes $y = x^e$, and sends it to Alice

```
[211]: y = pow_mod(x, e, N)
        y
```

[211]: 74088

Alice recovers x as $x \equiv y^d \pmod{N}$.

```
[212]: pow_mod(y, d, N)
```

[212]: 42

13 Libraries

13.1 SymPy

[SymPy](#) is a library for symbolic mathematics. This is useful when we want to understand algebraic manipulations involved in (quantum) algorithms, like using variables to represent the components of state vectors.

```
[213]: %%script false --no-raise-error # safeguard in case this is unintended -
      ↪ remove this line to install sympy
      %pip install sympy
```

```
[214]: import sympy as sy
```

Vectors can be represented as “matrices” constructed from 1-dimensional arrays, while matrices are constructed from 2-dimensional arrays

```
[215]: vec = sy.Matrix([1, 0]) # |0>
      Xmat = sy.Matrix([[0, 1],
                        [1, 0]])
```

The matrix product can be represented by the usual * operator. This also takes care of converting 1-dim arrays to 2-dim column vectors.

```
[216]: Xmat * vec # expect |1>
```

```
[216]: 
$$\begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

```

To create formal variables, use symbols command.

```
[217]: x1, x2, x3, x4 = sy.symbols('a b c d')
      test_mat = sy.Matrix([[x1, x2],
                           [x3, x4]])
      test_mat
```

```
[217]: 
$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

```

```
[218]: test_mat * vec # product with |0> would gives the first column
```

```
[218]: 
$$\begin{bmatrix} a \\ c \end{bmatrix}$$

```

The Kronecker product is sympy.physics.quantum.TensorProduct.

```
[219]: from sympy.physics.quantum import TensorProduct
      TensorProduct(Xmat, sy.eye(2)) # block matrix whose blocks are I_2 or
      ↪ the 0 matrix
```

```
[219]: 
$$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

```

```
[220]: TensorProduct(sy.eye(2), Xmat) # block matrix whose blocks are X or the
      ↪ 0 matrix
```

```
[220]:
```

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

13.2 NumPy

[NumPy](#) is an extensive scientific computing library. This is useful when we want to simulate algorithms involving some numerics, like probabilistic algorithms.

```
[221]: import numpy as np
```

Vectors can be represented by 1-dimensional arrays, while matrices are represented by 2-dimensional arrays.

```
[222]: vec = np.array([1, 0]) # |0>
Xmat = np.array([[0, 1],
                 [1, 0]])
```

The matrix product can be represented by the @ infix operator. This also takes care of converting 1-dim arrays to 2-dim column vectors.

```
[223]: Xmat @ vec # expect |1>
```

```
[223]: array([0, 1])
```

The Kronecker product is `np.kron`.

```
[224]: np.kron(Xmat, np.identity(2)) # block matrix whose blocks are I_2 or the 0
    ↪ matrix
```

```
[224]: array([[0., 0., 1., 0.],
              [0., 0., 0., 1.],
              [1., 0., 0., 0.],
              [0., 1., 0., 0.]])
```

```
[225]: np.kron(np.identity(2), Xmat) # block matrix whose blocks are X or the 0
    ↪ matrix
```

```
[225]: array([[0., 1., 0., 0.],
              [1., 0., 0., 0.],
              [0., 0., 0., 1.],
              [0., 0., 1., 0.]])
```

13.3 Qiskit

[Qiskit](#) is a quantum computing library developed by IBM. It takes a bit of effort to set up, but you can get a good feeling of how a quantum computing service looks like. Besides the official

documentation, the [tutorial](#) hosted at google is also a good entry point.

Edit and evaluate the next four cells to install the library.

```
[226]: %%script false --no-raise-error # safeguard in case this is unintended -  
      ↪ remove this line to install qiskit  
      %pip install qiskit
```

```
[227]: %%script false --no-raise-error # safeguard in case this is unintended -  
      ↪ remove this line to install qiskit  
      %pip install qiskit-ibm-runtime
```

```
[228]: %%script false --no-raise-error # safeguard in case this is unintended -  
      ↪ remove this line to install qiskit  
      %pip install pylatexenc
```

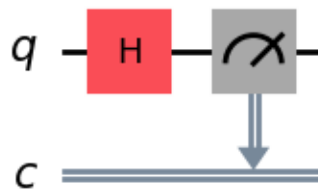
```
[229]: %%script false --no-raise-error # safeguard in case this is unintended -  
      ↪ remove this line to install qiskit  
      %pip install matplotlib
```

```
[230]: import qiskit as qs
```

Construct a quantum circuit with one qubit input. It applies the Hadamard gate to the input qubit, then perform measurement with respect to the computational basis. The result of measurement is recorded in the “classical register” bit.

```
[231]: circ = qs.QuantumCircuit(1,1) # quantum circuit with one qubit and one  
      ↪ classical bit  
      circ.h(0) # add the Hadamard gate to the qubit wire  
      circ.measure(0, 0) # add measurement apparatus, telling it to record the  
      ↪ result on the 0-th classical bit  
      circ.draw("mpl", cregbundle=False) # draw the circuit (cregbundle option  
      ↪ to suppress unwanted labels)
```

[231]:



To run the circuit (in a simulation), we need the following code

```
[232]: from qiskit.providers.basic_provider import BasicProvider
from qiskit.compiler import transpile
simulator = BasicProvider().get_backend()
circ_tr = transpile(circ, simulator)
job = simulator.run(circ_tr)
result = job.result()
result.get_counts(circ)
```

```
[232]: {'1': 528, '0': 496}
```

The default input qubit state is $|0\rangle$. So the above circuit first turns $|0\rangle$ to $|+\rangle$ by the Hadamard gate. If we measure this state in the computational basis, we expect the case 0 and the case 1 with both 50% probability.

for quantum circuits without measurements, it is also possible to get its effect as a unitary get acting on the input vector (by default $|0 \dots 0\rangle$) as follows:

```
[233]: from qiskit.quantum_info import Statevector

circ = qs.QuantumCircuit(1) # quantum circuit with one qubit
circ.h(0) # add the Hadamard gate to the qubit wire

state = Statevector.from_circuit(circ)
#draw using latex
state.draw('latex')
```

```
[233]:
```

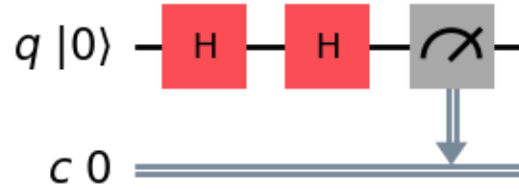
$$\frac{\sqrt{2}}{2}|0\rangle + \frac{\sqrt{2}}{2}|1\rangle$$

If we want a different initial state v for the algorithm, we can either add some gates to transform the $|0\rangle$ state into v , or use the `initialize` command of the circuit.

Suppose we want to run the above circuit with initial state $|+\rangle$.

```
[234]: circ = qs.QuantumCircuit(1,1) # same setup
circ.h(0) # to turn the default state |0> into |+>
# the rest is the same
circ.h(0)
circ.measure(0, 0)
circ.draw("mpl", initial_state=True, cregbundle=False)
```

```
[234]:
```



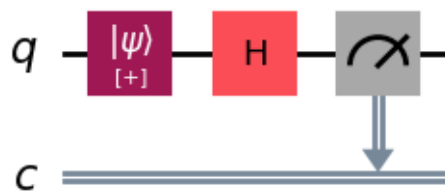
```
[235]: # then run the circuit
circ_tr = transpile(circ, simulator)
job = simulator.run(circ_tr)
result = job.result()
result.get_counts(circ)
```

[235]: {'0': 1024}

If we apply the Hadamard gate to $|+\rangle$, we get $|0\rangle$. So the measurement would be the case 0 for 100% of times.

```
[236]: circ = qs.QuantumCircuit(1,1) # same setup
circ.initialize("+",[0]) # prepare the |+> state on the 0-th qubit input
# the rest is the same
circ.h(0)
circ.measure(0, 0)
circ.draw("mpl", cregbundle=False)
```

[236]:



```
[237]: # then run the circuit
circ_tr = transpile(circ, simulator)
job = simulator.run(circ_tr)
```

```
result = job.result()
result.get_counts(circ)
```

[237]: {'0': 1024}

Qiskit comes with a collection of standard gates that you can add to the circuit by `circ.cx(control-qubit-index, target-qubit-index)` for the CNOT gate, `circ.y(qubit-index)` for the Pauli Y gate, etc. See the [document](#) for details.

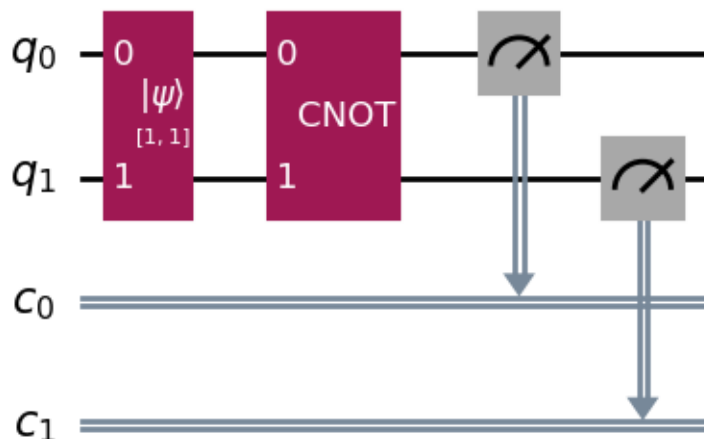
To achieve a more flexible algorithm, you want to supply a unitary matrix representing the desired quantum gate. This can be added to the circuit by the `unitary()` as described in the [document](#).

```
[238]: # construct the CNOT by hand
# computational basis
zeroKet = np.array([1, 0])
oneKet = np.array([0, 1])
zeroZeroKet = np.kron(zeroKet, zeroKet)
zeroOneKet = np.kron(zeroKet, oneKet)
oneZeroKet = np.kron(oneKet, zeroKet)
oneOneKet = np.kron(oneKet, oneKet)
# CNOT transforms 00 -> 00, 01 -> 01, 10 -> 11, 11 -> 10
# use atleast_2d to force 2d-array interpretation
CNOT_mat = np.atleast_2d(zeroZeroKet).T @ np.atleast_2d(zeroZeroKet)
CNOT_mat += np.atleast_2d(zeroOneKet).T @ np.atleast_2d(zeroOneKet)
CNOT_mat += np.atleast_2d(oneZeroKet).T @ np.atleast_2d(oneOneKet)
CNOT_mat += np.atleast_2d(oneOneKet).T @ np.atleast_2d(oneZeroKet)
CNOT_mat
```

```
[238]: array([[1, 0, 0, 0],
              [0, 1, 0, 0],
              [0, 0, 0, 1],
              [0, 0, 1, 0]])
```

```
[239]: circ = qs.QuantumCircuit(2,2) # two qubits, two classical registers
circ.initialize("11",[0,1]) # prepare the |11> state on the 0-th and the_
    1-st qubits
circ.unitary(CNOT_mat, [0, 1], label="CNOT")
circ.measure(0, 0)
circ.measure(1, 1)
circ.draw("mpl", cregbundle=False)
```

[239]:



```
[240]: circ_tr = transpile(circ, simulator)
job = simulator.run(circ_tr)
result = job.result()
result.get_counts(circ) # expect 10
```

```
[240]: {'10': 1024}
```

Qiskit bit convention

Integers: When interpreting bits as a number, bit 0 is the least significant bit, and bit $n - 1$ the most significant. This is helpful when coding because each bit has the value 2^{label} (label being the qubit's index in `QuantumCircuit.qubits`).

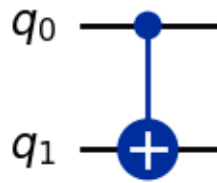
Strings: When displaying or interpreting a list of bits (or qubits) as a string, bit $n - 1$ is the leftmost bit, and bit 0 is the rightmost bit. This is because we usually write numbers with the most significant digit on the left, and in Qiskit, bit $n - 1$ is interpreted as the most significant bit. This occasionally causes confusion when interpreting a string of bits, as you might expect the leftmost bit to be bit 0, whereas it usually represents bit $n - 1$.

This means that the two-qubit states are represented by $v_1 \otimes v_0$, while v_0 still goes to the top wire in the circuit diagram.

Let's check this with the effect of CNOT gate.

```
[241]: circ = qs.QuantumCircuit(2) # two qubits
circ.cx(0,1)
circ.draw("mpl", cregbundle=False)
```

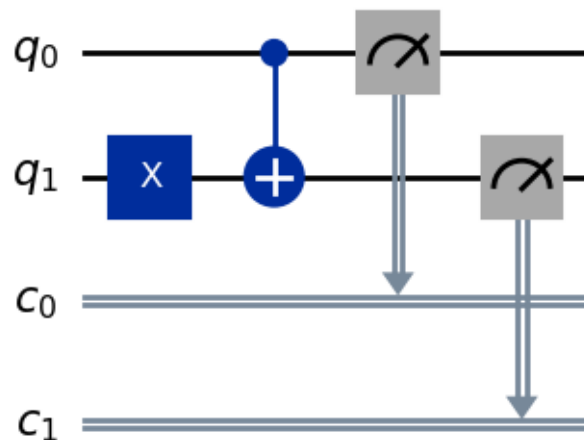
```
[241]:
```



If we give the input $|ij\rangle = |i\rangle \otimes |j\rangle$ to this circuit, the state $|j\rangle$ will be used as the control.

```
[242]: circ = qs.QuantumCircuit(2,2) # two qubits, two classical registers
# transform |00> to |10>
circ.x(1)
# we could also use the following
# circ.initialize("10",[0,1]) # prepare the |10> state on the 0-th and
# the 1-st qubits
circ.cx(0,1)
circ.measure(0, 0)
circ.measure(1, 1)
circ.draw("mpl", cregbundle=False)
```

[242]:

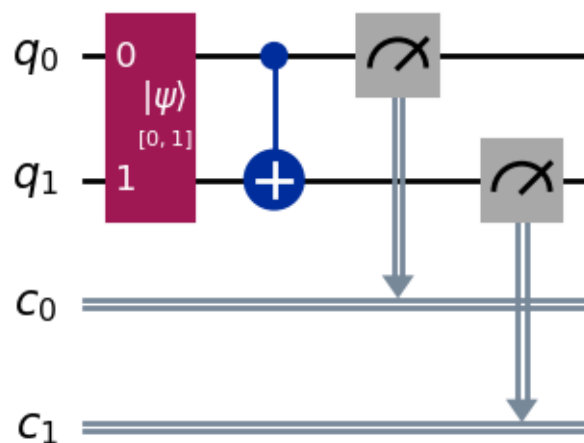


```
[243]: circ_tr = transpile(circ, simulator)
job = simulator.run(circ_tr)
result = job.result()
result.get_counts(circ) # expect 10 because 0 for the control bit
↳ wouldn't change the target bit
```

```
[243]: {'10': 1024}
```

```
[244]: circ = qs.QuantumCircuit(2,2)
circ.initialize("01",[0,1]) # prepare the |01> state on the 0-th and the
↳ 1-st qubits
circ.cx(0,1)
circ.measure(0, 0)
circ.measure(1, 1)
circ.draw("mpl", cregbundle=False)
```

```
[244]:
```



```
[245]: circ_tr = transpile(circ, simulator)
job = simulator.run(circ_tr)
result = job.result()
result.get_counts(circ) # expect 11 because 1 for the control bit would
↳ change the target bit
```

```
[245]: {'11': 1024}
```