1 Quantum Computation

1.1 Exam 1 Second part

1.2 Francisco Javier Vazquez Tavares

[116]:
```
import numpy as np



# Usefull function to print matrices (https://gist.github.com/braingineer/
↪d801735dac07ff3ac4d746e1f218ab75)
def matprint(mat, fmt="g"):
    col_maxes = [max([len(("{:"+fmt+"}").format(x)) for x in col]) for col in␣
↪mat.T]
    for x in mat:
        for i, y in enumerate(x):
            print(("{:"+str(col_maxes[i])+fmt+"}").format(y), end="  ")
        print("")
```

1.3 Eigenvalues of Pauli Matrices

[117]:
```
# Definition of the Pauli Matrices
sx = np.array([[0, 1],
              [1, 0]])

sy = np.array([[0, 0-1j],
              [0+1j, 0]])

sz = np.array([[1, 0],
              [0, -1]])

# Print the matrix
print("Sx Matrix\n")
matprint(sx, fmt="g")
print("\n")
print("Sy Matrix\n")
matprint(sy, fmt="g")
print("\n")
print("Sz Matrix\n")
matprint(sz, fmt="g")
print("\n")



# Compute the eigenvalues and eigenvectors for each matrix
eigenvaluesSx, eigenvectorsSx = np.linalg.eig(sx)
eigenvaluesSy, eigenvectorsSy = np.linalg.eig(sy)
eigenvaluesSz, eigenvectorsSz = np.linalg.eig(sz)
```

```python
# Print the answers
for i in range(len(eigenvaluesSx)):
    print(f"Eigenvalue {i+1}: {eigenvaluesSx[i]}")
    print(f"Corresponding eigenvector:\n{eigenvectorsSx[:, i]}\n")


for i in range(len(eigenvaluesSy)):
    print(f"Eigenvalue {i+1}: {eigenvaluesSy[i]}")
    print(f"Corresponding eigenvector:\n{eigenvectorsSy[:, i]}\n")


for i in range(len(eigenvaluesSz)):
    print(f"Eigenvalue {i+1}: {eigenvaluesSz[i]}")
    print(f"Corresponding eigenvector:\n{eigenvectorsSz[:, i]}\n")
```

```
Sx Matrix

0  1
1  0


Sy Matrix

0+0j  0-1j
0+1j  0+0j


Sz Matrix

1   0
0  -1


Eigenvalue 1: 1.0
Corresponding eigenvector:
[0.70710678 0.70710678]

Eigenvalue 2: -1.0
Corresponding eigenvector:
[-0.70710678  0.70710678]

Eigenvalue 1: (0.9999999999999996+0j)
Corresponding eigenvector:
[-0.        -0.70710678j  0.70710678+0.j        ]

Eigenvalue 2: (-0.9999999999999999+0j)
Corresponding eigenvector:
```

```
[0.70710678+0.j          0.          -0.70710678j]

Eigenvalue 1: 1.0
Corresponding eigenvector:
[1. 0.]

Eigenvalue 2: -1.0
Corresponding eigenvector:
[0. 1.]
```

In this part, the command np.linalg.eig computes the characteristic polynomial of a given squared matrix ($\det(A - \lambda I) = 0$) and find its roots to compute the eigenvalues. Then, using the eigenvalues, it computes the eigenvectors by solving the linear system of equation given by $(A - \lambda_n I)|n\rangle = |0\rangle$.

1.4 Gram Schmidt Orthonormalization

[118]:
```
def gram_schmidt(vectors):
    if len(vectors) == 0:
        return []

    # Orthonormalize the rest of the vectors recursively
    u_rest = gram_schmidt(vectors[1:])

    # Start with the first vector (keep complex type)
    u = vectors[0].copy()

    # Subtract projections onto each orthonormal vector in u_rest
    for v in u_rest:
        # For complex vectors, we need the conjugate of the inner product
        proj = np.vdot(v, u)  # This is <v, u> = vu (conjugate-linear in first␣
 ↪argument)
        u = u - proj * v

    # Normalize the resulting vector (complex norm)
    norm = np.linalg.norm(u)  # This handles complex numbers correctly
    u_normalized = u / norm

    return [u_normalized] + u_rest

def check_orthonormality(vectors):
    """
    Verify that a set of complex vectors is orthonormal.

    Parameters:
    vectors (list): List of numpy arrays representing vectors

    Returns:
```

```
    bool: True if vectors are orthonormal, False otherwise
    """
    n = len(vectors)

    # Check if all vectors have unit norm
    for i, v in enumerate(vectors):
        norm = np.linalg.norm(v)
        if not np.isclose(norm, 1.0):
            print(f"Vector {i} has non-unit norm: {norm}")
            return False

    # Check if all pairs of vectors are orthogonal
    for i in range(n):
        for j in range(i+1, n):
            # For complex vectors, we need to check both <v_i, v_j> and <v_j,␣
↪v_i>
            # But orthogonality means <v_i, v_j> = 0
            dot_product = np.vdot(vectors[i], vectors[j])
            if not np.isclose(dot_product, 0.0):
                print(f"Vectors {i} and {j} are not orthogonal: inner product =␣
↪{dot_product}")
                return False

    return True

def print_vectors(vectors, title="Vectors"):
    """
    Print vectors in a clean, readable format.

    Parameters:
    vectors (list): List of numpy arrays representing vectors
    title (str): Title for the printed section
    """
    print(f"\n{title}:")
    print("-" * 50)

    for i, v in enumerate(vectors):
        # Format complex numbers for readability
        formatted_components = []
        for component in v:
            if np.iscomplexobj(component):
                # Format complex numbers with proper formatting
                real_part = f"{component.real:.4f}".rstrip('0').rstrip('.')
                imag_part = f"{abs(component.imag):.4f}".rstrip('0').rstrip('.')

                if component.real != 0 and component.imag != 0:
                    sign = '+' if component.imag >= 0 else '-'
```

```python
                formatted = f"{real_part} {sign} {imag_part}i"
            elif component.imag != 0:
                sign = '' if component.imag >= 0 else '-'
                formatted = f"{sign}{imag_part}i"
            else:
                formatted = f"{real_part}"
        else:
            # Format real numbers
            formatted = f"{component:.4f}".rstrip('0').rstrip('.')
            formatted = formatted if formatted != '' else '0'

        formatted_components.append(formatted)

    # Create the vector representation
    vector_str = "[" + ", ".join(formatted_components) + "]"
    print(f"Vector {i}: {vector_str}")
```

[88]:
```python
# We define an arbitrary set of vectors
vecs=np.array([[1+1j, 2+2j,3+3j],
               [4+4j, 5+5j,6+6j],
               [7+7j,8+8j,9+9j]],)

# Apply the gram_schmidt function
orthonormVecs=gram_schmidt(vecs)

# Print the vectors
print_vectors(orthonormVecs, title="Vectors")

# Prove the orthonormality properties
check_orthonormality(orthonormVecs)
```

```
Vectors:
--------------------------------------------------
Vector 0: [-0.5752 - 0.5752i, -0.039 - 0.039i, 0.4094 + 0.4094i]
Vector 1: [-0.5389 - 0.5389i, -0.0415 - 0.0415i, 0.456 + 0.456i]
Vector 2: [0.3554 + 0.3554i, 0.4061 + 0.4061i, 0.4569 + 0.4569i]
Vectors 0 and 1 are not orthogonal: inner product = (0.9965117667838046+0j)
```

[88]: False

[119]:
```python
# We define an arbitrary set of vectors
vecs=np.array([[0+1j, 2+2j,3+3j],
               [4+4j, 0+5j,6+6j],
               [7+7j,8+8j,9+9j]],)

# Apply the gram_schmidt function
orthonormVecs=gram_schmidt(vecs)
```

```
# Print the vectors
print_vectors(orthonormVecs, title="Vectors")

# Prove the orthonormality properties
check_orthonormality(orthonormVecs)
```

```
Vectors:
--------------------------------------------------
Vector 0: [-0.735 - 0.3694i, 0.1404 - 0.0369i, 0.4469 + 0.3201i]
Vector 1: [0.2481 - 0.0958i, -0.8056 - 0.0074i, 0.5231 + 0.0811i]
Vector 2: [0.3554 + 0.3554i, 0.4061 + 0.4061i, 0.4569 + 0.4569i]
```

[119]: True

[90]:
```
# We define an arbitrary set of vectors
vecs=np.array([[0+1j, 2+2j],
              [4+4j, 0+5j]])

# Apply the gram_schmidt function
orthonormVecs=gram_schmidt(vecs)

# Print the vectors
print_vectors(orthonormVecs, title="Vectors")

# Prove the orthonormality properties
check_orthonormality(orthonormVecs)
```

```
Vectors:
--------------------------------------------------
Vector 0: [-0.6321 + 0.1975i, 0.6637 + 0.3477i]
Vector 1: [0.5298 + 0.5298i, 0.6623i]
```

[90]: True

[120]:
```
# We define an arbitrary set of vectors
vecs=np.array([[1+1j, 2+2j],
              [1+1j, 2+3j]])

# Apply the gram_schmidt function
orthonormVecs=gram_schmidt(vecs)

# Print the vectors
print_vectors(orthonormVecs, title="Vectors")

# Prove the orthonormality properties
```

```
check_orthonormality(orthonormVecs)
```

```
Vectors:
--------------------------------------------------
Vector 0: [0.1826 + 0.9129i, 0 - 0.3651i]
Vector 1: [0.2582 + 0.2582i, 0.5164 + 0.7746i]
```

[120]:
```
True
```

The Gram-Schmidt procedure is define as folows,

$$|v_1\rangle = \frac{|w_1\rangle}{\sqrt{\langle w_1|w_1\rangle}},$$
$$|v_{k+1}\rangle = \frac{|w_{k+1}\rangle - \sum_{i=1}^{k} \langle v_i|w_{k+1}\rangle |v_i\rangle}{\sqrt{\langle\cdot|\cdot\rangle}}$$

This procedures starts by normalizing an arbitrary vector. Then it substracts the projection of the first vetor onto the next arbitrary vector and normalize the new second vector. Then it repeats this substraction of projections and normalization for the rest of the arbitrary vectors.

1.5 Idemptonece of Pauli Matrices

[121]:
```
def check_matrix_square_identity(A):
    """
    Check if a matrix A satisfies A^2 = I (identity matrix).

    Parameters:
    A (numpy.ndarray): Input matrix

    Returns:
    bool: True if A^2 = I, False otherwise
    """
    # Calculate A^2
    A_squared = np.dot(A, A)

    # Create identity matrix of the same size as A
    I = np.eye(A.shape[0])

    # Check if A^2 is exactly equal to I
    return np.array_equal(A_squared, I)
```

[93]:
```
print(check_matrix_square_identity(sx))
print(check_matrix_square_identity(sy))
print(check_matrix_square_identity(sz))
```

```
True
True
True
```

Pauli matrices are not idempotent because raising them to the second power results in the identity matrix rather than the original matrix.

1.6 Normal Operators

[122]:
```
def commutes_with_adjoint(A):
    """
    Check if a matrix A commutes with its adjoint (A*A = A*A).

    Parameters:
    A (numpy.ndarray): Input matrix (can be real or complex)

    Returns:
    bool: True if A commutes with its adjoint, False otherwise
    """
    # Calculate the adjoint (conjugate transpose) of A
    A_adjoint = A.conj().T

    # Calculate A*A and A*A
    A_times_adjoint = np.dot(A, A_adjoint)
    adjoint_times_A = np.dot(A_adjoint, A)

    # Check if the two products are exactly equal
    return np.array_equal(A_times_adjoint, adjoint_times_A)

def print_normality(A):
    """
    Check if a matrix A is normal (commutes with its adjoint) and print a␣
 ↪formatted message.

    Parameters:
    A (numpy.ndarray): Input matrix (can be real or complex)
    """
    # Check if the two products are exactly equal
    is_normal = commutes_with_adjoint(A)

    # Format the matrix for beautiful printing
    def format_matrix(matrix):
        rows = []
        for row in matrix:
            elements = []
            for element in row:
                if np.iscomplexobj(element):
                    # Format complex numbers
                    real_part = f"{element.real:.4f}".rstrip('0').rstrip('.')
                    imag_part = f"{abs(element.imag):.4f}".rstrip('0').rstrip('.
 ↪')
```

```
                if element.real != 0 and element.imag != 0:
                    sign = '+' if element.imag >= 0 else '-'
                    formatted = f"{real_part} {sign} {imag_part}i"
                elif element.imag != 0:
                    sign = '' if element.imag >= 0 else '-'
                    formatted = f"{sign}{imag_part}i"
                else:
                    formatted = f"{real_part}"
            else:
                # Format real numbers
                formatted = f"{element:.4f}".rstrip('0').rstrip('.')
                formatted = formatted if formatted != '' else '0'

            elements.append(formatted)
        rows.append("[" + ", ".join(elements) + "]")
    return "[" + ", ".join(rows) + "]"

# Create the message
if is_normal:
    message = f"The matrix {format_matrix(A)} is normal"
else:
    message = f"The matrix {format_matrix(A)} does not fulfill the normal␣
↪conditions"

print(message)
```

[112]:

```
# Define an hermitian matrix
HM= np.array([[2, 0+1j],
              [0-1j, 3]]);

# Define a non-normal matrix
B = np.array([[1, 0],
              [1, 1]]);

# Print the results

print_normality(HM)
print_normality(sx)
print_normality(sy)
print_normality(sz)
print_normality(B)
```

```
The matrix [[2, 1i], [-1i, 3]] is normal
The matrix [[0, 1], [1, 0]] is normal
The matrix [[0, -1i], [1i, 0]] is normal
The matrix [[1, 0], [0, -1]] is normal
The matrix [[1, 0], [1, 1]] does not fulfill the normal conditions
```

1.6.1 About Hermitian and Unitary operators

Since Hermitian and Unitary operators can be represented as hermitian and unitary matrices, we know the following properties of those types of matrices:

Hermitian matrices (\$ A = A^†\$) Since the adjoint is equal to the original matrix, when proving the normality condition is the same as raising the matrix to the second power, hence it commutes.

1.6.2 Unitary matrices ($AA^{\dagger} = I \wedge A^{\dagger}A = I$)

Here, from the definition of the unitary matrix we can see that, the matrix commutes with its adjoint, but because both operations are defined to be equal the Identity matrix.

Finally, the fact tat the hermtian and unitary matrices are a subset, this implies that not all matrices fullfill the normality conditions.