1 Methods & Numerical implementation

1.1 Computational basis

In order to properly build our quantum ciruit, we need to initialize the computational basis as shown in the theory section. This is easily implemented using the arrays in the NumPy library, and we can build any n-qubit computational basis by using the kronecker product on a single qubit basis - much like we do when we write out such multi-qubit states analytically.
The following code snippet shows the computatinonal basis for a two-qubit system, as presented in the theory section:

```
    import  numpy as np
    def init_basis():
        q0 = np.array([1, 0])
        q1 = np.array([0, 1])

        return q0, q1

    def init_2basis():
        q0, q1 = init_basis()
        q00 = np.kron(q0, q0)
        q01 = np.kron(q0, q1)
        q10 = np.kron(q1, q0)
        q11 = np.kron(q1, q1)

        return q00, q01, q10, q11
```

This illustrates the basis concept of building the computational basis needed to carry out the quantum computations that we are to implement in the VQE algorithm.

1.2 Measurements

The way we implement the "randomness" of a quantum measurement, and how to properly find the expectation value of a quantum operator, is by use of the random module in NumPy. As an example, this is how one can implement a measurement of the expectation value of the Z operator on a single qubit (prepared in the state $|0\rangle$):

```
import numpy as np

wf = np.array([1, 0])
prob_wf = np.abs(wf)**2
n_measurements = 1000
expectation_value = 0

for i in range(n_measurements):
    measurement = np.random.choice([0, 1], p=prob_wf)
    if measurement == 0:
        expectation_value += 1
    else:
        expectation_value -= 1

expectation_value /= n_measurements
```

This is a simple example, but it shows how we can implement a measurement of the expectation value of the Pauli-Z operator on a single qubit. This would also work for a qubit prepared in a superposition state. How to implement this for more "advanced" measurements can be found in the source code available on the GitHub repository, but we will closely the procedure presented in the lecture notes in FYS5419, see **FYS5419-Lecture-notes**.

1.3 Quantum Gates

When we want to apply the quantum gates to our qubits, we can do this by matrix multiplication. This is also easily done for multi-qubit systems, by use of the kronecker product in the NumPy library. As an example, this is how one can implement the Hadamard gate on a two-qubit system (prepared in the state $|00\rangle$) in Python:

```python
import numpy as np

def Hadamard():
    H = 1/np.sqrt(2)*np.array([[1, 1], [1, -1]])
    return H

def apply_gate(gate, qubits):
    H = np.kron(Hadamard(), Hadamard())
    qbit = np.array([1,0])
    2qbit = np.kron(qbit, qbit)
    new_state = H @ 2qbit

    return new_state
```

Here we show how easily one can extend the single qubit operators to multi-qubit systems, by usage of the kronecker product (tensor product). And this generalizes to any number of qubits.

1.4 Hamiltonians

When we want to implement a Hamiltonian, we do this using the Pauli basis, as explained in the theory section. The following code snippet illustrates how one can implement the two-qubit Hamiltonian shown in the theory section:

```python
import numpy as np

def Hamiltonian():
    epsilon_list = [eps1, eps2, eps3, eps4]
    Hx = 1.0; Hz = 1.0;
    X = pauli_x()
    Z = pauli_z()

    H0 = np.diag(epsilon_list)
    x_term = Hx * np.kron(X, X)
    z_term = Hz * np.kron(Z, Z)
    H = H0 + x_term + z_term

    return H
```

This is the basic concept we will use to implement all the Hamiltonians needed in our report. We will start by building our code for the simple one-qubit system, before going to the two-qubit system. The full implementation will then be extended to the Lipkin model Hamiltonian.

1.5 Variational Quantum Eigensolver

The full VQE implementations can be found in the source code available on the GitHub repository. Here we will present the algorithm as pseudo code, to give a brief overview of the numerical implementation of the VQE algorithm:

Algorithm 1 VQE algorithm

```
Initialize $\theta$ randomly
Initialize the quantum circuit
Initialize the Hamiltonian
while not converged do
    → Apply the quantum circuit to the initial state
    → Measure the expectation value of the Hamiltonian
    → Find the gradients of the paramters w.r.t. the expectation value
    → Update the parameters $\theta$
end while
```

These are the algorithmic steps we've followed when developing our code for the VQE, as can readily be seen on GitHub.

1.6 Entanglement

As explained in the theory section, entanglement is a very important concept in our quantum system. When we are to make a measurement of the entanglement, we will make use of the Von Neumann entropy (see section (**??**)), and this can be done numerically by use of the NumPy library as shown in the following code snippet:

```python
import numpy as np
H = some_hamiltonian()
N = 2
eigval, eigvecs = np.linalg.eigh(H)

def density_matrix(eigenvector):
    return np.outer(eigenvector.T.conj(), eigenvector.T)

den_matrix = density_matrix(eigvecs[0])
den_b = np.trace(den_matrix.reshape(N,N,N,N), axis1=0, axis2=2)
prob_b = np.linalg.eigvalsh(den_a)
von_neumann_entropy = -np.sum(prob_a * np.log2(prob_a))
```

This is a simple example of how one can calculate the Von Neumann entropy of subsystem B for a 2-qubit system given some Hamiltonian that can be diagonalized. The procedure can also be generalized to any number of qubits, but in this report, we will only consider the 2-qubit system. The reason we re-shape the matrix is so that we can properly "trace out" the correct axises that correspond to subsystem A (or B).

1.7 Wavefunction ansatz

As explained in the theory section, it is important that we properly construct "good" wavefunction ansatzses for our quantum circuits in the VQE algorithm. These ansatz should be properly entangled, so that they capture the physical properties of the system. The way one would construct a simple, entangled wavefunction ansatz with a variational parameter θ is shown in the following code snippet

```python
def ansatz(theta):
    CNOT = np.array([[1,0,0,0], [0,1,0,0], [0,0,0,1], [0,0,1,0]])
    Y = np.array([[0, -1j], [1j, 0]])
    R_y = np.cos(theta/2) * np.eye(2) - 1j * np.sin(theta/2) * Y
    q00 = np.array([1,0,0,0])

    return CNOT @ R_y @ q00
```

Here we employ first the rotation about the Y-axis (on the bloch sphere), by usage of the Y-rotation gate, which is built from the pauli Y-gate σ_y and the identity, $R_y = \cos(\theta/2)\mathbf{1} - i\sin(\theta/2)\sigma_y$, before we add entanglement to our state by the controlled NOT gate (CNOT), with the first qubit being the control bit, second qubit the target bit.
This simple ansatz can be used in a system with some entanglement, but it is still a very simple ansatz - and more complex systems will require more complex ansatzses, as we shall see in later sections.