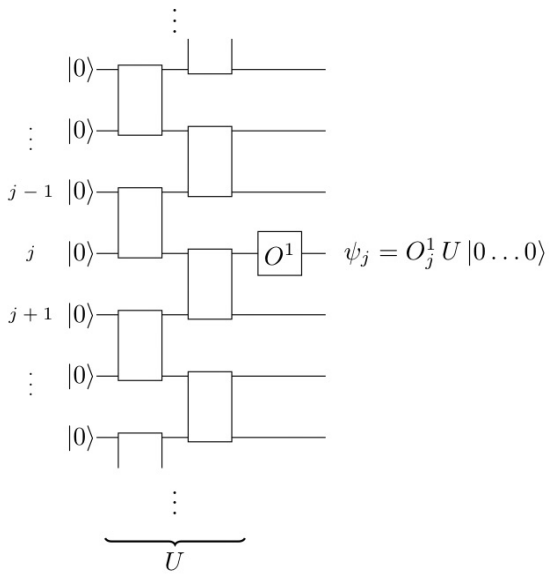
Exercise 12.1 (Light cone pattern of dynamical correlations in brick wall quantum circuits)

A common quantum circuit layout is a "brick wall" arrangement of two-qubit gates, as illustrated on the right. One can interpret such a layout as discretization of a quantum time evolution step, $U \approx \mathrm{e}^{-iH\Delta t}$ (cf. the TEBD algorithm).



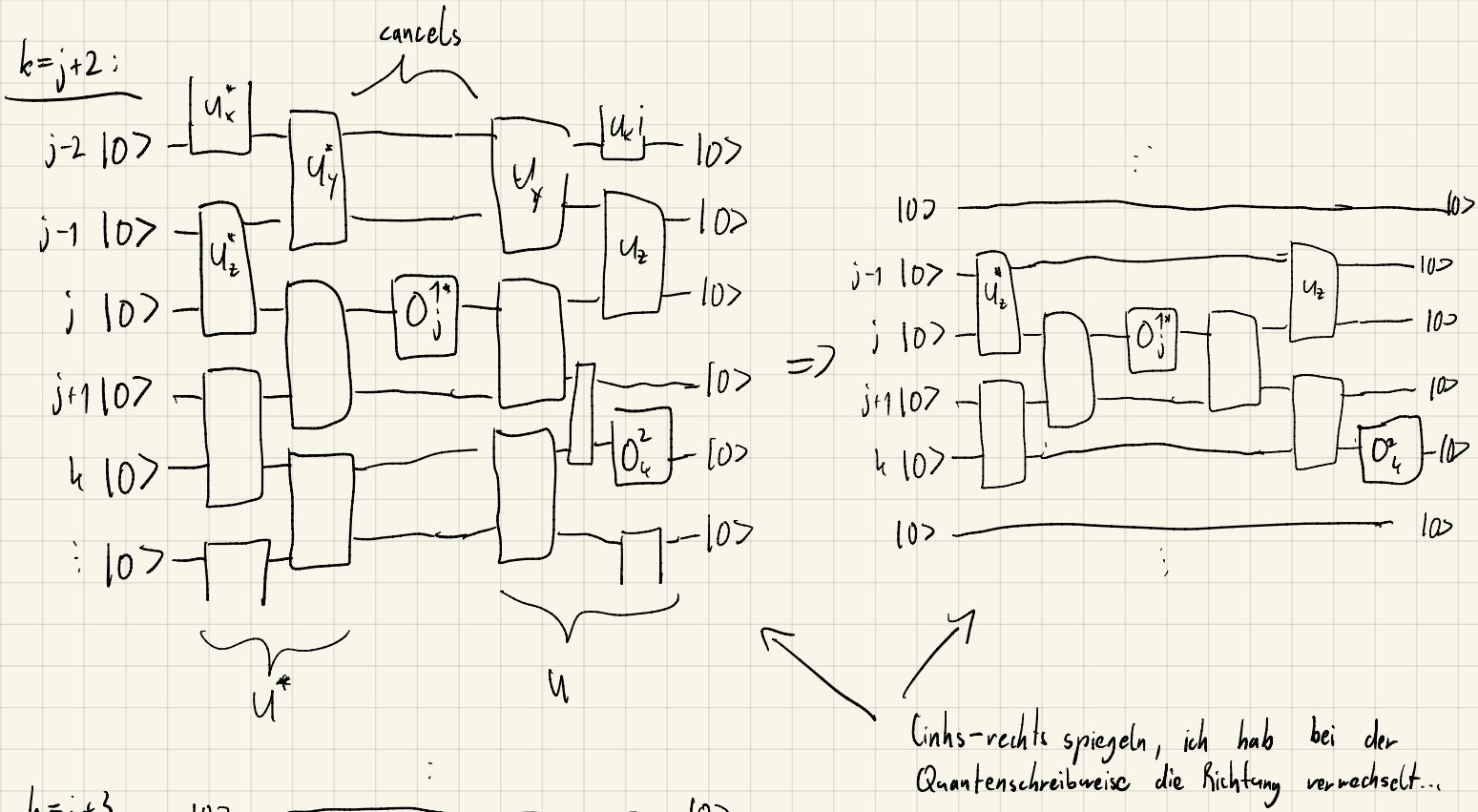
Our goal here is to compute the so-called *dynamical correlation function* between the j-th and k-th qubit ($k \geq j$). This is achieved via the following construction: The initial state $|0 \ldots 0\rangle$ time-evolves for a step Δt; we then apply a single-qubit gate O^1 to the j-qubit. O^1 is also assumed to be Hermitian, like for example one of the Pauli matrices, and plays the role of an observable. We denote the overall output state of this protocol by ψ_j, as illustrated on the right. (The total number of qubits is assumed to be large enough such that boundary effects are not relevant. The following considerations are valid even if all the two-qubit gates are different.)

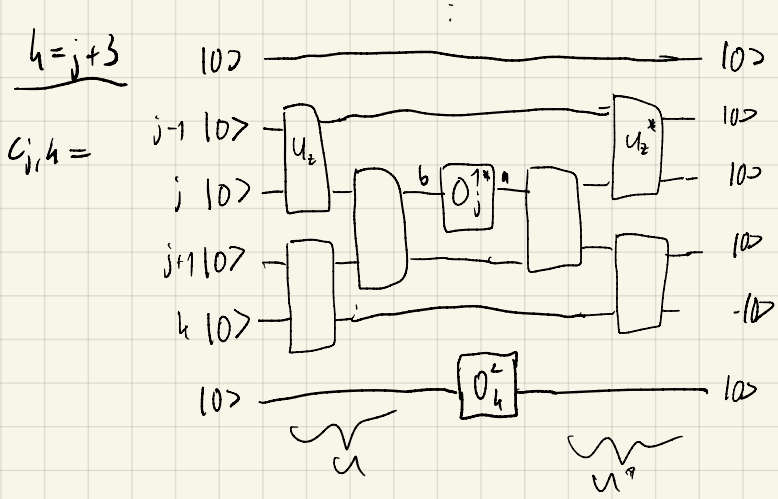
Regarding the k-th qubit, we again start from the state $|0 \ldots 0\rangle$, then first apply another gate O^2 to the k-qubit, and finally perform a time step. The overall output of this protocol is thus $\chi_k = U O_k^2 |0 \ldots 0\rangle$.

The dynamical correlation function[3] is now defined as

$$c_{j,k} = \langle \psi_j, \chi_k \rangle .$$

(a) Draw the diagram for evaluating $c_{j,k}$ for the cases $k = j + 2$ and $k = j + 3$, and simplify it as much as possible by canceling two-qubit gates (i.e., a two-qubit gate V directly followed by $V^\dagger$).



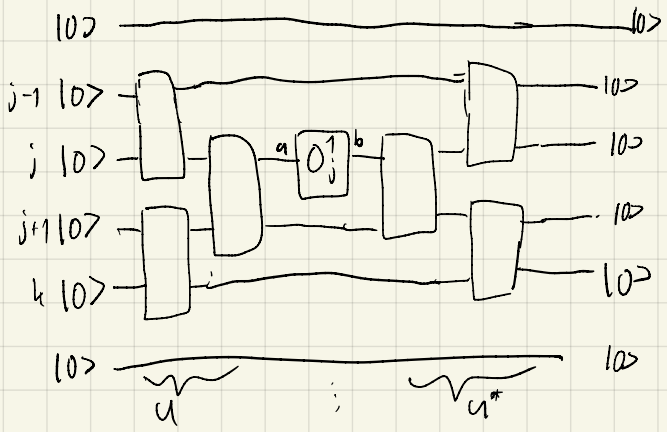


(b) For $k = j + 3$, compute the so-called connected correlation function

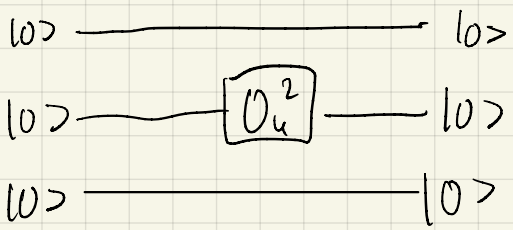
$$c^{\text{conn}}_{j,k} = c_{j,k} - \langle 0 \dots 0 | U^\dagger O^1_j U | 0 \dots 0 \rangle \cdot \langle 0 \dots 0 | O^2_k | 0 \dots 0 \rangle .$$

Does your result change when considering $k \geq j + 3$ in general?

$\langle 0 \dots 0 | U^\dagger O_j^1 U | 0 \dots 0 \rangle =$

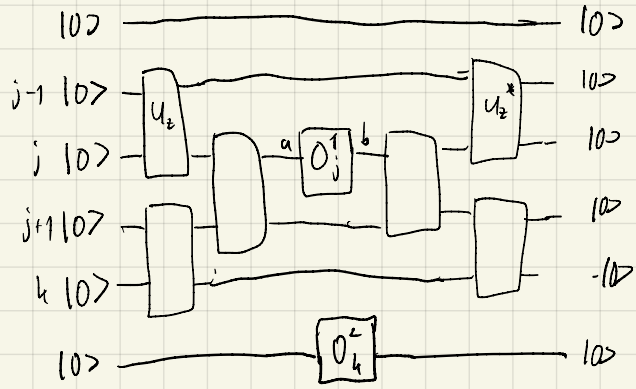


$\langle 0 \dots 0 | O_k^2 | 0 \dots 0 \rangle =$



$\langle 0 \dots | U^\dagger O_j^1 U | 0 \dots \rangle \cdot \langle 0 \dots | O_k^2 | 0 \dots \rangle$

$=$



$= c_{jlh}$ (weil hermitian ist)

$O_j^1 = O_j^{1*}$

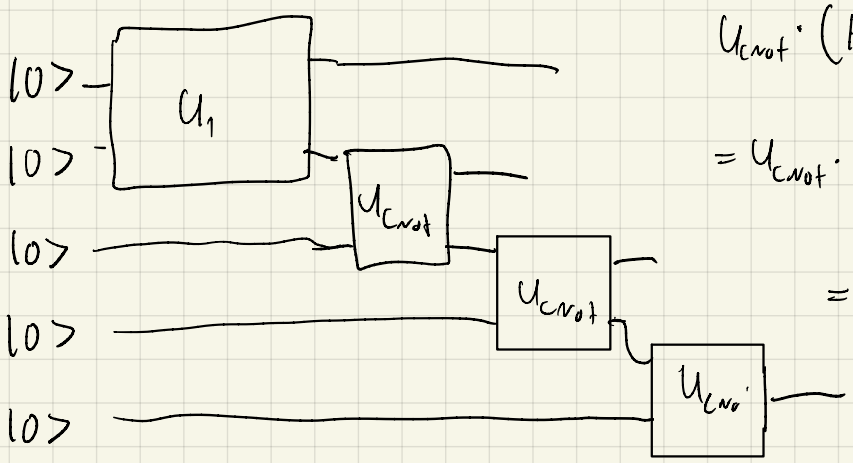
(beachte Beschriftung der Indexe hier und oben)

$\Rightarrow \quad c^{\text{conn}}_{jlh} = c_{jlh} - c_{jlh} = 0$

$\Rightarrow$ ändert sich auch bei $k > j+2$ nicht da nur weitere sich rauskürzende Verbindungen dazwischen entstehen,

(b) Calculate (with "pen and paper") the output quantum state when setting $U^1 = U_{\text{CNOT}} \cdot (H \otimes I)$, where H is the Hadamard gate, and $U^j = U_{\text{CNOT}}$ for all $j \geq 2$. Then use this state to test your implementation from (a): calculate the MPS representation for these gates, convert it to a statevector via `as_vector()`, and ensure this vector agrees with your analytical result.

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \qquad H \otimes I = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix} \qquad U_{CNot} = \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & 0 & 1 \\ & & 1 & 0 \end{pmatrix}$$



$$U_{CNot} \cdot (H \otimes I) \cdot (|0\rangle \otimes |0\rangle)$$

$$= U_{CNot} \cdot (H \otimes I) \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$= U_{CNot} \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

next step: $\left(\frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} \otimes |0\rangle\right) \cdot (I_2 \otimes U_{CNot})$

↓ same format every step

$$\frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 1 \\ 0 \end{pmatrix} \cdot (I \otimes \cdots \otimes U_{CNot}) = \left.\begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}\right\} 2^d \text{ Einträge}$$

exercise12.2_template

July 20, 2022

1 Shallow quantum circuits

[1]:
```
import numpy as np
```

1.1 Utility functions

[2]:
```
def crandn(size):
    """
    Draw random samples from the standard complex normal (Gaussian)␣
 ↪distribution.
    """
    # 1/sqrt(2) is a normalization factor
    return (np.random.normal(size=size) + 1j*np.random.normal(size=size)) / np.
 ↪sqrt(2)
```

[3]:
```
def retained_bond_indices(s, tol):
    """
    Indices of retained singular values based on given tolerance.
    """
    w = np.linalg.norm(s)
    if w == 0:
        return np.array([], dtype=int)
    # normalized squares
    s = (s / w)**2
    # accumulate values from smallest to largest
    sort_idx = np.argsort(s)
    s[sort_idx] = np.cumsum(s[sort_idx])
    return np.where(s > tol)[0]
```

[4]:
```
def split_matrix_svd(A, tol):
    """
    Split a matrix by singular value decomposition,
    and truncate small singular values based on tolerance.
    """
    assert A.ndim == 2
    u, s, v = np.linalg.svd(A, full_matrices=False)
    # truncate small singular values
```

```
    idx = retained_bond_indices(s, tol)
    u = u[:, idx]
    v = v[idx, :]
    s = s[idx]
    return u, s, v
```

1.2 MPS class and related utility functions

[5]:
```
def local_orthonormalize_left_qr(A, Anext):
    """
    Left-orthonormalize a MPS tensor `A` by a QR decomposition,
    and update tensor at next site.
    """
    # perform QR decomposition and replace A by reshaped Q matrix
    s = A.shape
    assert len(s) == 3
    Q, R = np.linalg.qr(np.reshape(A, (s[0]*s[1], s[2])), mode='reduced')
    A = np.reshape(Q, (s[0], s[1], Q.shape[1]))
    # update Anext tensor: multiply with R from left
    Anext = np.transpose(np.tensordot(R, Anext, (1, 1)), (1, 0, 2))
    return A, Anext
```

[6]:
```
def local_orthonormalize_right_qr(A, Aprev):
    """
    Right-orthonormalize a MPS tensor `A` by a QR decomposition,
    and update tensor at previous site.
    """
    # flip left and right virtual bond dimensions
    A = np.transpose(A, (0, 2, 1))
    # perform QR decomposition and replace A by reshaped Q matrix
    s = A.shape
    assert len(s) == 3
    Q, R = np.linalg.qr(np.reshape(A, (s[0]*s[1], s[2])), mode='reduced')
    A = np.transpose(np.reshape(Q, (s[0], s[1], Q.shape[1])), (0, 2, 1))
    # update Aprev tensor: multiply with R from right
    Aprev = np.tensordot(Aprev, R, (2, 1))
    return A, Aprev
```

[7]:
```
def merge_mps_tensor_pair(A0, A1):
    """
    Merge two neighboring MPS tensors.
    """
    A = np.tensordot(A0, A1, (2, 1))
    # pair original physical dimensions of A0 and A1
    A = A.transpose((0, 2, 1, 3))
    # combine original physical dimensions
```

```
    A = A.reshape((A.shape[0]*A.shape[1], A.shape[2], A.shape[3]))
    return A
```

[8]:
```
class MPS(object):
    """
    Matrix product state (MPS) class.

    The i-th MPS tensor has dimension `[d, D[i], D[i+1]]` with `d` the physical
    dimension at each site and `D` the list of virtual bond dimensions.
    """

    def __init__(self, d, D, fill='zero'):
        """
        Create a matrix product state.
        """
        self.d = d
        # leading and trailing bond dimensions must agree (typically 1)
        assert D[0] == D[-1]
        if fill == 'zero':
            self.A = [np.zeros((d, D[i], D[i+1])) for i in range(len(D)-1)]
        elif fill == 'random real':
            # random real entries
            self.A = [np.random.normal(size=(d, D[i], D[i+1])) / np.
 ↪sqrt(d*D[i]*D[i+1]) for i in range(len(D)-1)]
        elif fill == 'random complex':
            # random complex entries
            self.A = [crandn(size=(d, D[i], D[i+1])) / np.sqrt(d*D[i]*D[i+1])␣
 ↪for i in range(len(D)-1)]
        else:
            raise ValueError('fill = {} invalid.'.format(fill))

    @classmethod
    def from_tensors(cls, Alist):
        """
        Construct a MPS from a list of tensors.
        """
        # create a MPS with dummy tensors
        s = cls(2, (len(Alist) + 1) * [1])
        # assign the actual tensors from `Alist`
        s.A = [np.array(A) for A in Alist]
        s.d = s.A[0].shape[0]
        # consistency checks
        for j in range(len(s.A)):
            assert s.A[j].ndim == 3, "Each MPS tensor must be of degree 3."
            assert s.d == s.A[j].shape[0], "Physical dimension not consistent␣
 ↪across MPS tensors."
```

```
            assert s.A[j].shape[2] == s.A[(j+1) % len(s.A)].shape[1],␣
↪"Incompatible virtual bond dimensions."
        return s

    @property
    def local_dim(self):
        """Local (physical) dimension at each lattice site."""
        return self.d

    @property
    def nsites(self):
        """Number of lattice sites."""
        return len(self.A)

    @property
    def bond_dims(self):
        """Virtual bond dimensions."""
        if len(self.A) == 0:
            return []
        else:
            D = [self.A[i].shape[1] for i in range(len(self.A))]
            D.append(self.A[-1].shape[2])
            return D

    @property
    def dtype(self):
        """Data type of tensor entries."""
        return self.A[0].dtype

    def orthonormalize(self, mode='left'):
        """Left- or right-orthonormalize the MPS using QR decompositions."""
        if len(self.A) == 0:
            return

        if mode == 'left':
            for i in range(len(self.A) - 1):
                self.A[i], self.A[i+1] = local_orthonormalize_left_qr(self.
↪A[i], self.A[i+1])
            # last tensor
            self.A[-1], T = local_orthonormalize_left_qr(self.A[-1], np.
↪array([[[1.0]]]))
            # normalization factor (real-valued since diagonal of R matrix is␣
↪real)
            assert T.shape == (1, 1, 1)
            nrm = T[0, 0, 0].real
            if nrm < 0:
```

```
                # flip sign such that normalization factor is always␣
↪non-negative
                self.A[-1] = -self.A[-1]
                nrm = -nrm
            return nrm
        elif mode == 'right':
            for i in reversed(range(1, len(self.A))):
                self.A[i], self.A[i-1] = local_orthonormalize_right_qr(self.
↪A[i], self.A[i-1])
            # first tensor
            self.A[0], T = local_orthonormalize_right_qr(self.A[0], np.
↪array([[[1.0]]]))
            # normalization factor (real-valued since diagonal of R matrix is␣
↪real)
            assert T.shape == (1, 1, 1)
            nrm = T[0, 0, 0].real
            if nrm < 0:
                # flip sign such that normalization factor is always␣
↪non-negative
                self.A[0] = -self.A[0]
                nrm = -nrm
            return nrm
        else:
            raise ValueError('mode = {} invalid; must be "left" or "right".'.
↪format(mode))

    def as_vector(self):
        """Merge all tensors to obtain the vector representation on the full␣
↪Hilbert space."""
        psi = self.A[0]
        for i in range(1, len(self.A)):
            psi = merge_mps_tensor_pair(psi, self.A[i])
        assert psi.ndim == 3
        # contract leftmost and rightmost virtual bond (has no influence if␣
↪these virtual bond dimensions are 1)
        psi = np.trace(psi, axis1=1, axis2=2)
        return psi
```

[9]:
```
def split_mps_tensor(A, d0, d1, svd_distr, tol=0):
    """
    Split a MPS tensor with dimension `d0*d1 x D0 x D2` into two MPS tensors
    with dimensions `d0 x D0 x D1` and `d1 x D1 x D2`, respectively.
    """
    assert A.ndim == 3
    assert d0 * d1 == A.shape[0], 'physical dimension of MPS tensor must be␣
↪equal to d0 * d1'
```

```
    # reshape as matrix and split by SVD
    A = np.transpose(np.reshape(A, (d0, d1, A.shape[1], A.shape[2])), (0, 2, 1,␣
↪3))
    s = A.shape
    A0, sigma, A1 = split_matrix_svd(A.reshape((s[0]*s[1], s[2]*s[3])), tol)
    A0.shape = (s[0], s[1], len(sigma))
    A1.shape = (len(sigma), s[2], s[3])
    # use broadcasting to distribute singular values
    if svd_distr == 'left':
        A0 = A0 * sigma
    elif svd_distr == 'right':
        A1 = A1 * sigma[:, None, None]
    elif svd_distr == 'sqrt':
        s = np.sqrt(sigma)
        A0 = A0 * s
        A1 = A1 * s[:, None, None]
    else:
        raise ValueError('svd_distr parameter must be "left", "right" or "sqrt".
↪')
    # move physical dimension to the front
    A1 = A1.transpose((1, 0, 2))
    return A0, A1
```

[10]:
```
def is_left_orthonormal(A):
    """
    Test whether a MPS tensor `A` is left-orthonormal.
    """
    s = A.shape
    assert len(s) == 3
    A = np.reshape(A, (s[0]*s[1], s[2]))
    return np.allclose(A.conj().T @ A, np.identity(s[2]))
```

[11]:
```
def is_right_orthonormal(A):
    """
    Test whether a MPS tensor `A` is right-orthonormal.
    """
    # call `is_left_orthonormal` with flipped left and right virtual bond␣
↪dimensions
    return is_left_orthonormal(np.transpose(A, (0, 2, 1)))
```

1.3 Shifted sequence of two-qubit gates

[12]:
```
def shifted_gate_sequence_to_mps(Ulist: [np.ndarray]):
    """
    Represent the output of a quantum circuit consisting of
    a shifted sequence of two-qubit gates as MPS.
```

```
          ____
|0> ---|    |-----------------------------
      | U1 |    ____
|0> ---|____|---|    |---------------------
               | U2 |                   .
|0> ------------|____|---                .
                          ...      ____   .
                               ---|    |---
                                  | U* |
|0> -------------------------------|____|---

"""
Alist = []

for U in Ulist:
    U = U.reshape((2, 2, 2, 2))
    U = U.transpose((0, 2, 1, 3))
    Alist.append(U[:,:,:,0])

Alist[0] = Alist[0][:, 0, :]
Alist[0] = Alist[0][:, None, :]
Alist.append(np.identity(2)[:, :, None])


return MPS.from_tensors(Alist)
```

[13]:
```
# Hadamard gate
H = np.array([[1., 1.], [1., -1.]]) / np.sqrt(2)
H
```

[13]:
```
array([[ 0.70710678,  0.70710678],
       [ 0.70710678, -0.70710678]])
```

[14]:
```
# CNOT gate
Ucnot = np.identity(4)[[0, 1, 3, 2]]
Ucnot
```

[14]:
```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 0., 1.],
       [0., 0., 1., 0.]])
```

[15]:
```
d = 5
Ulist = [Ucnot @ np.kron(H, np.identity(2)) if j == 0 else Ucnot for j in␣
 ↪range(d-1)]
```

[16]:
```
 = shifted_gate_sequence_to_mps(Ulist)
```

[17]:
```
# TODO: compare this output with analytical prediction
reference_solution = np.array([0] * 32,dtype=float)
reference_solution[0] = (1.0 / np.sqrt(2))
reference_solution[-1] = (1.0 / np.sqrt(2))
np.allclose(reference_solution,  .as_vector())
```

[17]: True

1.3.1 Example: sequence of random gates (for additional testing)

[18]:
```
# for reference calculation
def apply_two_qubit_gate(psi, d, U, i, j):
    """
    Apply the two-qubit gate `U` (acting on qubits i, j) to state vector `psi`.
    `d` is the overall number of qubits.
    """
    assert 0 <= i < j < d
    assert len(psi) == 2**d
    # isolate dimensions corresponding to the qubits the gate acts on
    psi = np.reshape(psi, (2**i, 2, 2**(j-i-1), 2, 2**(d-j-1)))
    # reshape gate into a tensor of degree 4
    U = np.reshape(U, (2, 2, 2, 2))
    # actually apply gate (last argument determines dimension ordering of␣
 ↪output)
    psi = np.einsum(U, (1, 5, 2, 4), psi, (0, 2, 3, 4, 6), (0, 1, 3, 5, 6))
    # flatten back to a vector
    psi = np.reshape(psi, -1)
    return psi
```

[19]:
```
d = 5
# random unitary gates
Ulist = [np.linalg.qr(crandn((4, 4)))[0] for j in range(d-1)]
```

[20]:
```
# reference calculation
# start with unit vector corresponding to |00...0>
 = np.zeros(2**d)
[0] = 1
# apply gates
for j in range(d-1):
     = apply_two_qubit_gate( , d, Ulist[j], j, j+1)
```

[21]:
```
# should still be normalized
np.linalg.norm( )
```

[21]:
```
1.0
```

[22]:
```
# show entries

```

[22]:
```
array([-0.2843864 +0.00845899j,  0.38910846-0.05332498j,
        0.24797006-0.06643209j,  0.11794898+0.5119437j ,
        0.01382717-0.0648759j ,  0.01918302+0.09667098j,
        0.0614047 +0.10925018j, -0.14979523+0.03367667j,
        0.0082177 -0.04209684j, -0.00657993+0.09430536j,
       -0.0466889 +0.11962071j, -0.12644896-0.02596463j,
       -0.02354462-0.08729502j, -0.01866348+0.08121519j,
       -0.06624863-0.09147729j, -0.05773502+0.01112094j,
       -0.16917242+0.0667214j ,  0.22054053-0.07899511j,
        0.08291643-0.00836553j,  0.13205805+0.25401798j,
       -0.03948706-0.13921742j,  0.02771821+0.14039363j,
        0.01046917-0.05066774j, -0.14427856+0.08849897j,
       -0.06615993+0.00871189j,  0.10073408-0.0148938j ,
        0.07397812+0.00868657j,  0.02270926+0.12732226j,
       -0.03381526-0.02117008j,  0.03257719+0.03787441j,
       -0.02309906+0.0224121j , -0.03332165+0.03626002j])
```

[23]:
```
 = shifted_gate_sequence_to_mps(Ulist)
.as_vector()
```

[23]:
```
array([-0.2843864 +0.00845899j,  0.38910846-0.05332498j,
        0.24797006-0.06643209j,  0.11794898+0.5119437j ,
        0.01382717-0.0648759j ,  0.01918302+0.09667098j,
        0.0614047 +0.10925018j, -0.14979523+0.03367667j,
        0.0082177 -0.04209684j, -0.00657993+0.09430536j,
       -0.0466889 +0.11962071j, -0.12644896-0.02596463j,
       -0.02354462-0.08729502j, -0.01866348+0.08121519j,
       -0.06624863-0.09147729j, -0.05773502+0.01112094j,
       -0.16917242+0.0667214j ,  0.22054053-0.07899511j,
        0.08291643-0.00836553j,  0.13205805+0.25401798j,
       -0.03948706-0.13921742j,  0.02771821+0.14039363j,
        0.01046917-0.05066774j, -0.14427856+0.08849897j,
       -0.06615993+0.00871189j,  0.10073408-0.0148938j ,
        0.07397812+0.00868657j,  0.02270926+0.12732226j,
       -0.03381526-0.02117008j,  0.03257719+0.03787441j,
       -0.02309906+0.0224121j , -0.03332165+0.03626002j])
```

[24]:
```
# compare
np.allclose( .as_vector(),  )
```

[24]:
```
True
```

1.4 Shifted double sequence of two-qubit gates

[25]:
```
def shifted_double_gate_sequence_to_mps(Ulist, Vlist):
    """
    Represent the output of a quantum circuit consisting of
    two shifted sequences of two-qubit gates as MPS.


             ____                ____
    |0> ---|    |--------------|    ␣
 ↪|-----------------------------------------------
           | U1 |    ____     | V1 |    ____
    |0> ---|____|---|    |---|____|---|    ␣
 ↪|--------------------------------------
                   | U2 |    ____     | V2 |
    |0> ------------|____|---|    |---|____|---                                  ␣
 ↪   .
                              | U3 |                                             ␣
 ↪   .
    |0> ---------------------|____|---        .                                  ␣
 ↪   .
                                                    .                 ____
                                                        .       ---|    ␣
 ↪|-------------
                                                                ____     | V* |     ____
                                                           ---|    |---|____|---|    ␣
 ↪|---
                                                              | U* |             | V*␣
 ↪|
    |0>␣
 ↪-------------------------------------------------|____|-------------|____|---

    """
    # TODO (voluntary): implement this function for part (c)
```

1.4.1 Example: sequence of random gates (for testing)

[26]:
```
d = 5
# random unitary gates
Ulist = [np.linalg.qr(crandn((4, 4)))[0] for j in range(d-1)]
Vlist = [np.linalg.qr(crandn((4, 4)))[0] for j in range(d-1)]
```

[27]:
```
# reference calculation
# start with unit vector corresponding to |00...0>
 = np.zeros(2**d)
[0] = 1
# apply gates
```

```
for j in range(d-1):
     = apply_two_qubit_gate( , d, Ulist[j], j, j+1)
for j in range(d-1):
     = apply_two_qubit_gate( , d, Vlist[j], j, j+1)
```

[28]:
```
# should still be normalized
np.linalg.norm( )
```

[28]: 1.0

[29]:
```
  = shifted_double_gate_sequence_to_mps(Ulist, Vlist)
```

[30]:
```
# compare
np.allclose( .as_vector(),  )
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Input In [30], in <cell line: 2>()
      1 # compare
----> 2 np.allclose( .as_vector(),  )

AttributeError: 'NoneType' object has no attribute 'as_vector'
```