(b) Assuming for simplicity that $n_\ell = n$ for $\ell = 1, \ldots, d$ and $D_\ell = D'_\ell = D$ for $\ell = 1, \ldots, d-1$, what is the asymptotic computational cost (in terms of powers of n, D and d) of this algorithm (when ignoring optimizations as described in Tutorial 4)?

Hint: First identify the most expensive step of the algorithm.

Most expensive step is the loop.

In every step:

1. contraction : $O(\underbrace{nD^2}_{\#\text{ of elem}} \cdot \underbrace{D}_{\text{sum}})$

2. contraction: $O(\underbrace{D^2}_{\#\text{of elem.}} \cdot \underbrace{nD}_{\text{sums}})$

$$\Rightarrow O(\underbrace{(d-2)}_{\substack{\text{first and last}\\ \text{contraction are less expensive}}} \cdot 2 \cdot nD^3) = O(d \cdot 2 \cdot nD^3) = \underline{\underline{O(d \cdot n \cdot D^3)}}$$

exercise6.2_template

June 8, 2022

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

[2]:
```python
def mps_vdot(Alist, Blist):
    """
    Compute the inner product of two tensors in MPS format, with the convention␣
↪that
    the complex conjugate of the tensor represented by the first argument is␣
↪used.

    The i-th MPS tensor Alist[i] is expected to have dimensions (n[i], Da[i],␣
↪Da[i+1]),
    and similarly Blist[i] must have dimensions                 (n[i], Db[i],␣
↪Db[i+1]),
    with `n` the list of logical dimensions and `Da`, `Db` the lists of virtual␣
↪bond dimensions.
    """

    l = len(Alist)
    R = np.tensordot(Blist[l-1], Alist[l-1].conjugate(), ([0], [0]))

    for i in range(l-2, -1, -1):
        R = np.tensordot(Alist[i].conjugate(), R, ([2], [2]))
        R = np.tensordot(Blist[i], R, ([0,2], [0,2]))
        R = R.transpose((0,2,1,3))

    return R[0][0][0][0]
```

[3]:
```python
def mps_to_full_tensor(Alist):
    """
    Construct the full tensor corresponding to the MPS tensors `Alist`.

    The i-th MPS tensor Alist[i] is expected to have dimensions (n[i], D[i],␣
↪D[i+1]),
    with `n` the list of logical dimensions and `D` the list of virtual bond␣
↪dimensions.
    """
    # consistency check: dummy singleton dimension
```

```
    assert Alist[0].ndim == 3 and Alist[0].shape[1] == 1
    # formally remove dummy singleton dimension
    T = np.reshape(Alist[0], (Alist[0].shape[0], Alist[0].shape[2]))
    # contract virtual bonds
    for i in range(1, len(Alist)):
        T = np.tensordot(T, Alist[i], axes=(-1, 1))
    # consistency check: trailing dummy singleton dimension
    assert T.shape[-1] == 1
    # formally remove trailing singleton dimension
    T = np.reshape(T, T.shape[:-1])
    return T
```

[4]:
```
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)
```

[5]:
```
# logical dimensions
n = [2, 5, 1, 4, 3]

# virtual bond dimensions (rather arbitrarily chosen)
Da = [1, 3, 4, 7, 6, 1]
Db = [1, 4, 9, 8, 5, 1]

# random MPS matrices (the scaling factor keeps the norm of the full tensor in␣
 ↪a reasonable range)
np.random.seed(42)
Alist = [0.4 * crandn((n[i], Da[i], Da[i+1])) for i in range(len(n))]
Blist = [0.4 * crandn((n[i], Db[i], Db[i+1])) for i in range(len(n))]
```

[6]:
```
len(Alist)
```

[6]:
```
5
```

[7]:
```
Alist[1].shape
```

[7]:
```
(5, 3, 4)
```

[8]:
```
# show entries of one of the MPS tensors, as illustration
Alist[1]
```

[8]:
```
array([[[ 0.06843727-0.01013313j, -0.54115737+0.44254806j,
         -0.48788044-0.74097581j, -0.15903893+0.23246913j],
```

```
   [-0.2864719 +0.02462063j,  0.08888257-0.08457205j,
    -0.25682799+0.02595387j, -0.39945981-0.56216938j],
   [ 0.41454807-0.06213259j, -0.06385918+0.10100669j,
     0.01909986+0.41801156j, -0.40297964-0.14658895j]],

  [[-0.15397469-0.22867652j,  0.03137365-0.14191832j,
    -0.32555015+0.25891482j,  0.10626345+0.09298486j],
   [-0.16988628-0.14983881j, -0.08250345+0.14517395j,
    -0.17018833+0.02745768j,  0.52390339+0.27397418j],
   [-0.00381759-0.1985706j , -0.29916583-0.09267685j,
     0.23265083-0.11090493j, -0.34530673-0.41394454j]],

  [[ 0.05907555+0.08375546j, -0.55427841+0.07383758j,
    -0.37566774+0.0014463j ,  0.05568077-0.06635126j],
   [ 0.20886989-0.4003273j ,  0.04847027-0.11897646j,
    -0.03271027-0.0969343j , -0.08516499-0.22691828j],
   [-0.41818917-0.04561849j, -0.20360269+0.11428284j,
    -0.13028832+0.53349394j,  0.29899932+0.04937806j]],

  [[ 0.09718993+0.07284625j, -0.49866306-0.02105648j,
     0.09166479-0.54271046j, -0.10891772-0.00749926j],
   [-0.19146245+0.01703568j,  0.17300818+0.69671008j,
     0.2916107 -0.0544079j ,  0.26340579+0.08529047j],
   [-0.23736656-0.00981797j, -0.08745847-0.33055207j,
     0.09369545+0.3232391j ,  0.27592583+0.21267878j]],

  [[-0.13553094+0.22373762j, -0.05251229-0.25721361j,
    -0.31291878+0.39677015j, -0.33833833-0.39650336j],
   [ 0.22981701+0.16598825j,  0.38360261+0.61955441j,
    -0.02036754-0.28016598j,  0.28384197-0.16017319j],
   [ 0.10228611+0.02818566j, -0.18246742-0.14240442j,
     0.10221811-0.43859385j,  0.43502243+0.01939254j]]]])
```

[9]:

```python
# construct S and T as full tensors
# (only for testing - in practice one usually works with the MPS matrices␣
 ↪directly!)
S = mps_to_full_tensor(Alist)
T = mps_to_full_tensor(Blist)

# should all agree
print("n:", n)
print("S.shape:", S.shape)
print("T.shape:", T.shape)

# dimension consistency checks
assert np.array_equal(np.array(S.shape), np.array(n))
assert np.array_equal(np.array(T.shape), np.array(n))
```

```
n: [2, 5, 1, 4, 3]
S.shape: (2, 5, 1, 4, 3)
T.shape: (2, 5, 1, 4, 3)
```

[10]:
```
# reference value for inner product
inner_ref = np.vdot(np.reshape(S, -1), np.reshape(T, -1))
inner_ref
```

[10]:
```
(0.2520280507127002-0.24177253548110633j)
```

[11]:
```
# compare with implementation based on efficient contraction
inner = mps_vdot(Alist, Blist)
inner
```

[11]:
```
(0.25202805071270024-0.24177253548110622j)
```

[12]:
```
# relative error (should be zero up to numerical rounding errors)
print("relative error:", abs(inner - inner_ref) / abs(inner_ref))
```

```
relative error: 3.5541437616807225e-16
```