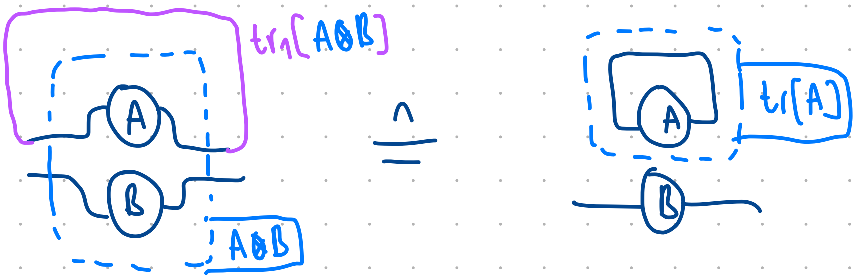
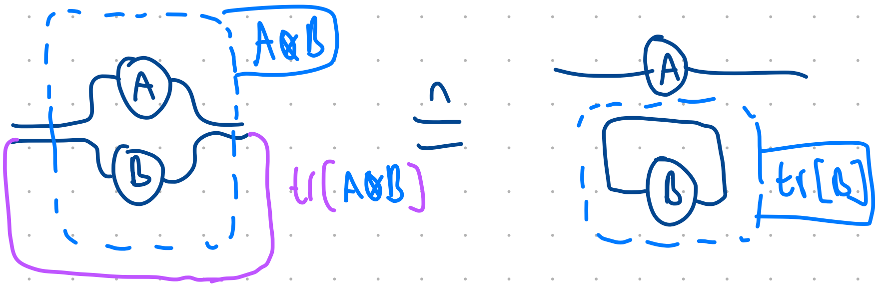
Group 2:
— Armin Ettenhofer
— Atul Agarwal
— Johannes Spies

5.1

(a) • $\mathrm{tr}_1[A \otimes B] \overset{!}{=} \mathrm{tr}[A] \cdot B$



• $\mathrm{tr}_2[A \otimes B] \overset{!}{=} \mathrm{tr}[B] \cdot A$



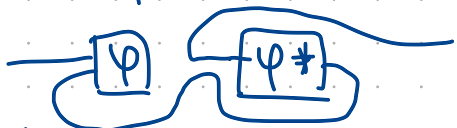
(b) $\mathrm{tr}_2[\Psi \circ \Psi^*] =$ 

$= \sqrt{\rho}\,(\sqrt{\rho}^*)^T$

$= \sqrt{\rho} \cdot \sqrt{\rho}^\dagger = \sqrt{\rho} \cdot \sqrt{\rho} = \rho$

$\sqrt{\rho}$ inherits Hermitian property

(c) • Let φ = matricization of Ψ ⇒ can write Ψ as $-\boxed{\varphi}-$

• $\mathrm{tr}_2(\Psi \circ \Psi^*] =$ 

$= \varphi(\varphi^*)^T = \varphi \cdot \varphi^\dagger$

• ↳ $\mathrm{tr}_2[\Psi \circ \Psi^*]$ is Hermitian:

$(\varphi \cdot \varphi^\dagger)^\dagger = ((\varphi \cdot \varphi^\dagger)^*)^T = (\varphi^* \cdot \varphi^T)^T = \varphi(\varphi^*)^T = \varphi \cdot \varphi^\dagger$

• Lecture says that $A^\dagger A$ is positive semidefinite, therefor $\varphi \cdot \varphi^\dagger$ as well.

5.2 (c)

- Tucker format tensor can be regarded as repeated application of U_l:
  $T \leftarrow C$
  for $l \in \{1, \dots, d\}$ do $T \leftarrow U_l \circ_l T$
- Each l-mode multiplication is contraction | matrix product along l-axis:
  → Last case: AB with: $\begin{cases} A = U_d \in \mathbb{C}^{n_d \times k_d} \\ B = \mathrm{lmode}(C, d) \in \mathbb{C}^{k_d \times (k_1 \cdots k_{d-1})} \end{cases}$

  general cases: AB with: $\begin{cases} A = U_l \in \mathbb{C}^{n_l \times k_l} \\ B = \mathrm{lmode}(U_{l+1} \cdots U_d \cdot C, l) \in \mathbb{C}^{k_l \times (k_1 \cdots k_{l-1} k_{l+1} \cdots k_d)} \end{cases}$
- using Hint: $\mathrm{rank}(AB) \leq \min\{n_l, k_l, \underbrace{k_1 \cdots k_{l-1} k_{l+1} \cdots k_d}_{\text{ignored for this multiplication}}\}$
- Since we choose $n_l \leq k_l$, $\mathrm{rank}(AB) \leq k_l$

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

In [2]:
```
# TODO: implement part (a)
def lmodemat(T, l):
    """Return l-mode matricization of tensor T"""
    d = T.ndim
    assert 0 <= l < d
    return np.reshape(
        np.transpose(T, axes=([l] + [ax for ax in range(d) if ax != l])),
        (T.shape[l], -1),
    )
```

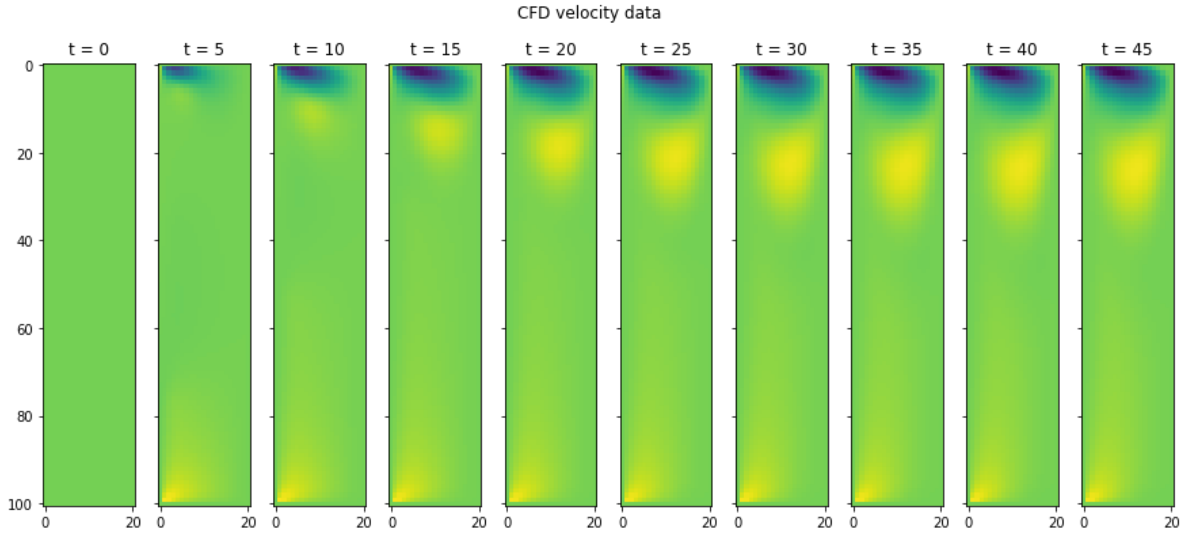
In [3]:
```
# TODO: implement part (b)
def lmodematprod(A, T, l):
    """Return l-mode matrix product between A and T"""
    return np.moveaxis(np.tensordot(A, T, axes=([1], [l])), 0, l)
```

In [4]:
```
# TODO: implement part (d)
def hosvd(T, ks):
    """Return the higher-order SVD of T"""
    d = len(ks)
    # compute SVDs of l-mode matricizations
    Uls_oss = (np.linalg.svd(lmodemat(T, l), full_matrices=False)[:2] for l
    Uls, oss = zip(*Uls_oss)
    # truncate Uls
    Ul_tildes = [Ul[:, :k] for k, Ul in zip(ks, Uls)]
    # form core tensor
    C_tilde = T
    for l, Ul_tilde in enumerate(Ul_tildes):
        C_tilde = lmodematprod(np.conj(Ul_tilde).T, C_tilde, l)
    return Ul_tildes, C_tilde, oss
```

In [5]:
```
# load computational fluid dynamics (CFD) velocity data from disk
# (array only contains the y-component of the velocity vector)
vcfd = np.load("cfd_velocity.npy")
# dimensions are (x, y, time)
vcfd.shape
```

Out[5]:
```
(21, 101, 50)
```

In [6]:
```
# visualize data
fig, ax = plt.subplots(1, 10, sharey=True, figsize=(15, 6))
plt.suptitle("CFD velocity data")
# use same color range in all subplots
vmin_dat = np.min(vcfd)
vmax_dat = np.max(vcfd)
for j in range(10):
    ax[j].imshow(vcfd[:, :, 5*j].T, vmin=vmin_dat, vmax=vmax_dat)
    ax[j].set_title("t = {}".format(5*j))
plt.show()
```

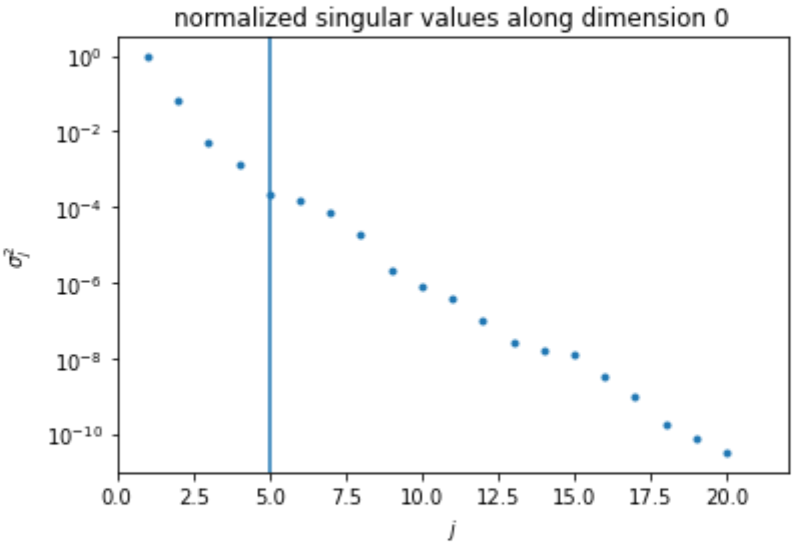


In [7]:
```
# perform HOSVD
max_ranks = [5, 10, 8]
# TODO: uncomment to call your implementation of the HOSVD here
Ulist, C, σlist = hosvd(vcfd, max_ranks)
# must agree with `max_ranks`
print("C.shape:", C.shape)
np.allclose(C.shape, max_ranks)
```

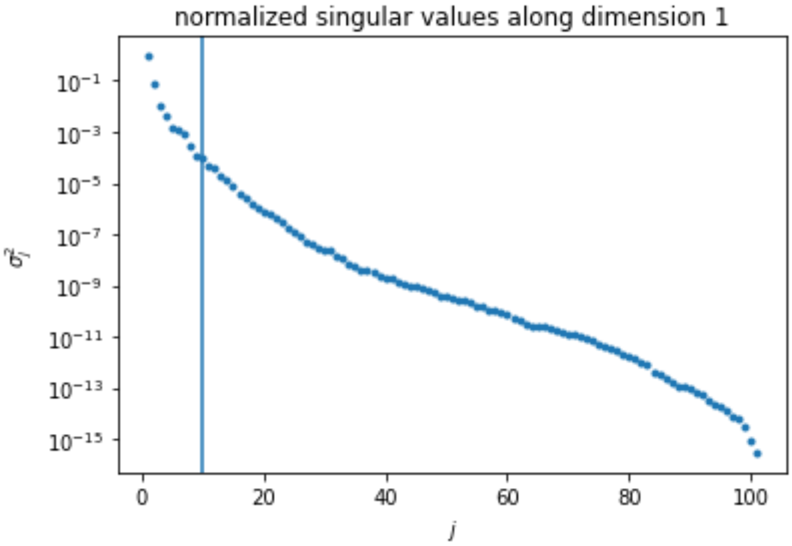
```
C.shape: (5, 10, 8)
```

Out[7]:
```
True
```

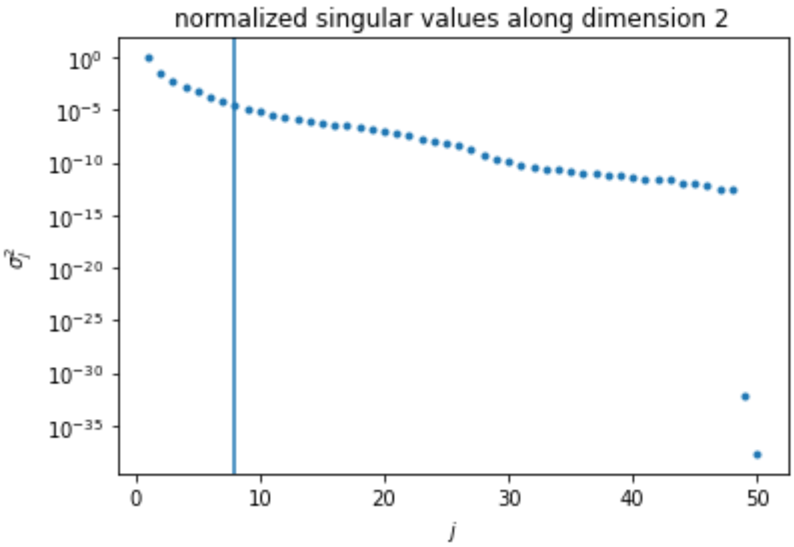
In [8]:
```
plt.semilogy(range(1, len(σlist[0]) + 1), σlist[0]**2 / np.sum(σlist[0]**2)
plt.axvline(x=max_ranks[0])
plt.ylabel("$\\sigma_j^2$")
plt.xlabel("$j$")
plt.title("normalized singular values along dimension 0");
plt.show()
```



In [9]:
```
plt.semilogy(range(1, len(σlist[1]) + 1), σlist[1]**2 / np.sum(σlist[1]**2)
plt.axvline(x=max_ranks[1])
plt.ylabel("$\\sigma_j^2$")
plt.xlabel("$j$")
plt.title("normalized singular values along dimension 1");
plt.show()
```



In [10]:
```
plt.semilogy(range(1, len(σlist[2]) + 1), σlist[2]**2 / np.sum(σlist[2]**2)
plt.axvline(x=max_ranks[2])
plt.ylabel("$\\sigma_j^2$")
plt.xlabel("$j$")
plt.title("normalized singular values along dimension 2");
plt.show()
```

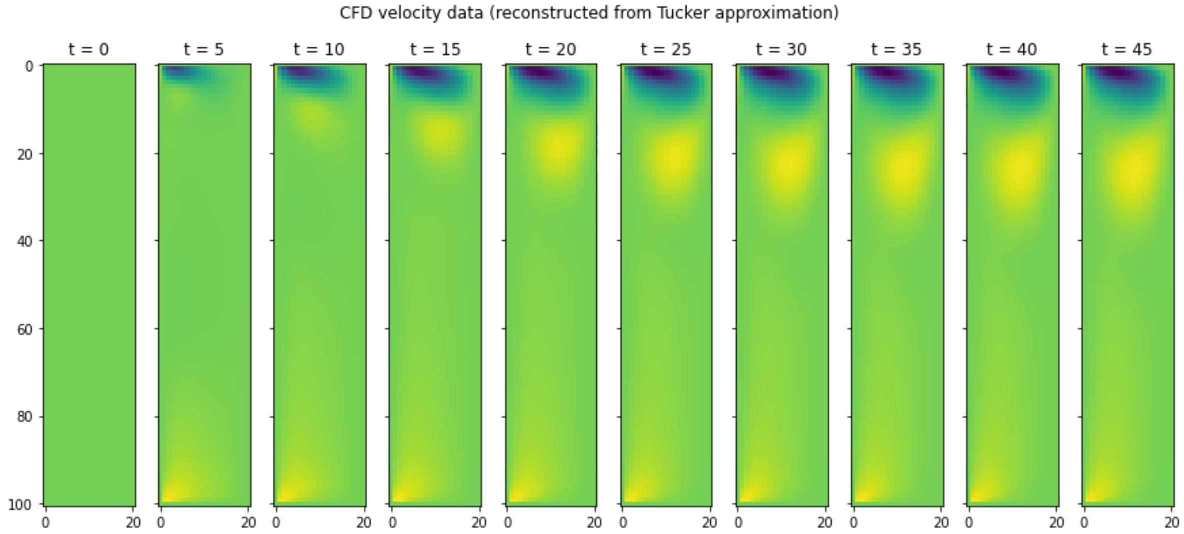


In [11]:
```
# The Tucker tensor as "full" tensor should only be constructed for debuggi
# Typically one works with the `U` matrices and the core tensor `C` directl
def construct_tucker_tensor(Ulist, C):
    """
    Construct the full Tucker tensor based on the `U` matrices and the core
    """
    assert C.ndim == len(Ulist)
    T = C
    for j in range(T.ndim):
        # apply Uj to j-th dimension
        # TODO: uncomment to call your function from part (b) here
        T = lmodematprod(Ulist[j], T, j)
    return T
```

In [12]:
```
vcfd_tucker = construct_tucker_tensor(Ulist, C)
# should be equal to original dimensions
vcfd_tucker.shape
```

Out[12]:
```
(21, 101, 50)
```

In [13]:
```
# visualize reconstructed Tucker approximation (should visually match the o
fig, ax = plt.subplots(1, 10, sharey=True, figsize=(15, 6))
plt.suptitle("CFD velocity data (reconstructed from Tucker approximation)")
for j in range(10):
    ax[j].imshow(vcfd_tucker[:, :, 5*j].T, vmin=vmin_dat, vmax=vmax_dat)
    ax[j].set_title("t = {}".format(5*j))
plt.show()
```



In [ ]: