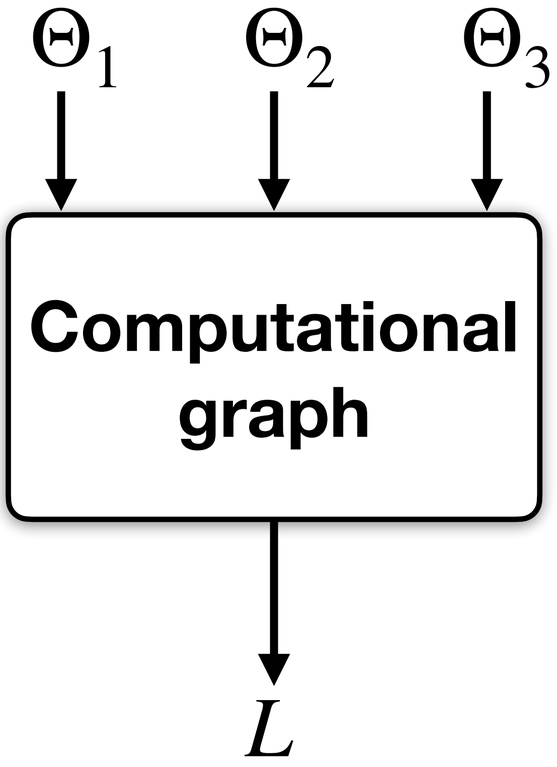
Rapid prototyping of quantum information processing tasks with deep learning libraries

Ilia Luchnikov

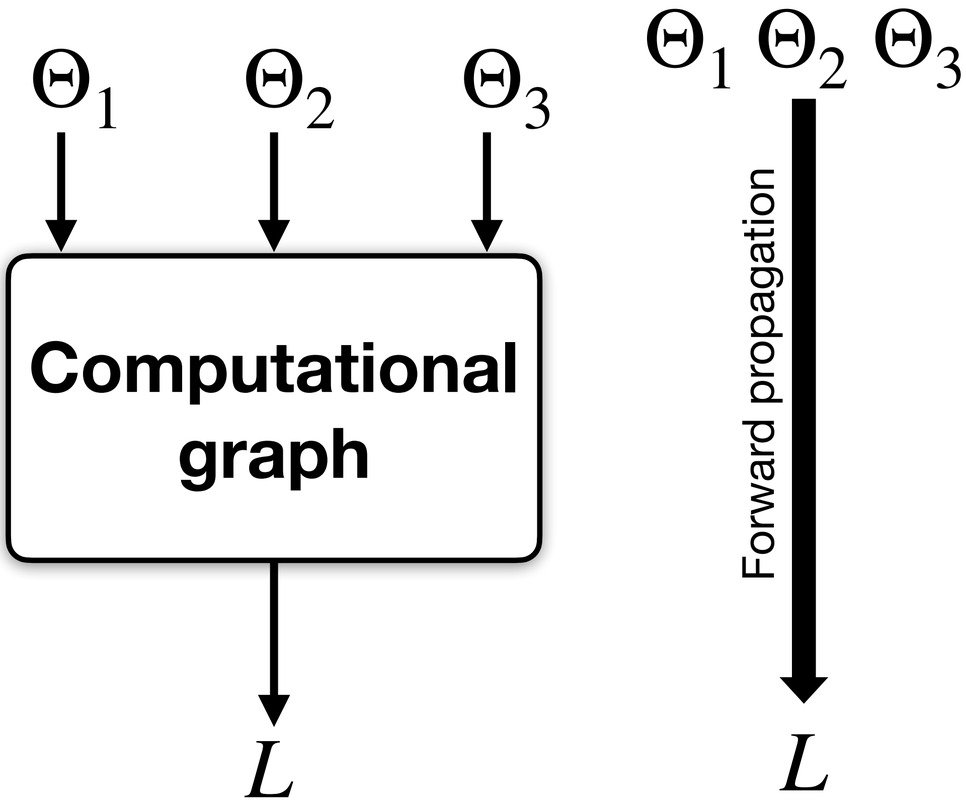
Plan

1. Introduction to TensorFlow
2. Spin echo simulation and optimization
3. AD based optimization of a matrix product state
4. Constrained optimization in quantum technologies with QGOpt

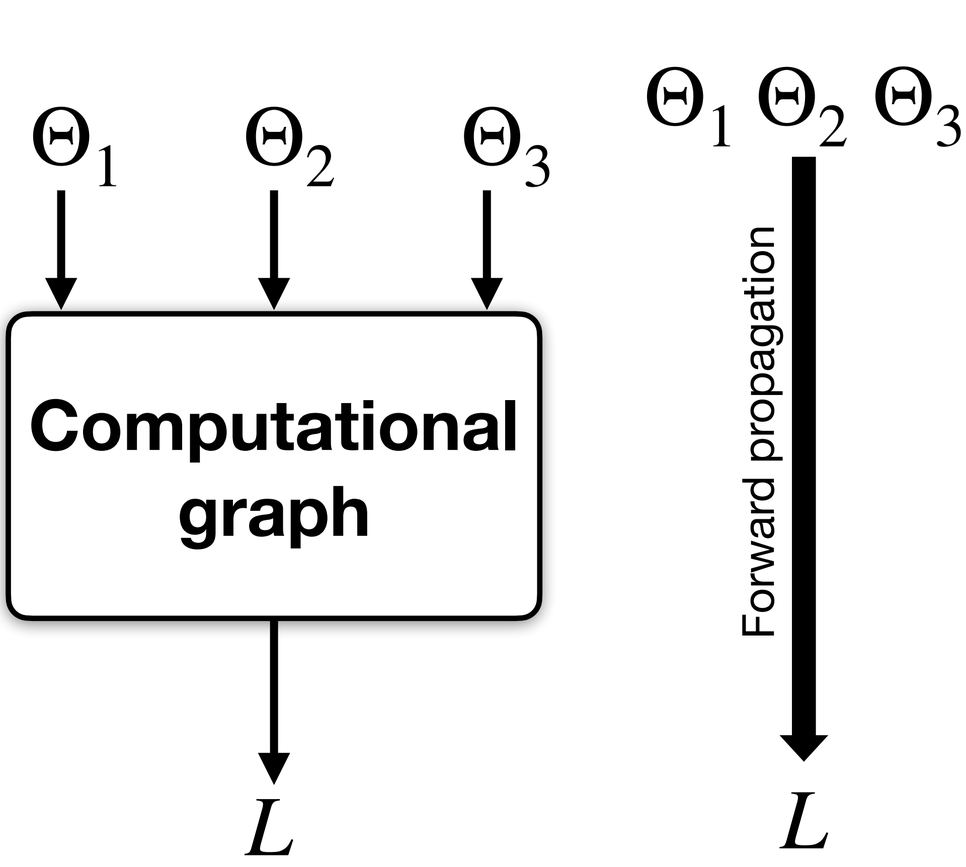
Key feature of deep learning (or AD) libraries: automatic differentiation

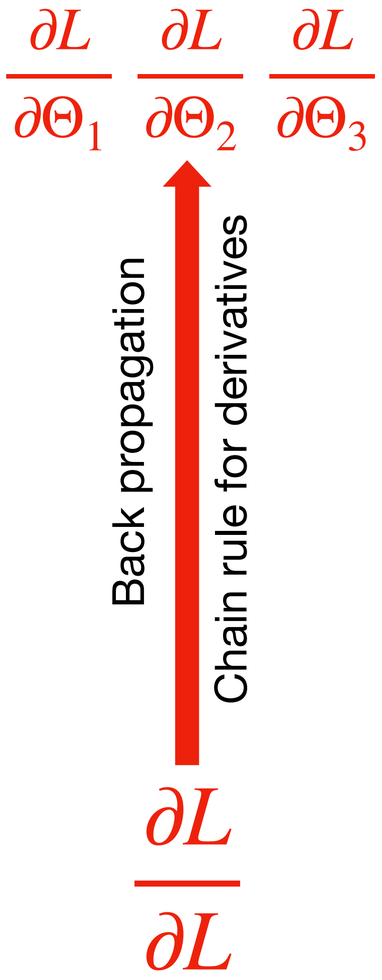


Key feature of deep learning (or AD) libraries: automatic differentiation

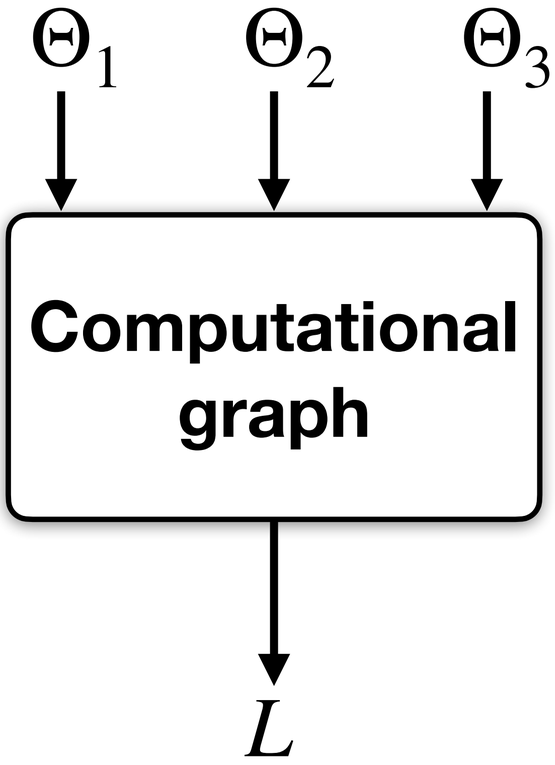


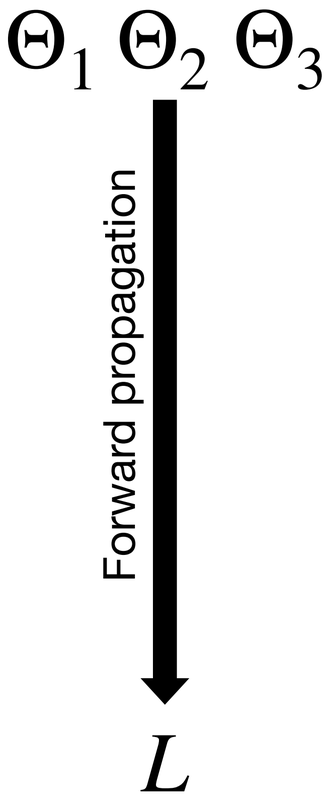
Key feature of deep learning (or AD) libraries: automatic differentiation





Key feature of deep learning (or AD) libraries: automatic differentiation

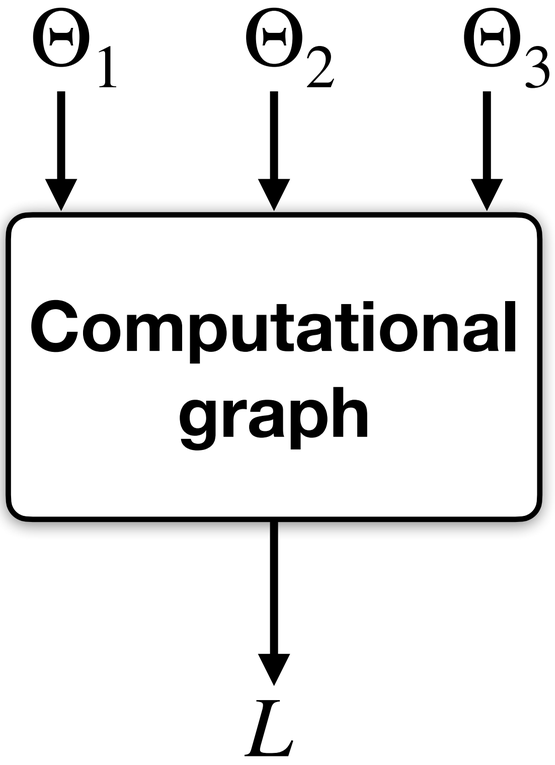


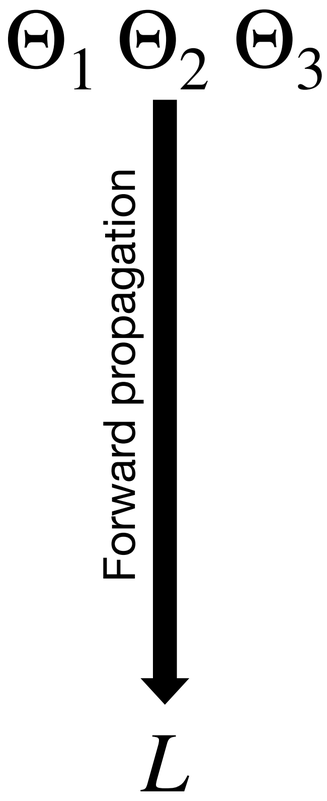


- **Caffe**
- **PyTorch**
- **TensorFlow**
- **Jax**
- **Flux**

etc

Key feature of deep learning (or AD) libraries: automatic differentiation





- **Caffe**
- **PyTorch**

Latest version for python

- **TensorFlow**
- **Jax**
- **Flux**

etc

The extended version of all pieces of code is available here

https://github.com/Luchnikovl/RQC_School_Rapid_prototyping

Introduction to TensorFlow

Tensors in TensorFlow

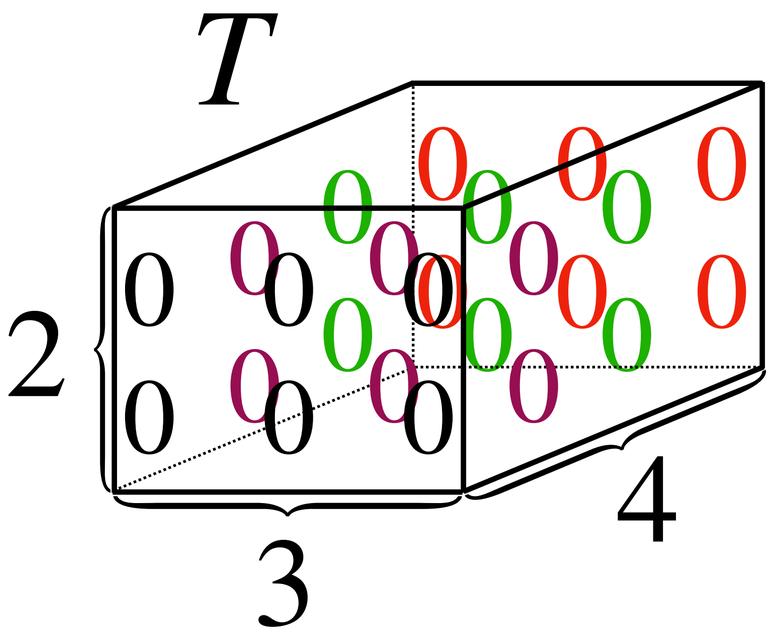
$$T \in \mathbb{C}^{n_1 \times, \ldots, \times n_k}$$

Initialization of tensors

```
1   T = tf.zeros((2, 3, 4), dtype=tf.complex128)
```

Initialization of tensors

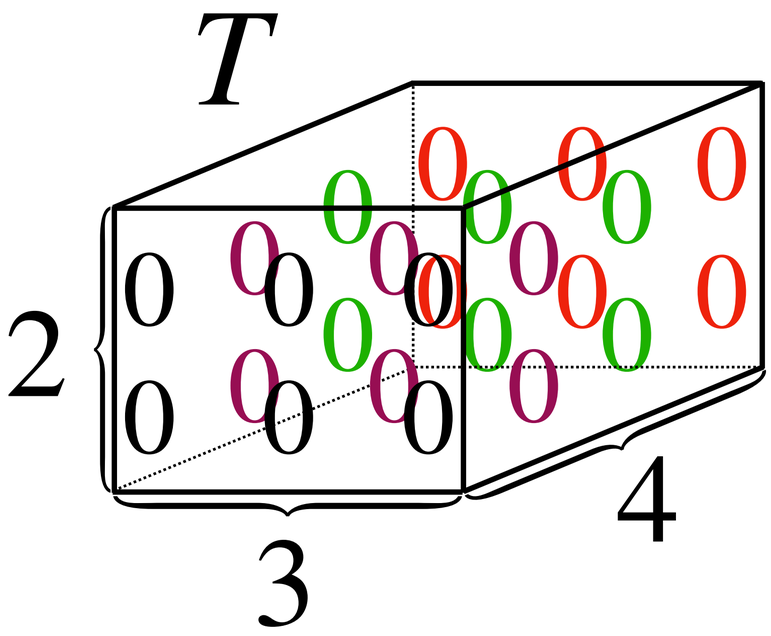
```
1   T = tf.zeros((2, 3, 4), dtype=tf.complex128)
```



$$T \in \mathbb{C}^{2\times3\times4}$$

Initialization of tensors

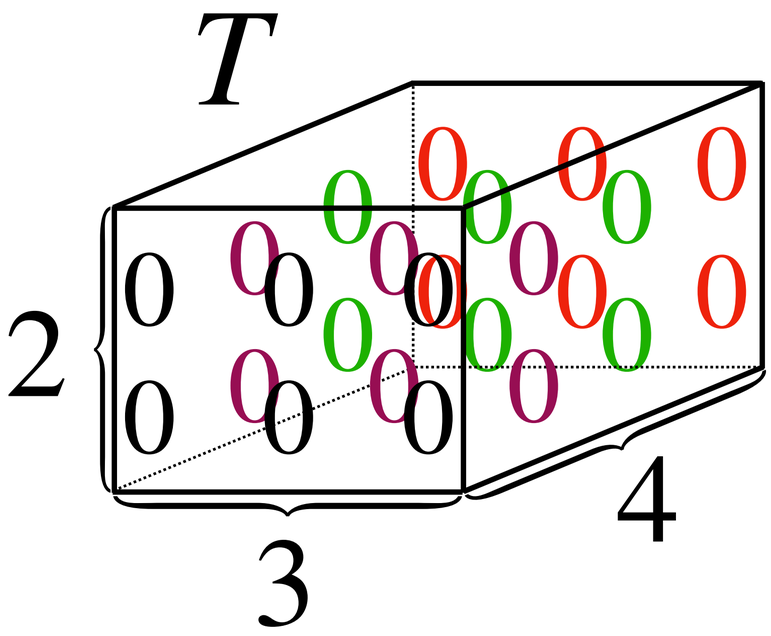
```
1   T = tf.zeros((2, 3, 4), dtype=tf.complex128)
```



$$T \in \mathbb{C}^{2\times3\times4}$$

Initialization of tensors

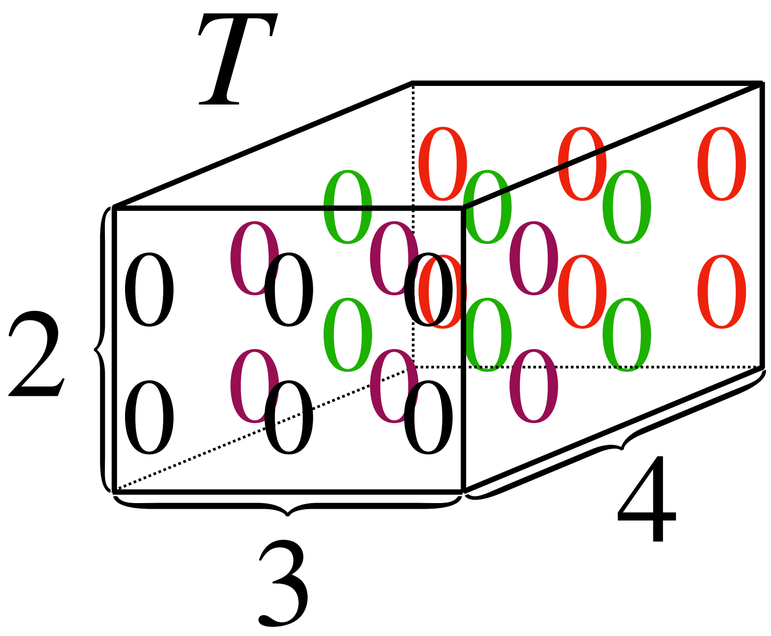
```
1   T = tf.zeros((2, 3, 4), dtype=tf.complex128)
```



$$T \in \mathbb{C}^{2\times3\times4}$$

Initialization of tensors

```
1   T = tf.zeros((2, 3, 4), dtype=tf.complex128)
```



$$T \in \mathbb{C}^{2\times3\times4}$$

```
1   print(T.shape)
2   print(T.dtype)
```

```
(2, 3, 4)
<dtype: 'complex128'>
```

Initialization of tensors

$T \in \mathbb{R}^{2\times3\times4},\ T \sim N(0,\ I)$

```
1   T = tf.random.normal((2, 3, 4), dtype=tf.float64)
```

Initialization of tensors

$T \in \mathbb{R}^{2\times3\times4},\ T \sim N(0,\ I)$

```
1   T = tf.random.normal((2, 3, 4), dtype=tf.float64)
```

$T = I \in \mathbb{C}^{4\times4}$

```
1   T = tf.eye(4, dtype=tf.complex128)
```

Initialization of tensors

$T \in \mathbb{R}^{2\times3\times4},\ T \sim N(0,\ I)$

```
1   T = tf.random.normal((2, 3, 4), dtype=tf.float64)
```

$T = I \in \mathbb{C}^{4\times4}$

```
1   T = tf.eye(4, dtype=tf.complex128)
```

```
1   T = tf.constant([1, 2, 5, 10], dtype=tf.complex128)
```

Initialization of tensors

$T \in \mathbb{R}^{2\times3\times4}$, $T \sim N(0, I)$

```
T = tf.random.normal((2, 3, 4), dtype=tf.float64)
```

$T = I \in \mathbb{C}^{4\times4}$

```
T = tf.eye(4, dtype=tf.complex128)
```

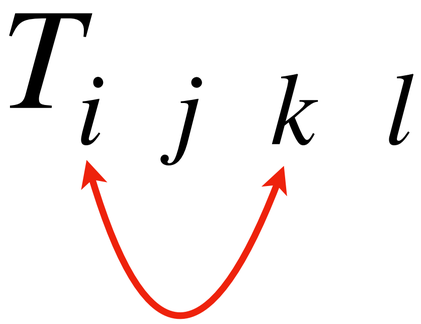
```
T = tf.constant([1, 2, 5, 10], dtype=tf.complex128)
```

etc

Transpose tensors

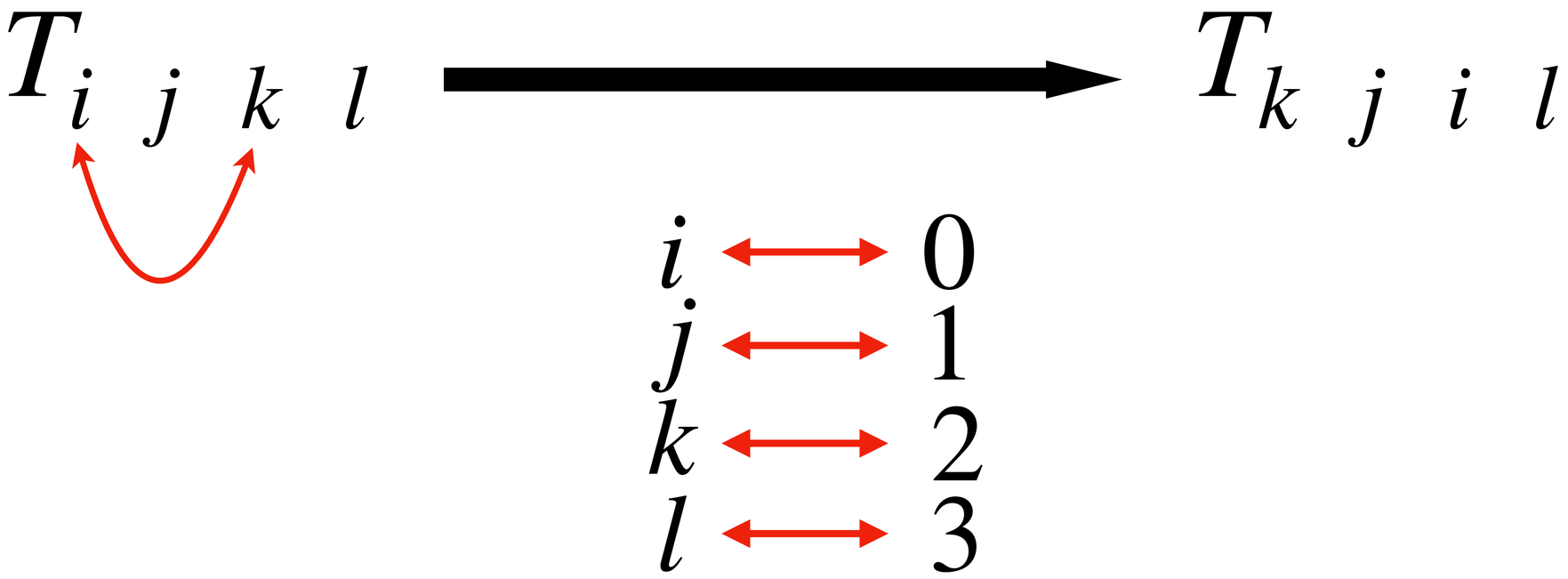
$$T_{i\ j\ k\ l}$$

Transpose tensors

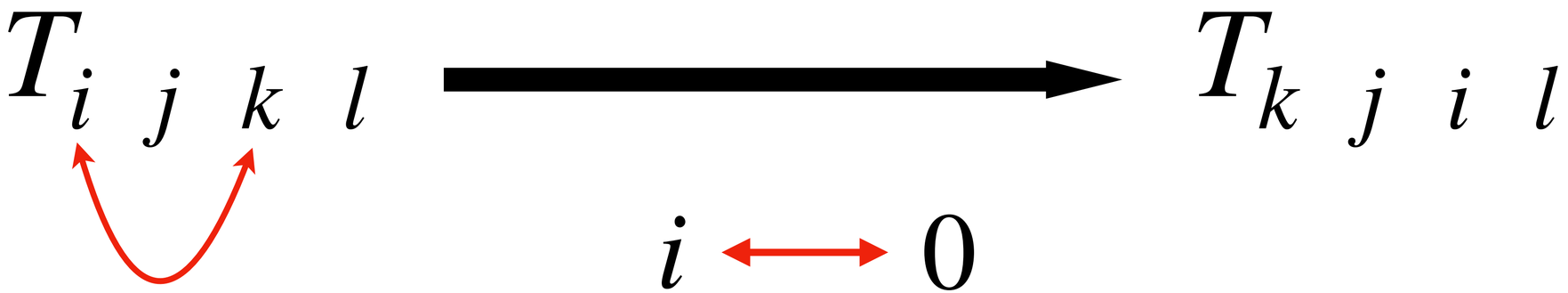


Transpose tensors

Transpose tensors



Transpose tensors



$$i \longleftrightarrow 0$$
$$j \longleftrightarrow 1$$
$$k \longleftrightarrow 2$$
$$l \longleftrightarrow 3$$

$k\ \ j\ \ i\ \ l$

```
1    T = tf.transpose(T, (2, 1, 0, 3))
```

Reshape tensors

$$T \in \mathbb{R}^{3\times2\times3\times2}$$

Reshape tensors

$$T \in \mathbb{R}^{3\times2\times3\times2}$$

$$T_{\underbrace{i\ j}_{q}\ k\ l}$$

Reshape tensors

$$T \in \mathbb{R}^{3\times2\times3\times2}$$

$$T_{\underbrace{i\ \ j}_{q}\ \ k\ \ l}$$

$$2 \times 2 \to 4$$

$$\begin{aligned} 00 &\to 0 \\ 01 &\to 1 \\ 10 &\to 2 \\ 11 &\to 3 \end{aligned}$$

Reshape tensors

$$T \in \mathbb{R}^{3\times2\times3\times2}$$

$$T_{\underbrace{i\ j}_{q}\ k\ l} \longrightarrow T_{q\ k\ l} \in \mathbb{R}^{6\times3\times2}$$

$2 \times 2 \to 4$

$00 \to 0$
$01 \to 1$
$10 \to 2$
$11 \to 3$

Reshape tensors

$$T \in \mathbb{R}^{3\times2\times3\times2}$$

$$T_{\underbrace{i\ j}_{q}\ k\ l} \longrightarrow T_{q\ k\ l} \in \mathbb{R}^{6\times3\times2}$$

```
1    T = tf.reshape(T, (6, 3, 2))
```

Reshape tensors

$$T \in \mathbb{R}^{3\times2\times3\times2}$$

$$T_{\underbrace{i\ j}_{q}\ k\ l} \longrightarrow T_{q\ k\ l} \in \mathbb{R}^{6\times3\times2}$$

```
1    T = tf.reshape(T, (6, 3, 2))
```

Reshape tensors

$$T \in \mathbb{R}^{6\times3\times2}$$

$$T_{q\ k\ l} \longrightarrow T_{i\ j\ k\ l} \in \mathbb{R}^{3\times2\times3\times2}$$

```
1    T = tf.reshape(T, (3, 2, 3, 2))
```

Partial density matrices

Bell state initialization

$$|\psi\rangle = \left(|1\rangle \otimes |1\rangle + |0\rangle \otimes |0\rangle\right)$$

```
psi = tf.constant([1, 0, 0, 1], dtype=tf.complex128)
```

Partial density matrices

Bell state initialization

$$|\psi\rangle = \left(|1\rangle \otimes |1\rangle + |0\rangle \otimes |0\rangle\right)$$

```
psi = tf.constant([1, 0, 0, 1], dtype=tf.complex128)
```

$$|\psi\rangle = \frac{1}{\sqrt{2}}\left(|1\rangle \otimes |1\rangle + |0\rangle \otimes |0\rangle\right)$$

```
psi = psi / tf.linalg.norm(psi)
```

Partial density matrices

Bell state initialization

$$|\psi\rangle = \left(|1\rangle \otimes |1\rangle + |0\rangle \otimes |0\rangle\right)$$

```
psi = tf.constant([1, 0, 0, 1], dtype=tf.complex128)
```

$$|\psi\rangle = \frac{1}{\sqrt{2}}\left(|1\rangle \otimes |1\rangle + |0\rangle \otimes |0\rangle\right)$$

```
psi = psi / tf.linalg.norm(psi)
```

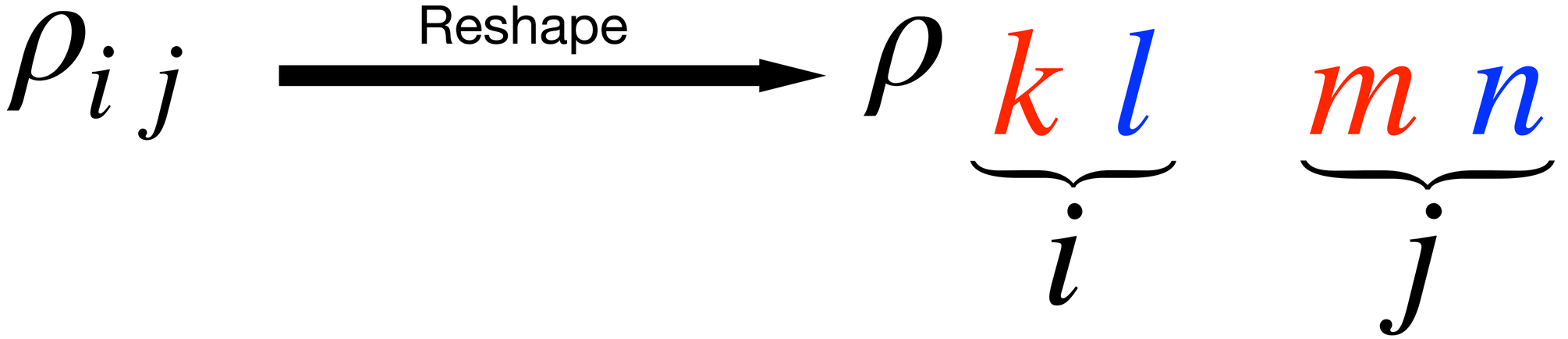
$$\rho = |\psi\rangle\langle\psi|$$

```
rho = tf.tensordot(psi, tf.math.conj(psi), axes=0)
```

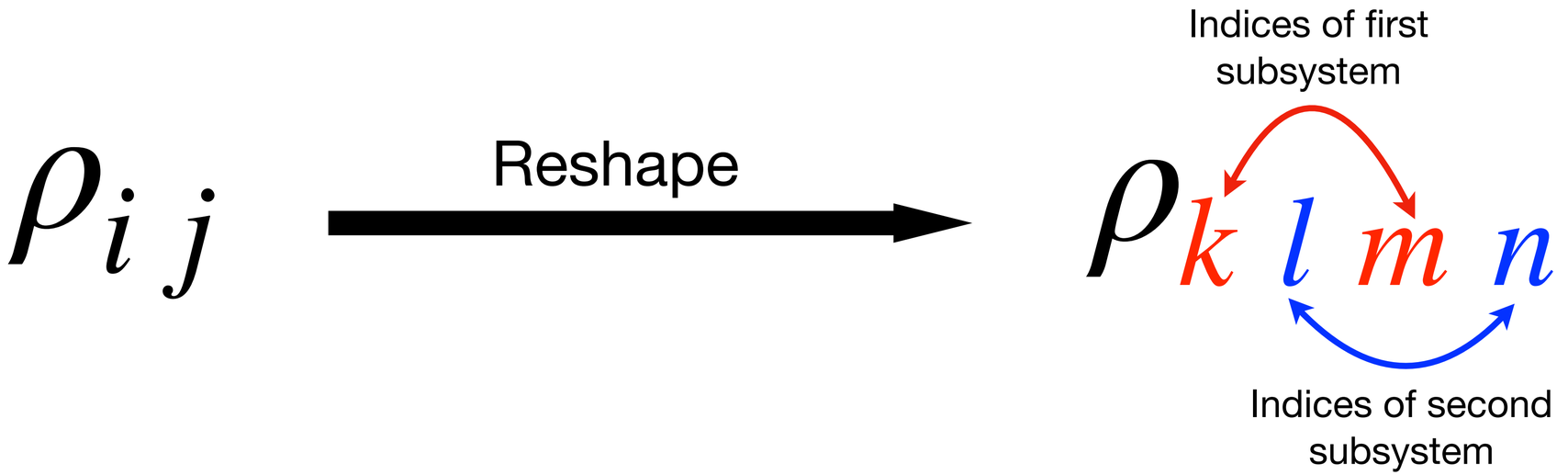
Partial density matrices

$$\rho_{i\,j}$$

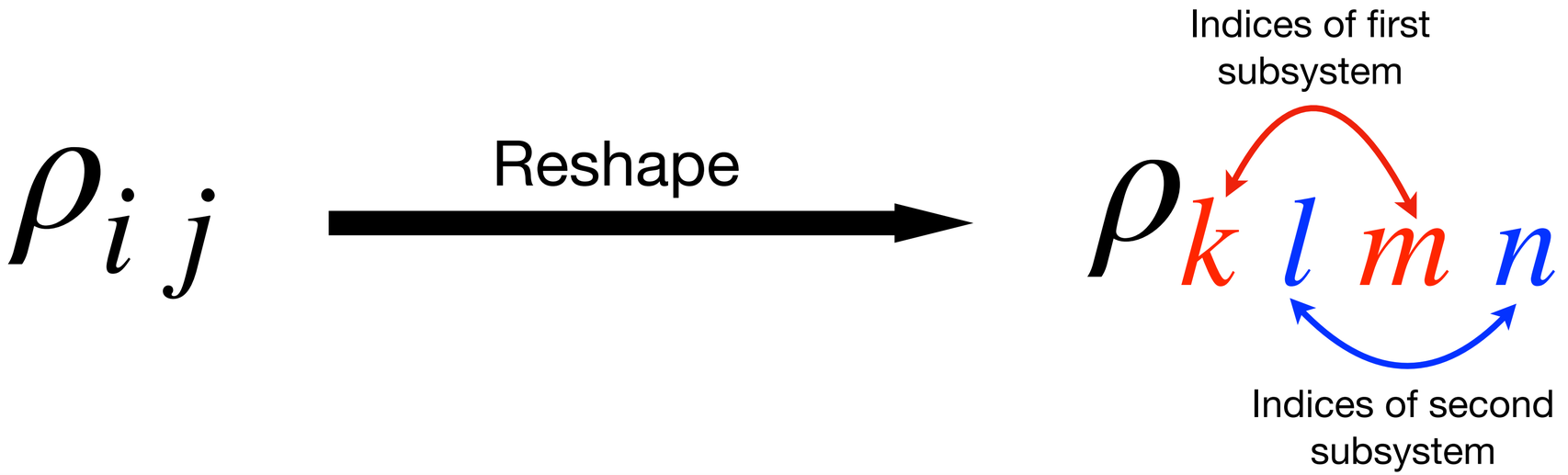
Partial density matrices



Partial density matrices

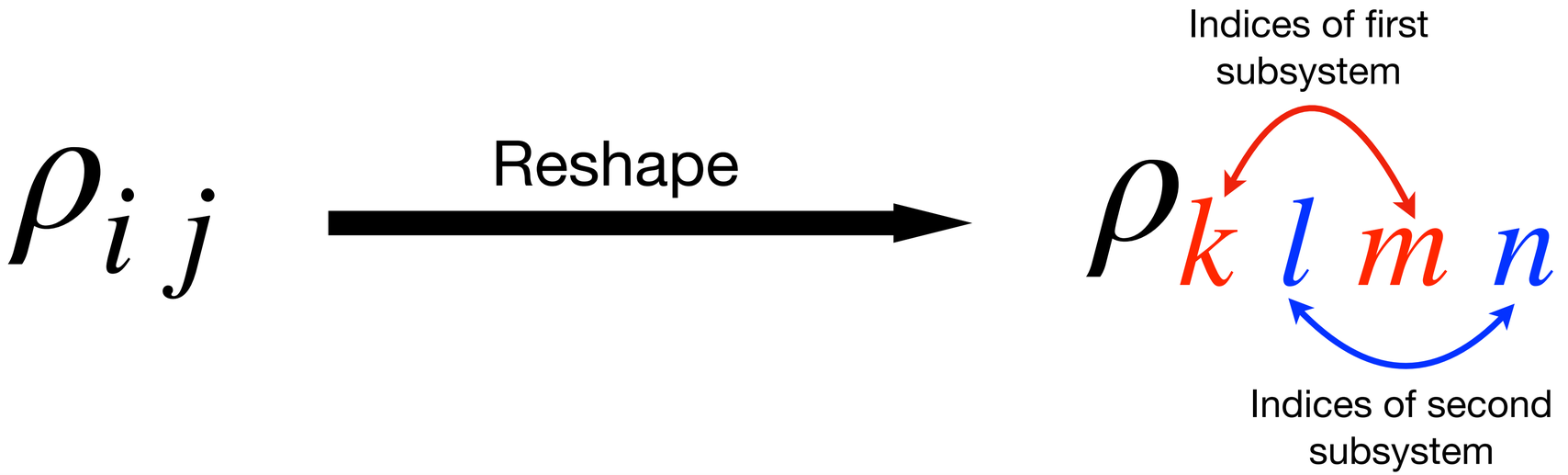


Partial density matrices




```
rho_resh = tf.reshape(rho, (2, 2, 2, 2))
```

Partial density matrices

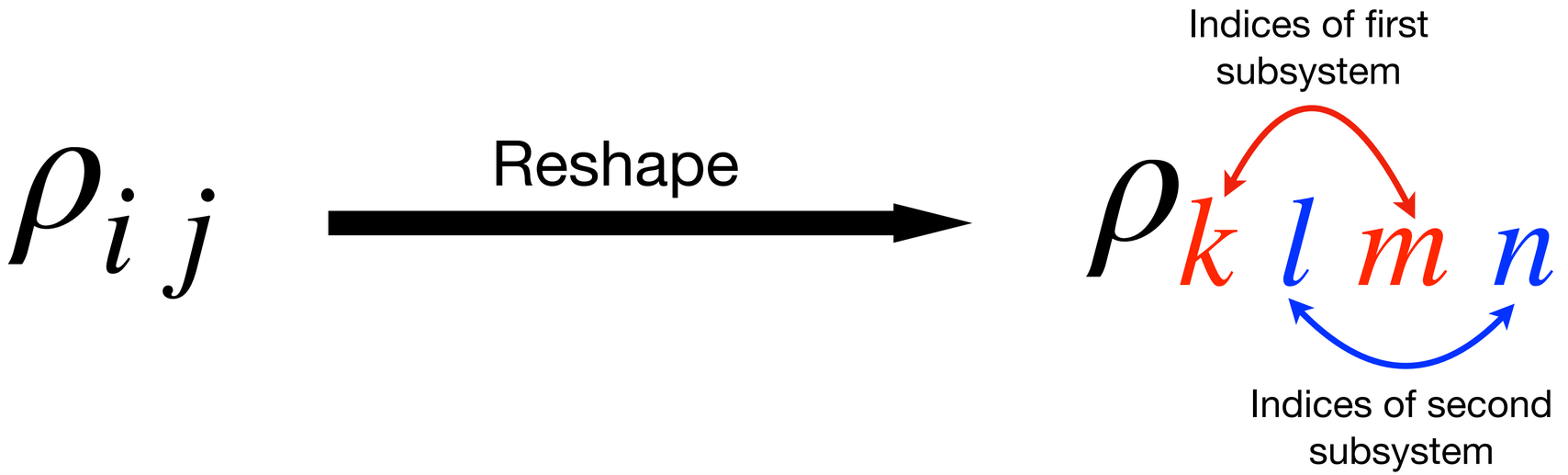



```
rho_resh = tf.reshape(rho, (2, 2, 2, 2))
```

$$\rho^{L}_{km} = \sum_{l} \rho_{k\ l\ m\ l}$$

```
rhoL = tf.einsum('klml->km', rho_resh)
```

Partial density matrices




```
rho_resh = tf.reshape(rho, (2, 2, 2, 2))
```

$$\rho^{L}_{km} = \sum_{l} \rho_{k\ l\ m\ l}$$

```
rhoL = tf.einsum('klml->km', rho_resh)
```

$$\rho^{R}_{ln} = \sum_{k} \rho_{k\ l\ k\ n}$$

```
rhoR = tf.einsum('klkn->ln', rho_resh)
```

Partial density matrices

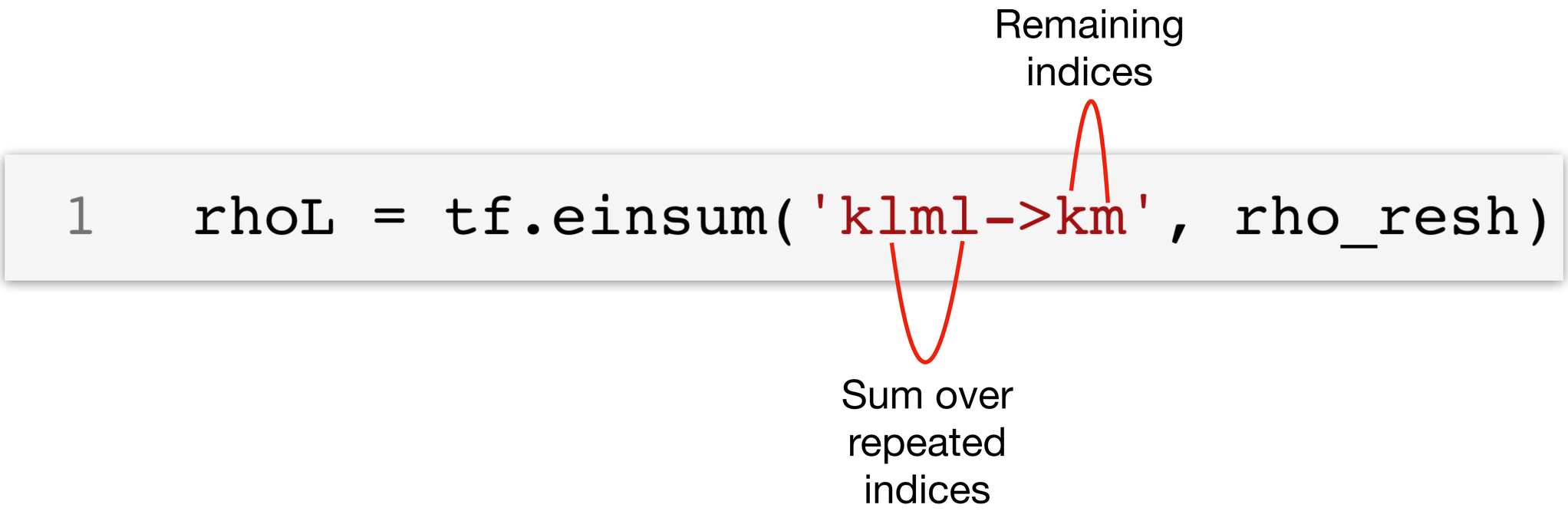
```
rhoL = tf.einsum('klml->km', rho_resh)
```

Partial density matrices

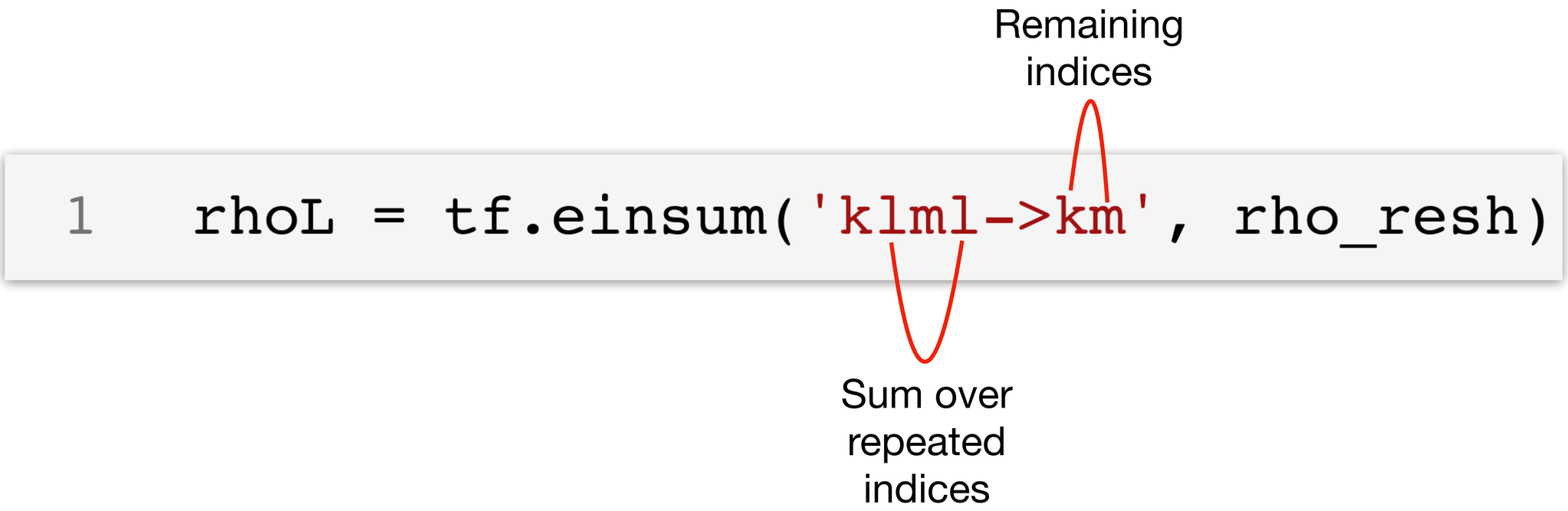
```
1   rhoL = tf.einsum('klml->km', rho_resh)
```

Sum over repeated indices

Partial density matrices



Partial density matrices




```
1   rhoL = tf.einsum('klml->km', rho_resh)
```

```
1   print(rhoL)
```

```
tf.Tensor(
[[0.5+0.j 0. +0.j]
 [0. +0.j 0.5+0.j]], shape=(2, 2), dtype=complex128)
```

```
1   print(rhoR)
```

```
tf.Tensor(
[[0.5+0.j 0. +0.j]
 [0. +0.j 0.5+0.j]], shape=(2, 2), dtype=complex128)
```

Partial density matrices

Remaining indices

```
1   rhoL = tf.einsum('klml->km', rho_resh)
```

Sum over repeated indices

$$\rho^L = \rho^R = \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{bmatrix}$$

```
1   print(rhoL)
```

```
tf.Tensor(
[[0.5+0.j 0. +0.j]
 [0. +0.j 0.5+0.j]], shape=(2, 2), dtype=complex128)
```

```
1   print(rhoR)
```

```
tf.Tensor(
[[0.5+0.j 0. +0.j]
 [0. +0.j 0.5+0.j]], shape=(2, 2), dtype=complex128)
```

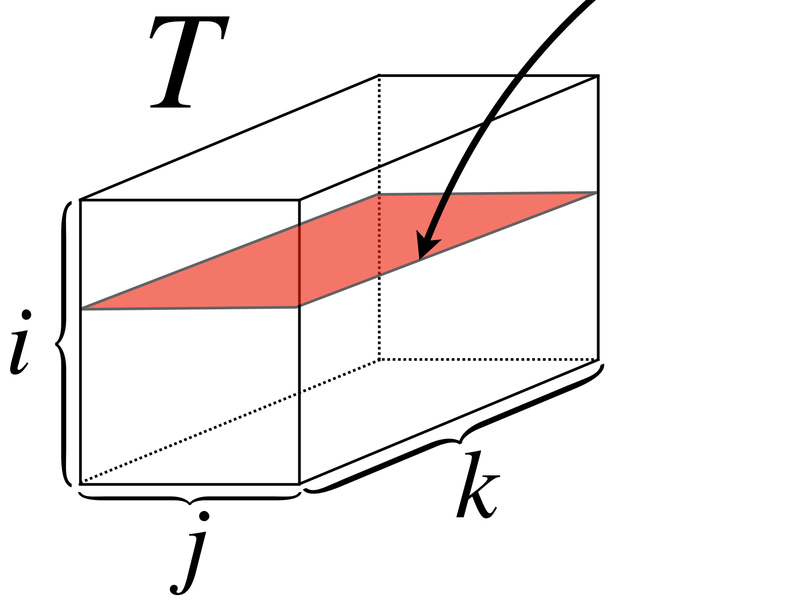
Slicing

$$T \in \mathbb{R}^{5 \times 3 \times 8}$$

Slicing

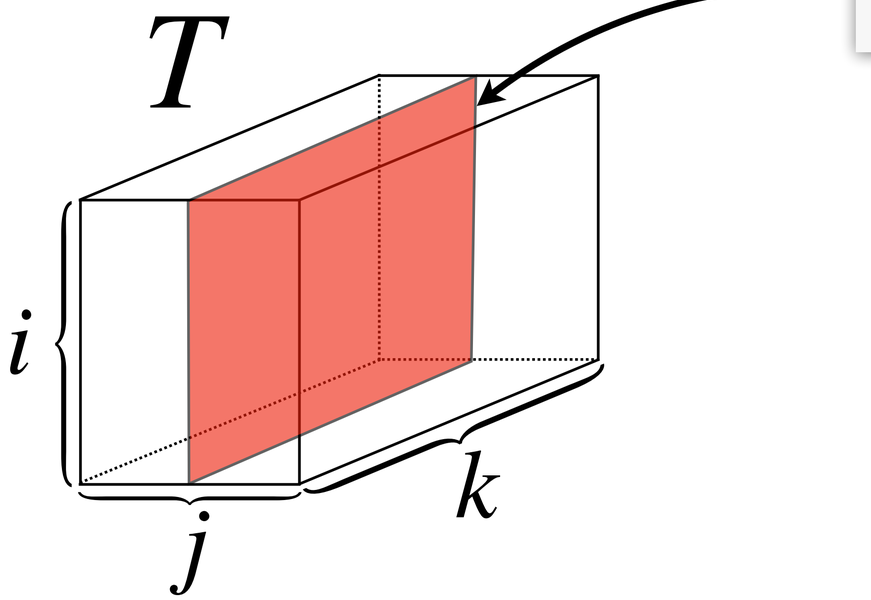
$$T \in \mathbb{R}^{5\times3\times8}$$

```
1   T[2]  =  1   T[-3]
```



Slicing

$$T \in \mathbb{R}^{5\times3\times8}$$



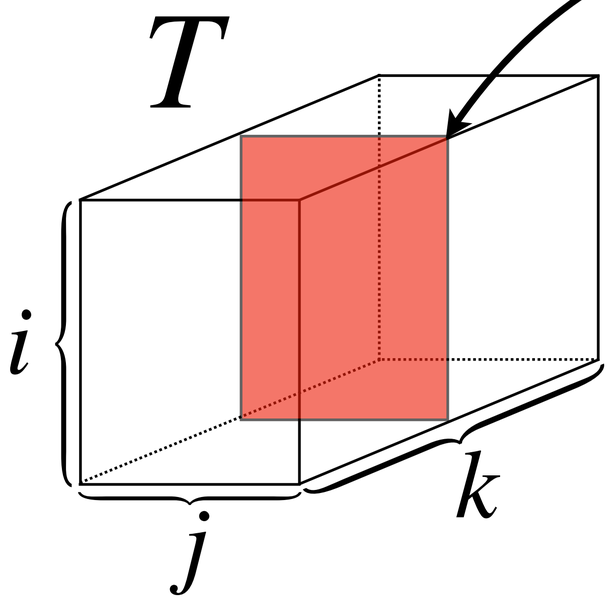

```
1   T[:, 1]
```

=

```
1   T[:, -2]
```

Slicing

$$T \in \mathbb{R}^{5 \times 3 \times 8}$$




```
T[:, :, 4]
```

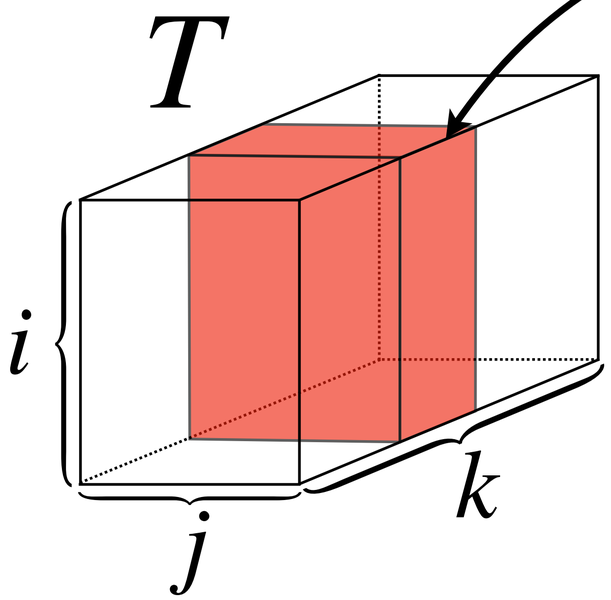
=

```
T[:, :, -4]
```

Slicing

$$T \in \mathbb{R}^{5 \times 3 \times 8}$$

```
1   T[:, :, 2:5]
```



Element—wise operations

If the shape of D is equal to the shape of E

```
D * E  # element wise multiplication
D + E  # element wise summation
D - E  # element wise subtraction
D / E  # element wise division
D ** E  # element wise exponentiation
D > E  # element wise comparison, returns tensor of type bool
D < E  # element wise comparison, returns tensor of type bool
D == E  # element wise comparison, returns tensor of type bool
D != E  # element wise comparison, returns tensor of type bool
```

etc

Element—wise operations

```
1   tf.math.exp(D)  # element wise exponent
2   tf.math.log(D)  # element wise logarithm
3   tf.math.atan(D)  # element wise atan
4   tf.math.sqrt(D)  # element wise sqrt
```

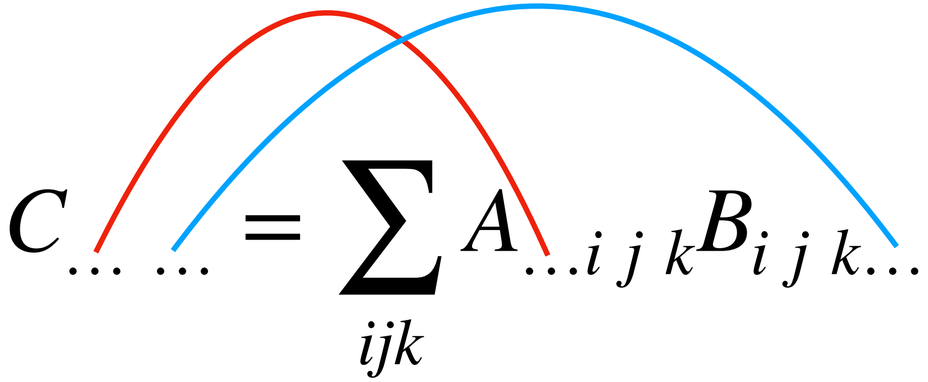
etc

Linear algebra

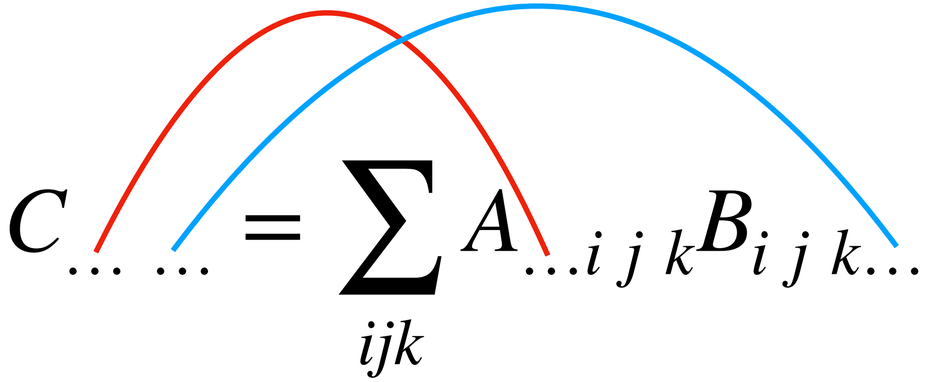
```
Q, R = tf.linalg.qr(K)  # QR decomposition of a matrix
S, U, V = tf.linalg.svd(K)  # SVD decomposition of a matrix
eigvals, right_eigvecs = tf.linalg.eig(K)  # eigen decomposition of a matrix
expC = tf.linalg.expm(K)  # matrix exponent
C_inv = tf.linalg.inv(K)  # inverse matrix
K_adj = tf.linalg.adjoint(K)  # conjugate transpose matrix
det = tf.linalg.det(K)  # determinant of a matrix
sqrt_K = tf.linalg.sqrtm(K)  # matrix sqrt
```

etc

Convolutions of tensors

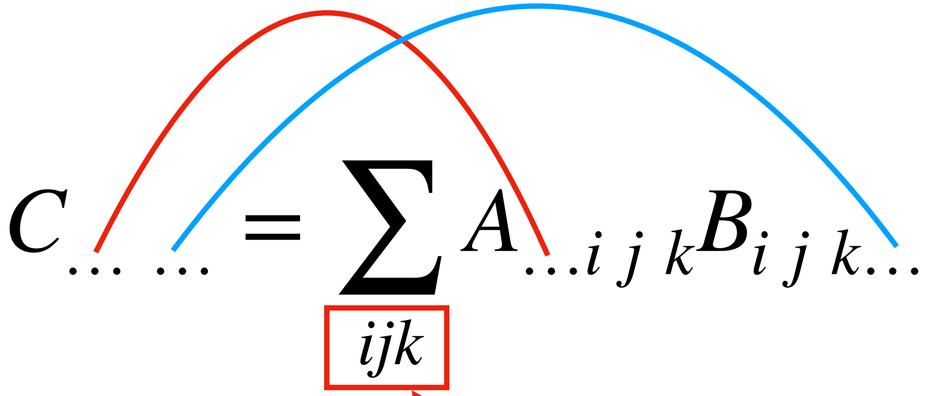


Convolutions of tensors




```
C = tf.tensordot(A, B, axes=3)
```

Convolutions of tensors




```
1    C = tf.tensordot(A, B, axes=3)
```

Convolutions of tensors

$$\sum_{ln} A_{j\ l\ n} B_{n\ i\ l} = C_{i\ j}$$

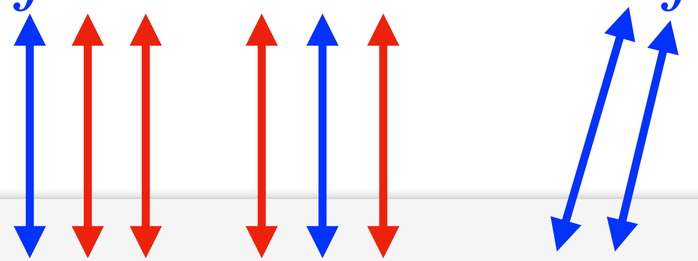
Convolutions of tensors

$$\sum_{ln} A_{j\,l\,n} B_{n\,i\,l} = C_{i\,j}$$

```
1   C = tf.einsum('jln,nil->ij', A, B)
```

Convolutions of tensors

$$\sum_{ln} A_{j\,l\,n} B_{n\,i\,l} = C_{i\,j}$$



```
1   C = tf.einsum('jln,nil->ij', A, B)
```

Convolutions of tensors

$$\sum_{ln} A_{j\,l\,n} B_{n\,i\,l} = C_{i\,j}$$

```
1   C = tf.einsum('jln,nil->ij', A, B)
```

There is no Red indices there, it means that we summed them out

Matrix multiplication

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

```
1    C = A @ B
```

Matrix multiplication

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

```
1    C = A @ B
```

But!

$$C_{aij} = \sum_k A_{aik} B_{akj}$$

```
1    C = A @ B
```

Matrix multiplication

$$C_{aij} = \sum_k A_{aik} B_{akj}$$

```
1    C = A @ B
```

Only last two indices are considered as matrix indices!

Matrix multiplication

```
1   A = tf.random.normal((2, 3, 4, 5))
2   B = tf.random.normal((2, 3, 5, 4))
3   print((A @ B).shape)
```

```
(2, 3, 4, 4)
```

Matrix multiplication

Aren't touched

```
1   A = tf.random.normal((2, 3, 4, 5))
2   B = tf.random.normal((2, 3, 5, 4))
3   print((A @ B).shape)
```

```
(2, 3, 4, 4)
```

Matrix multiplication

Aren't touched

mat. mul.

```
1   A = tf.random.normal((2, 3, 4, 5))
2   B = tf.random.normal((2, 3, 5, 4))
3   print((A @ B).shape)
```

```
(2, 3, 4, 4)
```

```
1   A = tf.random.normal((100, 100, 5, 5))
2   s, u, v = tf.linalg.svd(A)
3   print(s.shape)
4   print(u.shape)
5   print(v.shape)
```

```
(100, 100, 5)
(100, 100, 5, 5)
(100, 100, 5, 5)
```

Broadcasting

```
1   A = tf.random.normal((3, 2, 1, 5, 6))
2   B = tf.random.normal((3, 1, 4, 5, 1))
3   print((A + B).shape)
```

```
(3, 2, 4, 5, 6)
```

Broadcasting

Shapes are different

```
1   A = tf.random.normal((3, 2, 1, 5, 6))
2   B = tf.random.normal((3, 1, 4, 5, 1))
3   print((A + B).shape)
```

```
(3, 2, 4, 5, 6)
```

Broadcasting

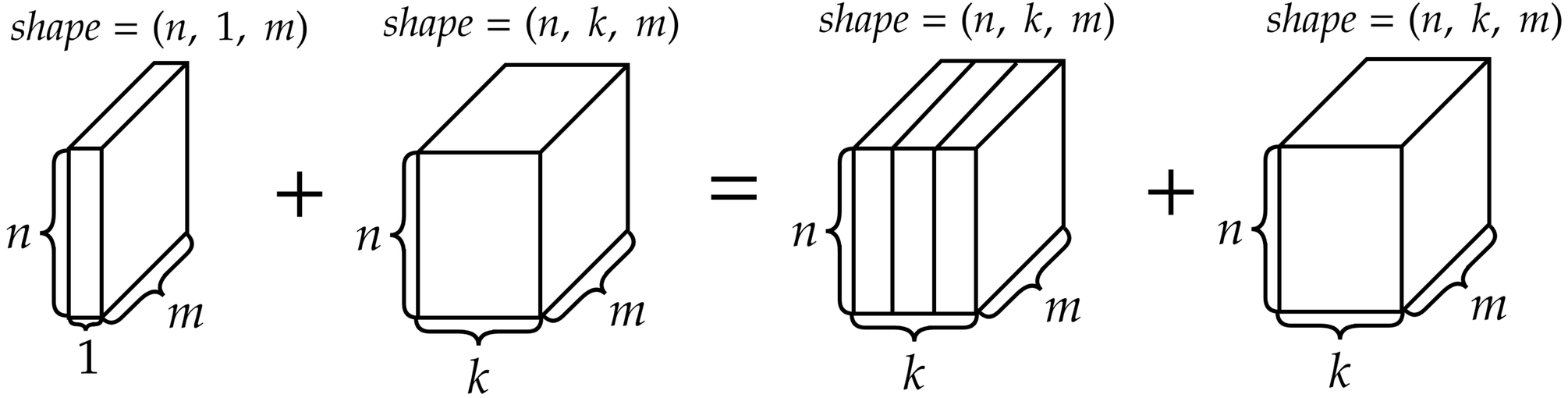
Shapes are different

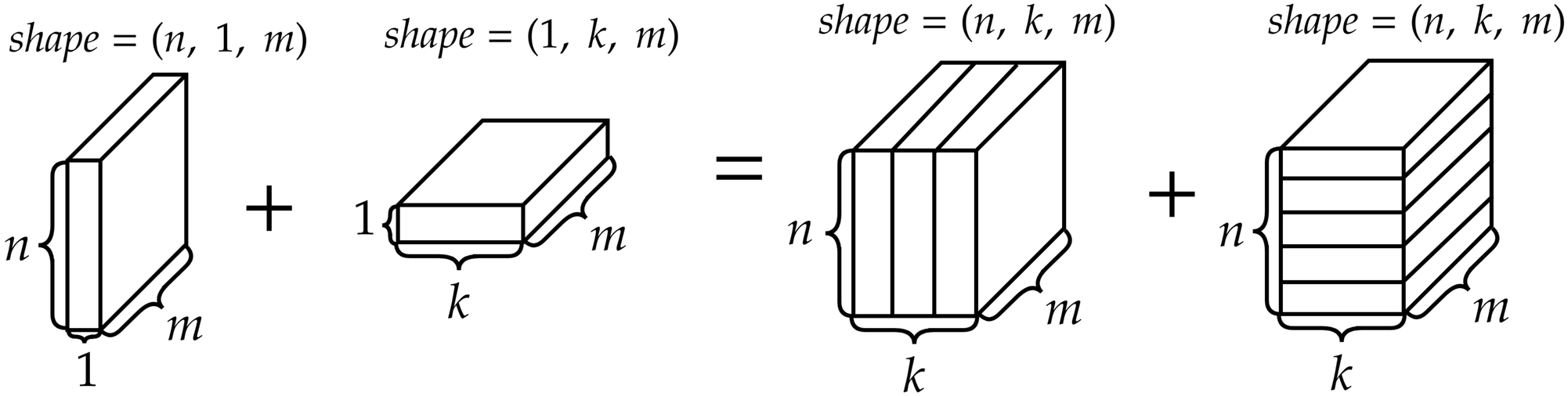
```
1   A = tf.random.normal((3, 2, 1, 5, 6))
2   B = tf.random.normal((3, 1, 4, 5, 1))
3   print((A + B).shape)
```

```
(3, 2, 4, 5, 6)
```

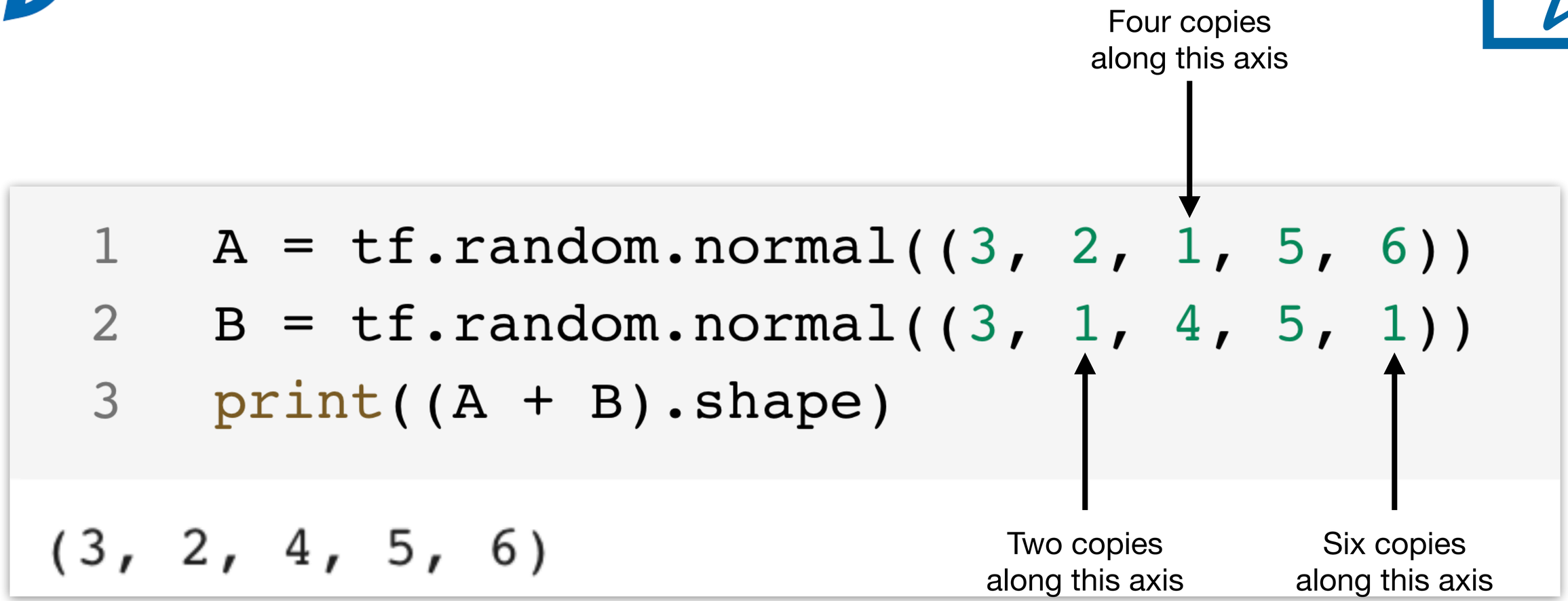
Why don't we have a mistake???

Broadcasting





Broadcasting



Tensor product of Pauli matrices

$$\sigma_x = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \sigma_y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \sigma_z = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Tensor product of Pauli matrices

$$\sigma_x = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \sigma_y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \sigma_z = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

```
# Pauli matrices
sigma_x = tf.constant([[0, 1], [1, 0]], dtype=tf.complex128)
sigma_y = tf.constant([[0 + 0j, -1j], [1j, 0 + 0j]], dtype=tf.complex128)
sigma_z = tf.constant([[1, 0], [0, -1]], dtype=tf.complex128)

# All Pauli matrices in one tensor of shape (3, 2, 2)
sigma = tf.concat([sigma_x[tf.newaxis],
                   sigma_y[tf.newaxis],
                   sigma_z[tf.newaxis]], axis=0)
```

Tensor product of Pauli matrices

$$\sigma_x = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \sigma_y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \sigma_z = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Want to calculate

$$\sigma_x \otimes \sigma_x,\ \sigma_y \otimes \sigma_y,\ \sigma_z \otimes \sigma_z$$

```
# Pauli matrices
sigma_x = tf.constant([[0, 1], [1, 0]], dtype=tf.complex128)
sigma_y = tf.constant([[0 + 0j, -1j], [1j, 0 + 0j]], dtype=tf.complex128)
sigma_z = tf.constant([[1, 0], [0, -1]], dtype=tf.complex128)

# All Pauli matrices in one tensor of shape (3, 2, 2)
sigma = tf.concat([sigma_x[tf.newaxis],
                   sigma_y[tf.newaxis],
                   sigma_z[tf.newaxis]], axis=0)
```

Tensor product of Pauli matrices

Auxiliary indices, new shape is (3, 2, 2, 1, 1)

```
left_sigma = sigma[:, :, :, tf.newaxis, tf.newaxis]
```

Tensor product of Pauli matrices

Auxiliary indices, new shape is (3, 2, 2, 1, 1)

```
1   left_sigma = sigma[:, :, :, tf.newaxis, tf.newaxis]
```

Auxiliary indices, new shape is (3, 1, 1, 2, 2)

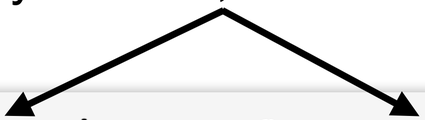
```
2   right_sigma = sigma[:, tf.newaxis, tf.newaxis]
```

Tensor product of Pauli matrices

Auxiliary indices, new shape is (3, 2, 2, 1, 1)

```
1   left_sigma = sigma[:, :, :, tf.newaxis, tf.newaxis]
```

Auxiliary indices, new shape is (3, 1, 1, 2, 2)



```
2   right_sigma = sigma[:, tf.newaxis, tf.newaxis]
```

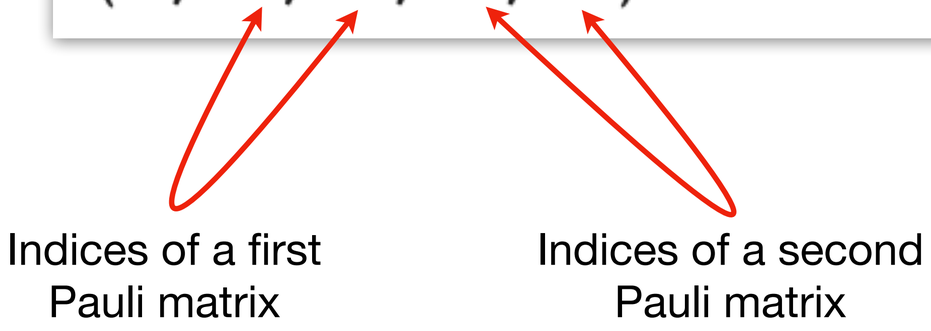
Broadcasting, new shape is (3, 2, 2, 2, 2)

```
3   tensor_prod = right_sigma * left_sigma
```

Tensor product of Pauli matrices

```
1   print(tensor_prod.shape)
```

```
(3, 2, 2, 2, 2)
```



Tensor product of Pauli matrices

```
1   print(tensor_prod.shape)
```

```
(3, 2, 2, 2, 2)
```

Let us swap these two indices

Tensor product of Pauli matrices

```
1   print(tensor_prod.shape)
```

```
(3, 2, 2, 2, 2)
```

Let us swap these two indices

```
1   tensor_prod = tf.transpose(tensor_prod, (0, 1, 3, 2, 4))
```

Tensor product of Pauli matrices

```
1   print(tensor_prod.shape)
```

```
(3, 2, 2, 2, 2)
```

Let us swap these two indices

```
1   tensor_prod = tf.transpose(tensor_prod, (0, 1, 3, 2, 4))
```

Tensor product of Pauli matrices

```
1   print(tensor_prod.shape)
```

```
(3, 2, 2, 2, 2)
```

Let us swap these two indices

```
1   tensor_prod = tf.transpose(tensor_prod, (0, 1, 3, 2, 4))
```

Reshape to a set of matrices

```
1   tensor_prod = tf.reshape(tensor_prod, (3, 4, 4))
```

Tensor product of Pauli matrices

```
1  print(tensor_prod[0])
```

```
tf.Tensor(
[[0.+0.j 0.+0.j 0.+0.j 1.+0.j]
 [0.+0.j 0.+0.j 1.+0.j 0.+0.j]
 [0.+0.j 1.+0.j 0.+0.j 0.+0.j]
 [1.+0.j 0.+0.j 0.+0.j 0.+0.j]], shape=(4, 4), dtype=complex128)
```

$$= \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} = \sigma_x \otimes \sigma_x$$

```
1  print(tensor_prod[1])
```

```
tf.Tensor(
[[ 0.+0.j  0.-0.j  0.-0.j -1.+0.j]
 [ 0.+0.j  0.+0.j  1.-0.j  0.-0.j]
 [ 0.+0.j  1.-0.j  0.+0.j  0.-0.j]
 [-1.+0.j  0.+0.j  0.+0.j  0.+0.j]], shape=(4, 4), dtype=complex128)
```

$$= \begin{bmatrix} 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \end{bmatrix} = \sigma_y \otimes \sigma_y$$

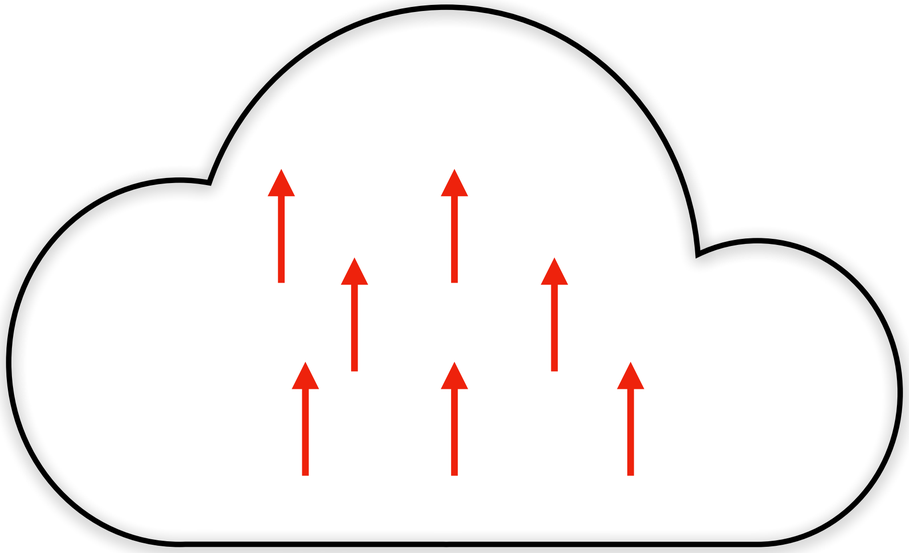
```
1  print(tensor_prod[2])
```

```
tf.Tensor(
[[ 1.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j -1.+0.j  0.+0.j -0.+0.j]
 [ 0.+0.j  0.+0.j -1.+0.j -0.+0.j]
 [ 0.+0.j -0.+0.j -0.+0.j  1.-0.j]], shape=(4, 4), dtype=complex128)
```

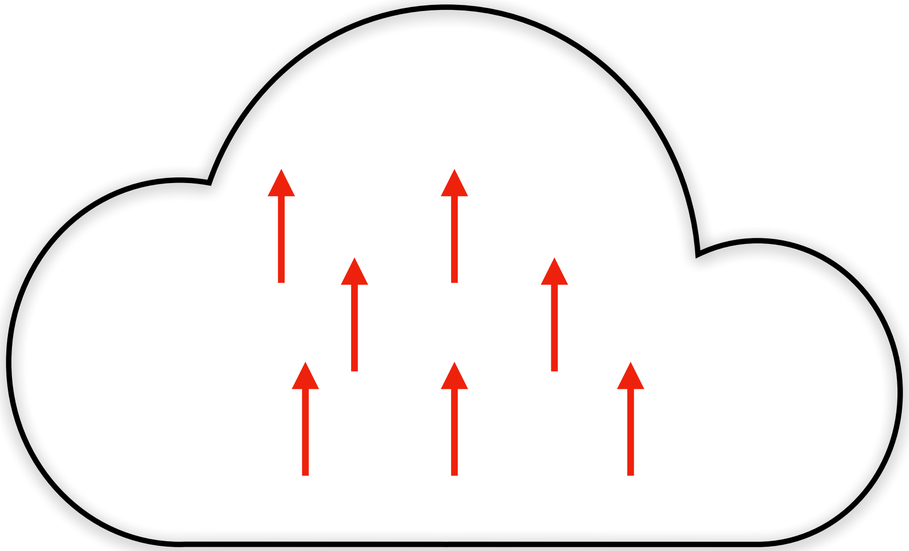
$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \sigma_z \otimes \sigma_z$$

Spin echo

Spin echo

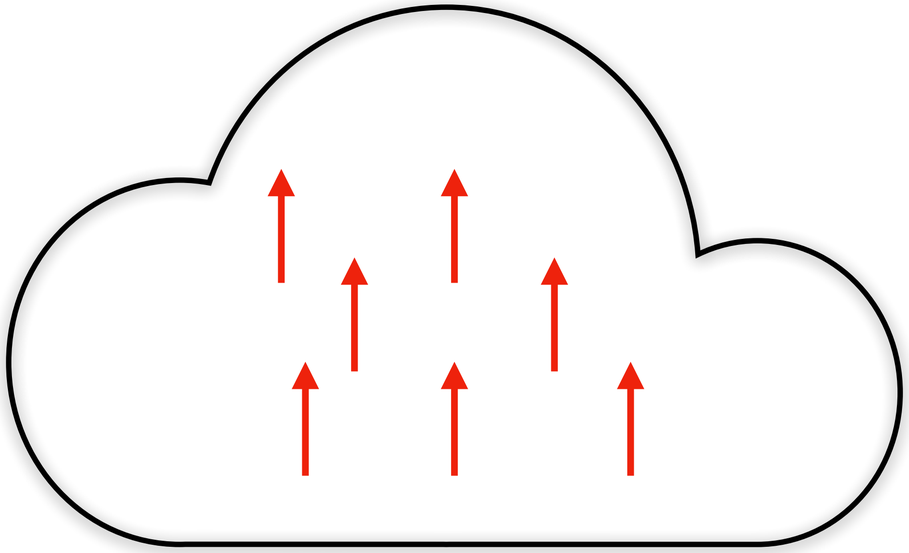


Spin echo



$$|\psi_{\text{in}}\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

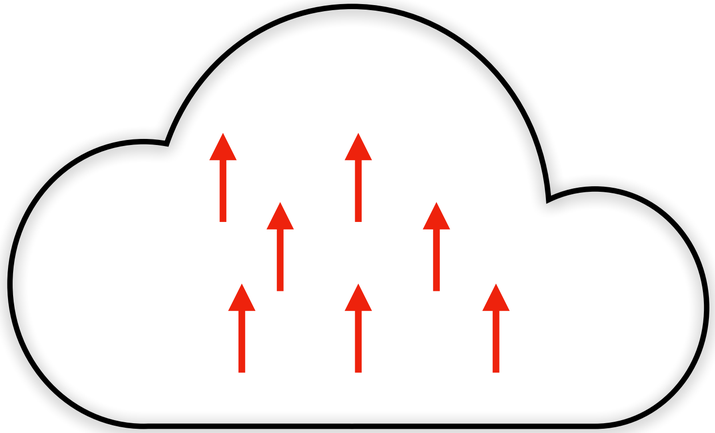
Spin echo



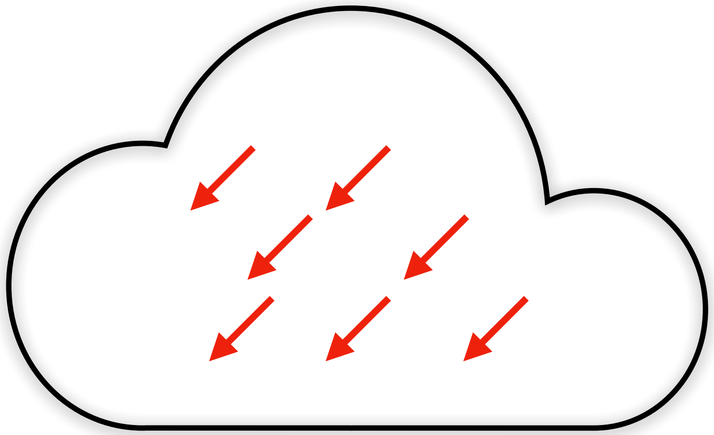
$$|\psi_{\text{in}}\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

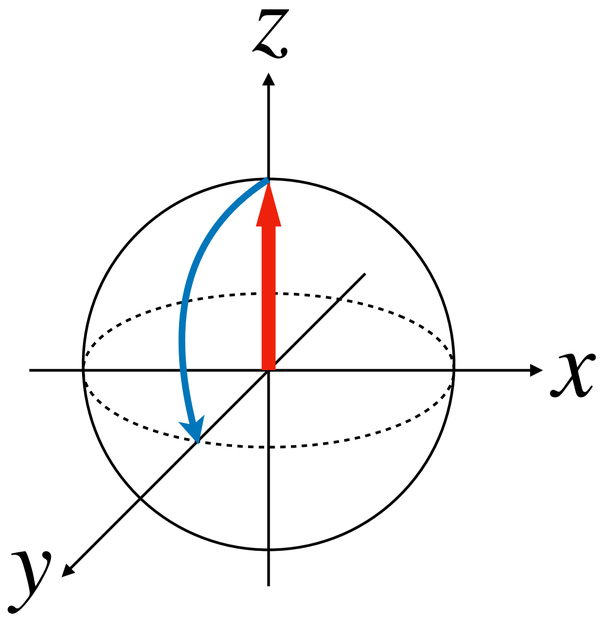
$$H = \epsilon\sigma_z,\ \epsilon \sim N(0,1)$$

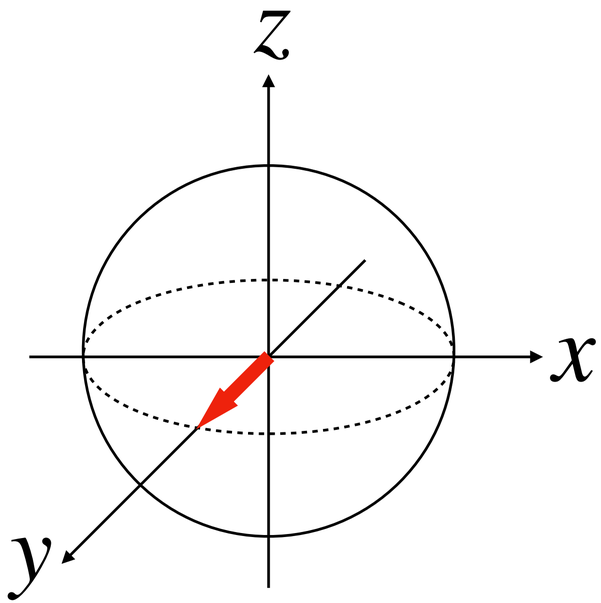
Spin echo



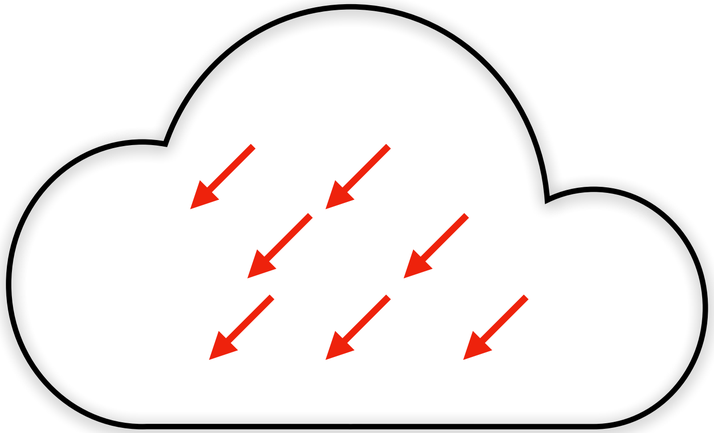
$$H_c = \epsilon\sigma_z - \frac{\pi}{4T_c}\sigma_x$$



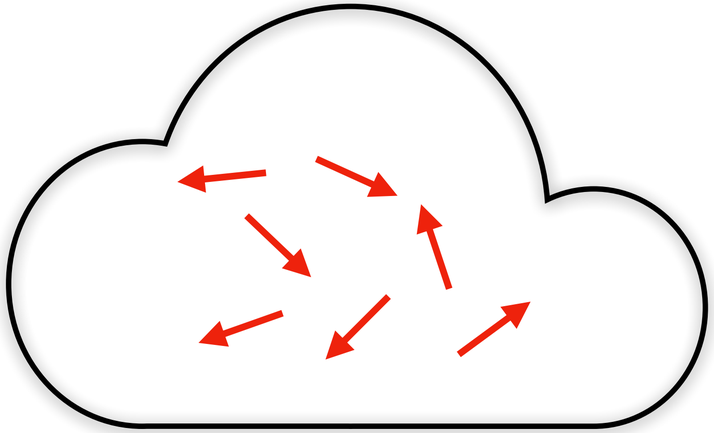


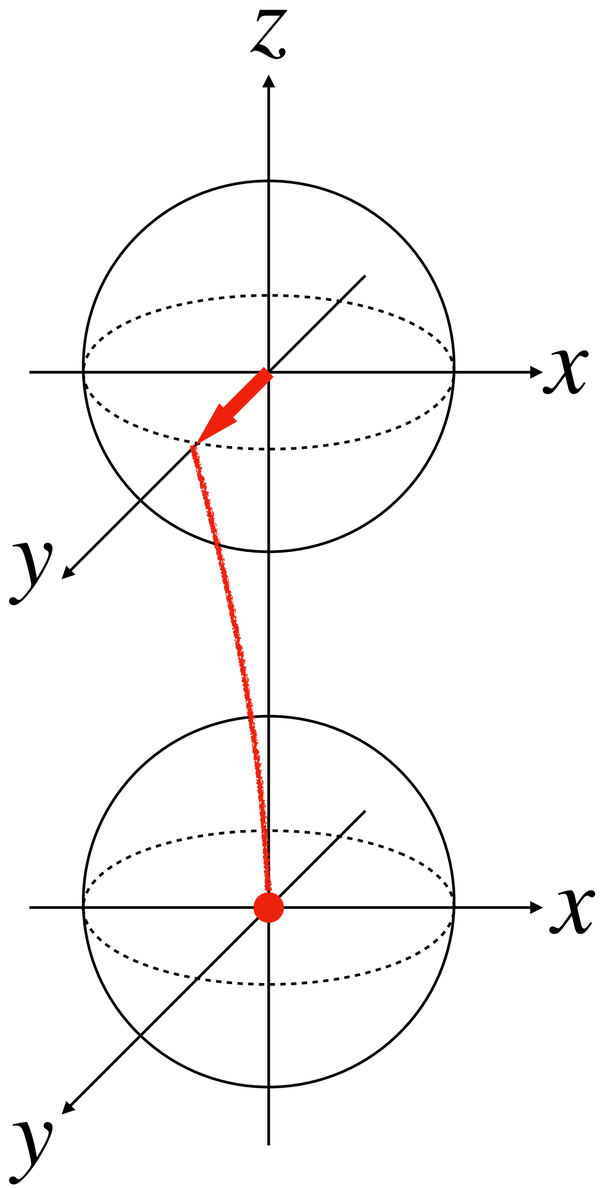


Spin echo

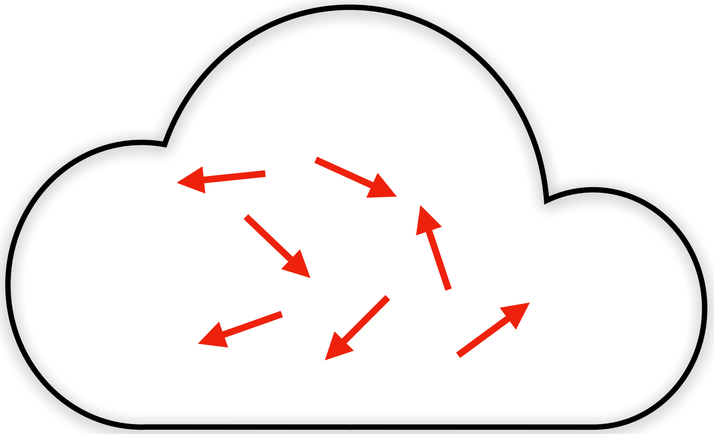


$$H_c = \epsilon \sigma_z$$



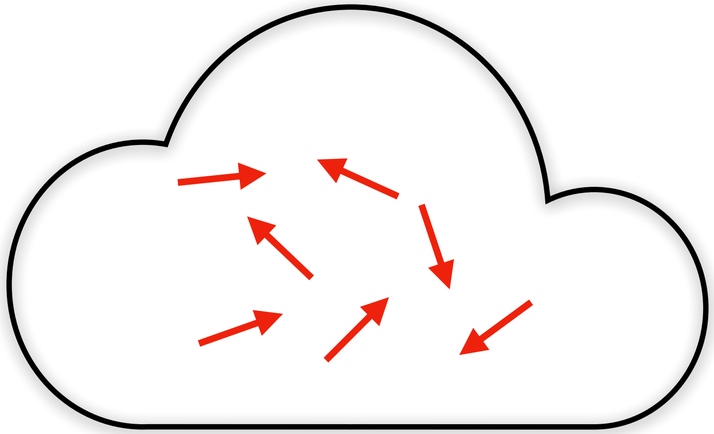


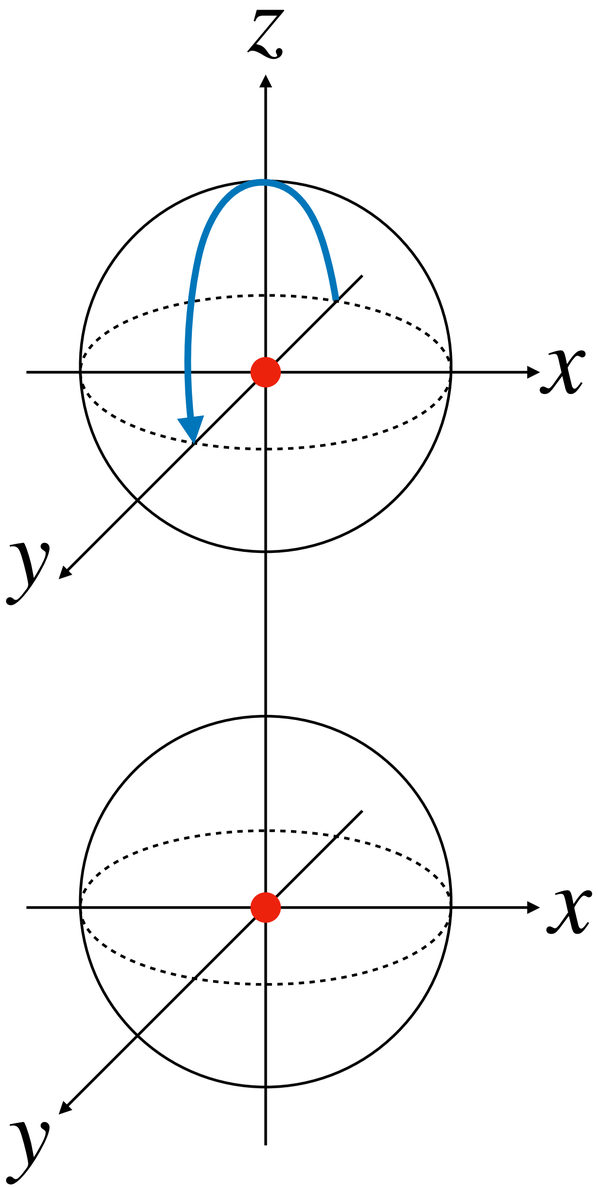
Spin echo



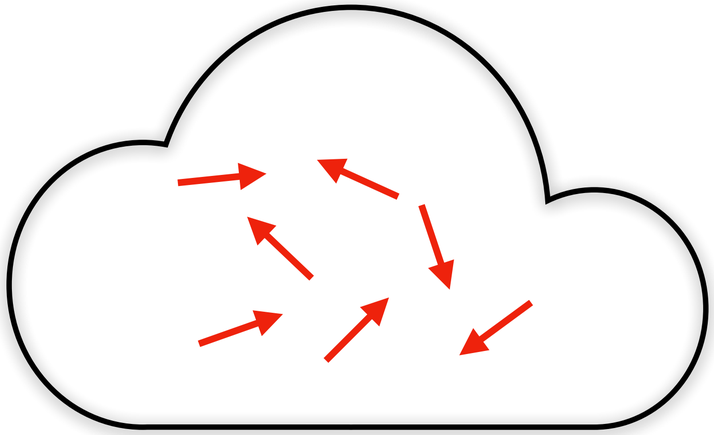
$$H_c = \epsilon\sigma_z - \frac{\pi}{2T_c}\sigma_x$$

Vectors were reflected

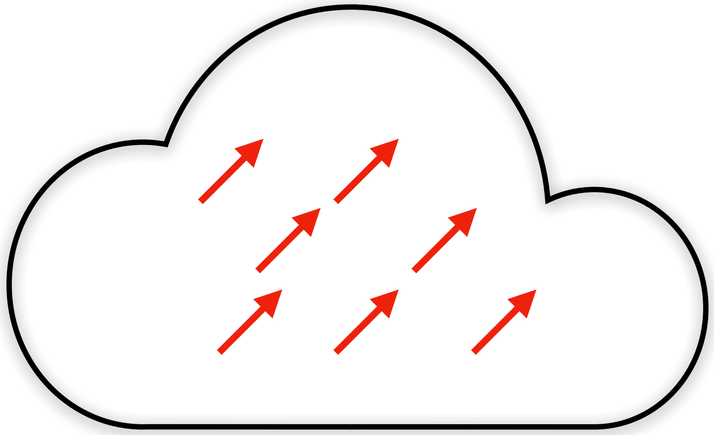


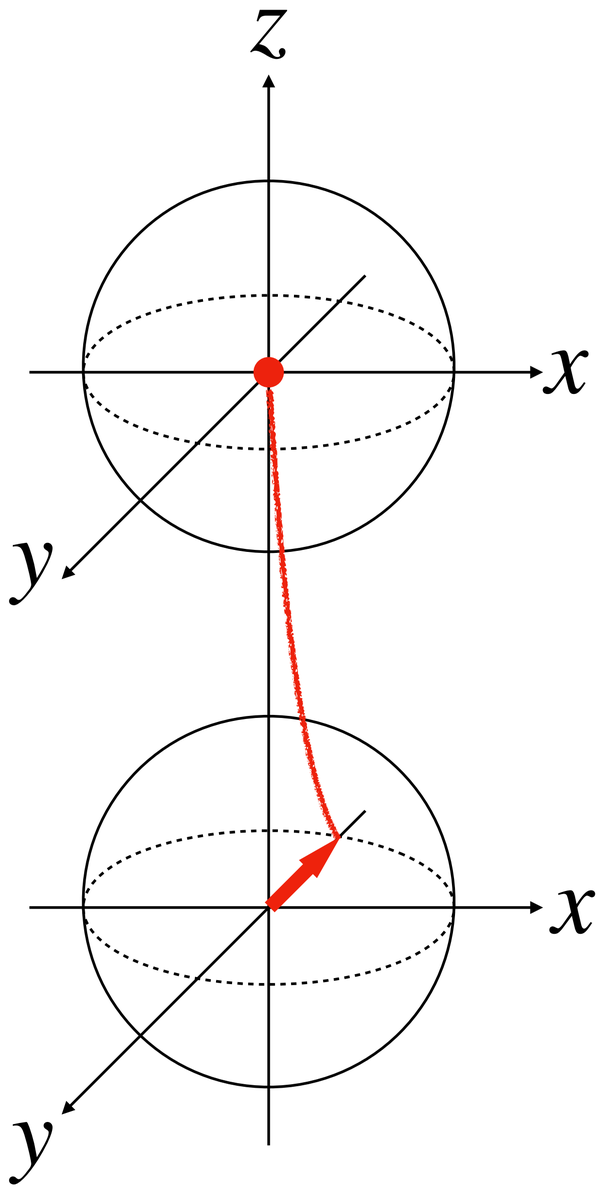


Spin echo

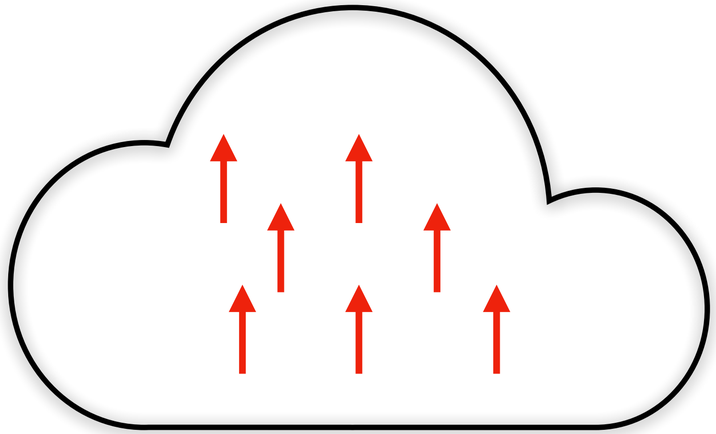


$$H_c = \epsilon \sigma_z$$





Spin echo simulation

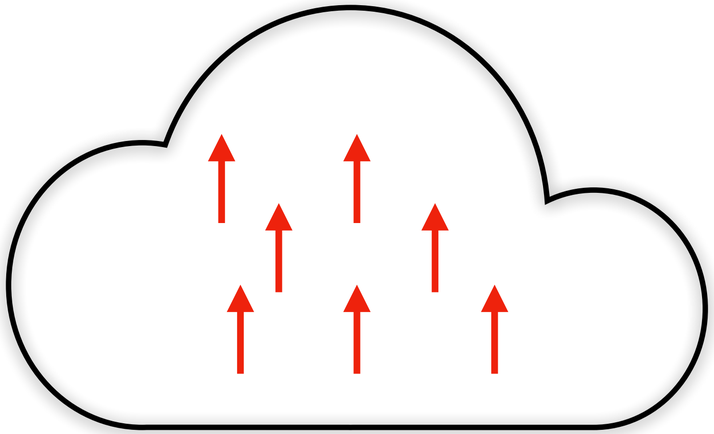


Initial density matrices

```
1   rho = tf.constant([[1, 0], [0, 0]], dtype=tf.complex128)
2   rho = rho[tf.newaxis]
3   rho = tf.tile(rho, (10000, 1, 1))
```

Last function makes 10000 copies of density matrix and stack them together. The resulting tensor has shape (10000, 2, 2)

Spin echo simulation

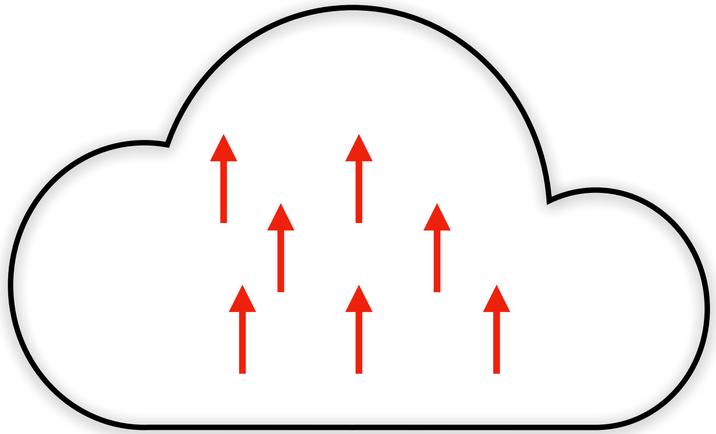


Disordered Hamiltonians

```
1   H = tf.constant([[0.5, 0], [0, -0.5]], dtype=tf.complex128)
2   delta = tf.random.normal((N, 1, 1))
3   delta = tf.cast(delta, dtype=tf.complex128)
4   H = H * delta
```

Resulting tensor has shape (10000, 2, 2)

Spin echo simulation



Disordered Hamiltonians

```
1   H = tf.constant([[0.5, 0], [0, -0.5]], dtype=tf.complex128)
2   delta = tf.random.normal((N, 1, 1))
3   delta = tf.cast(delta, dtype=tf.complex128)
4   H = H * delta
```

Resulting tensor has shape (10000, 2, 2)

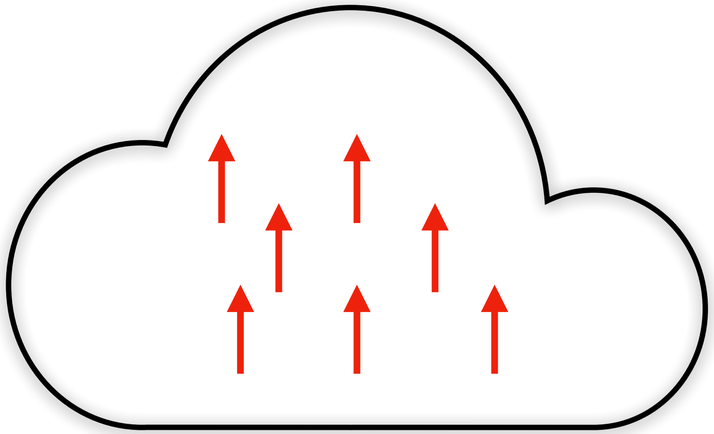
Corresponding unitary transformations

Time step

```
1   U = tf.linalg.expm(-1j * H * 0.1)
```

Resulting tensor has shape (10000, 2, 2)

Spin echo simulation

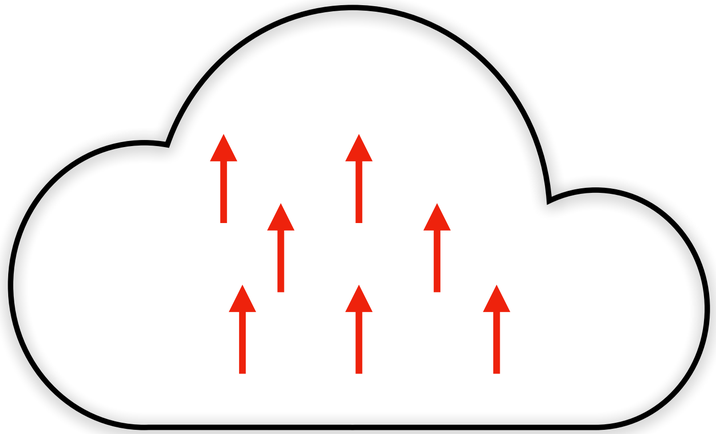


Disordered Hamiltonians with control

```
1   H_c = H - 0.5 * math.pi * sigma_x
```

Resulting tensor has shape (10000, 2, 2)

Spin echo simulation



Disordered Hamiltonians with control

```
1   H_c = H - 0.5 * math.pi * sigma_x
```

Resulting tensor has shape (10000, 2, 2)

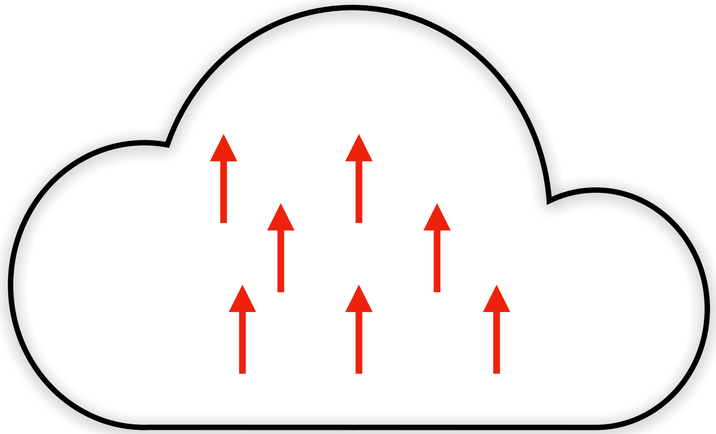
Corresponding unitary transformations

Time step

```
1   U_c = tf.linalg.expm(-1j * H_c * 0.1)
```

Resulting tensor has shape (10000, 2, 2)

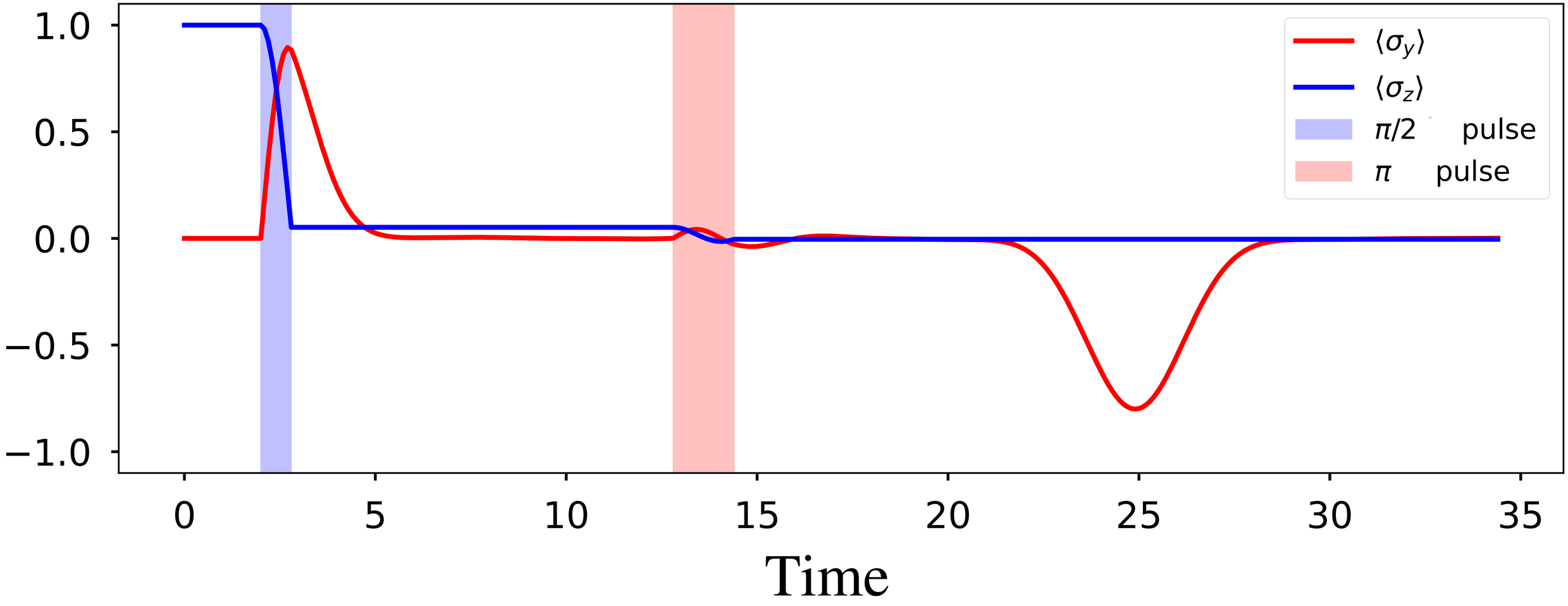
Spin echo simulation



Dynamics of whole ensemble

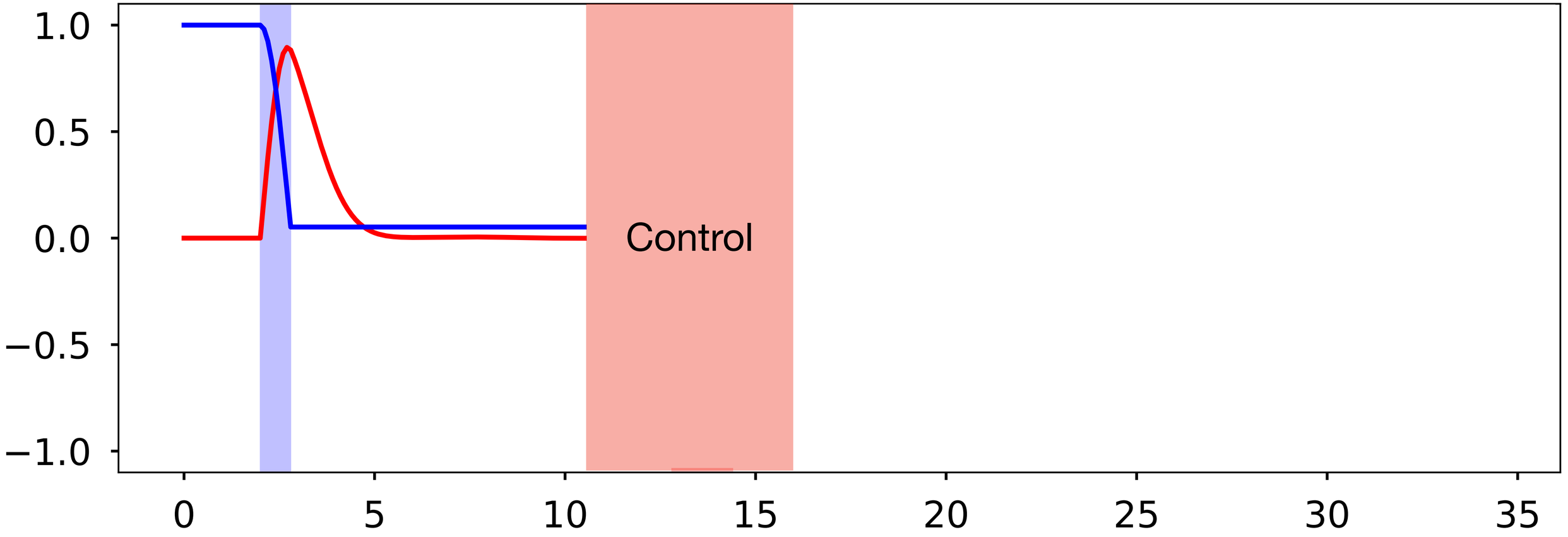
```
rho = U_c @ rho @ tf.linalg.adjoint(U_c)
```

Spin echo simulation



Spin echo optimization

$$H_c(t) = \epsilon\sigma_z + A(t)\sigma_x$$



Spin echo optimization

Discritization of $A(t)$

```
1   control = tf.Variable(tf.random.normal((30,), dtype=tf.float64))
```

The value of tf.Varianle can be changed using one of the assign methods

Number of time steps

Spin echo optimization

Discritization of $A(t)$

```
control = tf.Variable(tf.random.normal((30,), dtype=tf.float64))
```

The value of tf.Varianle can be changed using one of the assign methods

Number of time steps

```
gated_control = tf.nn.tanh(control)
```

Restrict control on the interval $(-1, 1)$

Spin echo optimization

```
complex_gated_control = tf.cast(gated_control, dtype=tf.complex128)
```

Spin echo optimization

```
complex_gated_control = tf.cast(gated_control, dtype=tf.complex128)
```

```
H_c = tf.tensordot(complex_gated_control, sigma_x, axes=0)
```

Control Hamiltonian of shape
(30, 2, 2)

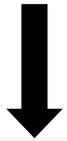
Spin echo optimization

```
1   H_full = H[:, tf.newaxis] + H_c
```

Full Hamiltonian of shape
(10000, 30, 2, 2)

Spin echo optimization

```
control
```



```
with tf.GradientTape() as tape:
```

Dynamics calculation

```
last_y
```

Spin echo optimization

```
control
```

```
with tf.GradientTape() as tape:
```

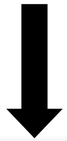
Dynamics calculation

```
last_y
```

```
loss = L1 + last_y
```

Spin echo optimization

```
control
```



```
with tf.GradientTape() as tape:
```

Dynamics calculation

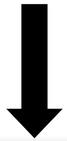
```
last_y
```

```
loss = L1 + last_y
```

```
opt = tf.optimizers.Adam(0.1)
```

Spin echo optimization

```
control
```



```
with tf.GradientTape() as tape:
```

Dynamics calculation

```
last_y
```

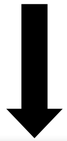
```
loss = L1 + last_y
```

```
opt = tf.optimizers.Adam(0.1)
```

```
grad = tape.gradient(loss, control)
```

Spin echo optimization

```
control
```



```
with tf.GradientTape() as tape:
```

Dynamics calculation

```
last_y
```

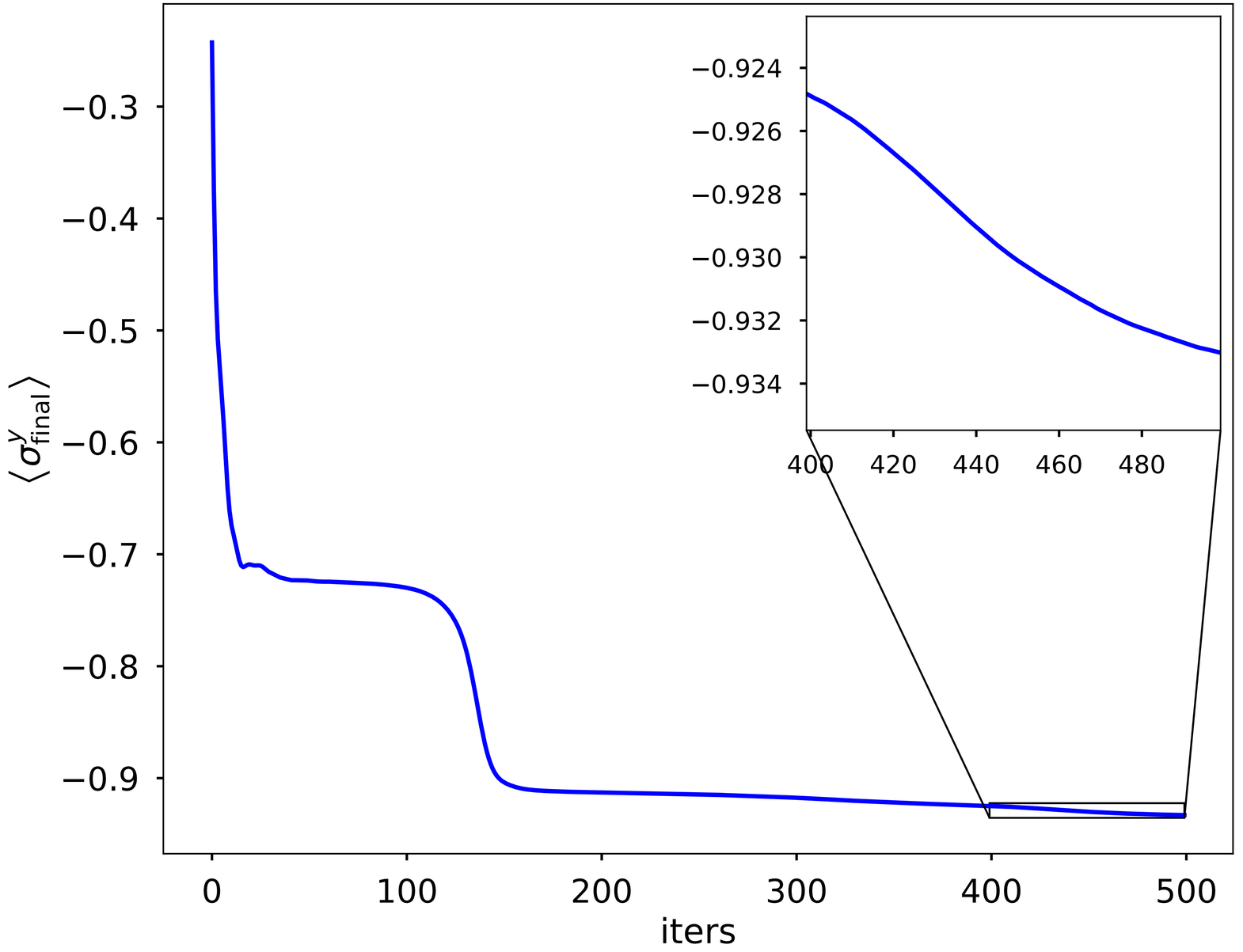
```
loss = L1 + last_y
```

```
opt = tf.optimizers.Adam(0.1)
```

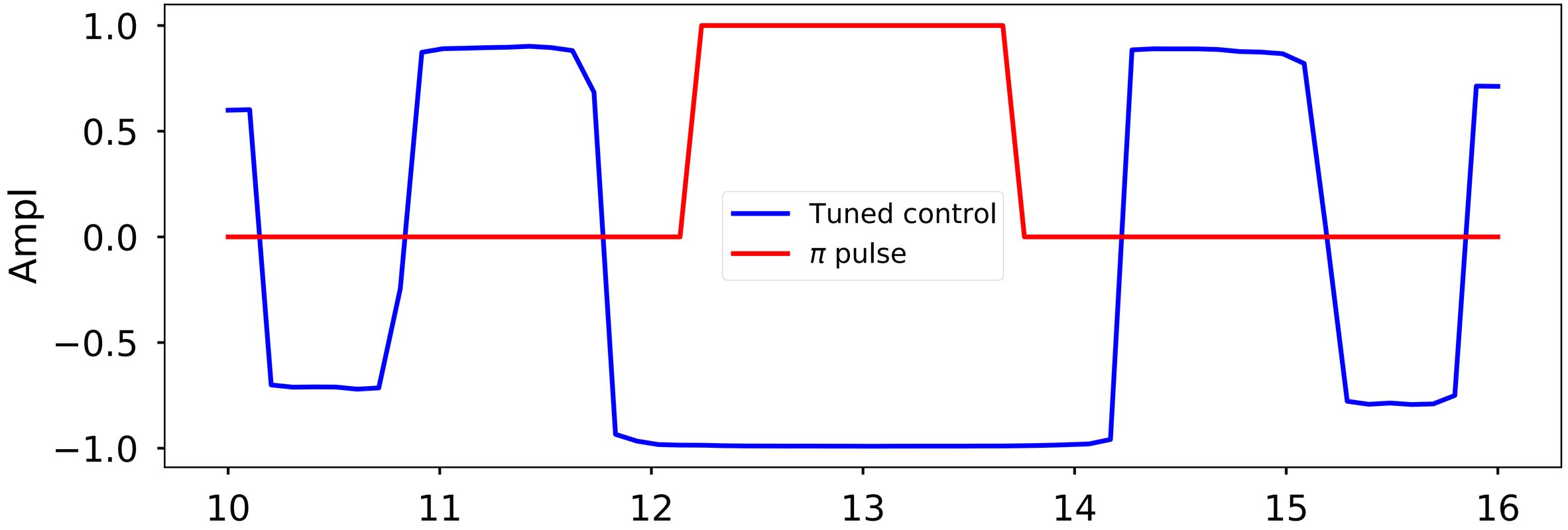
```
grad = tape.gradient(loss, control)
```

```
opt.apply_gradients(zip([grad], [control]))
```

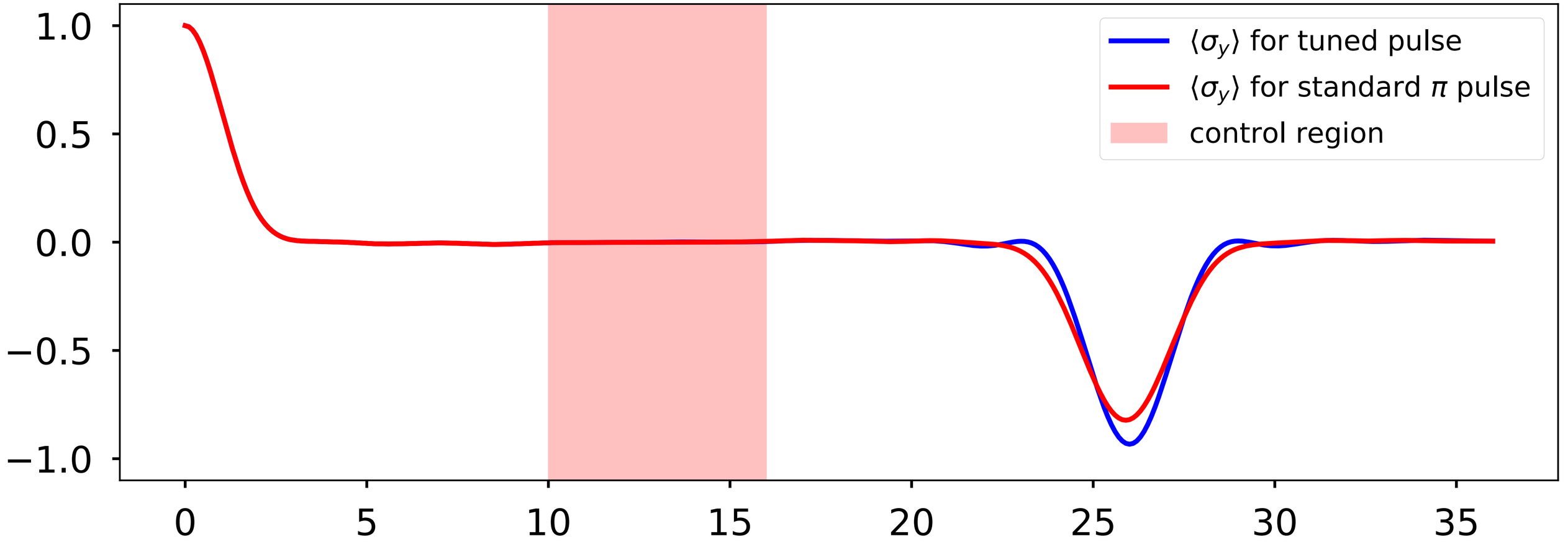
Spin echo optimization



Spin echo optimization



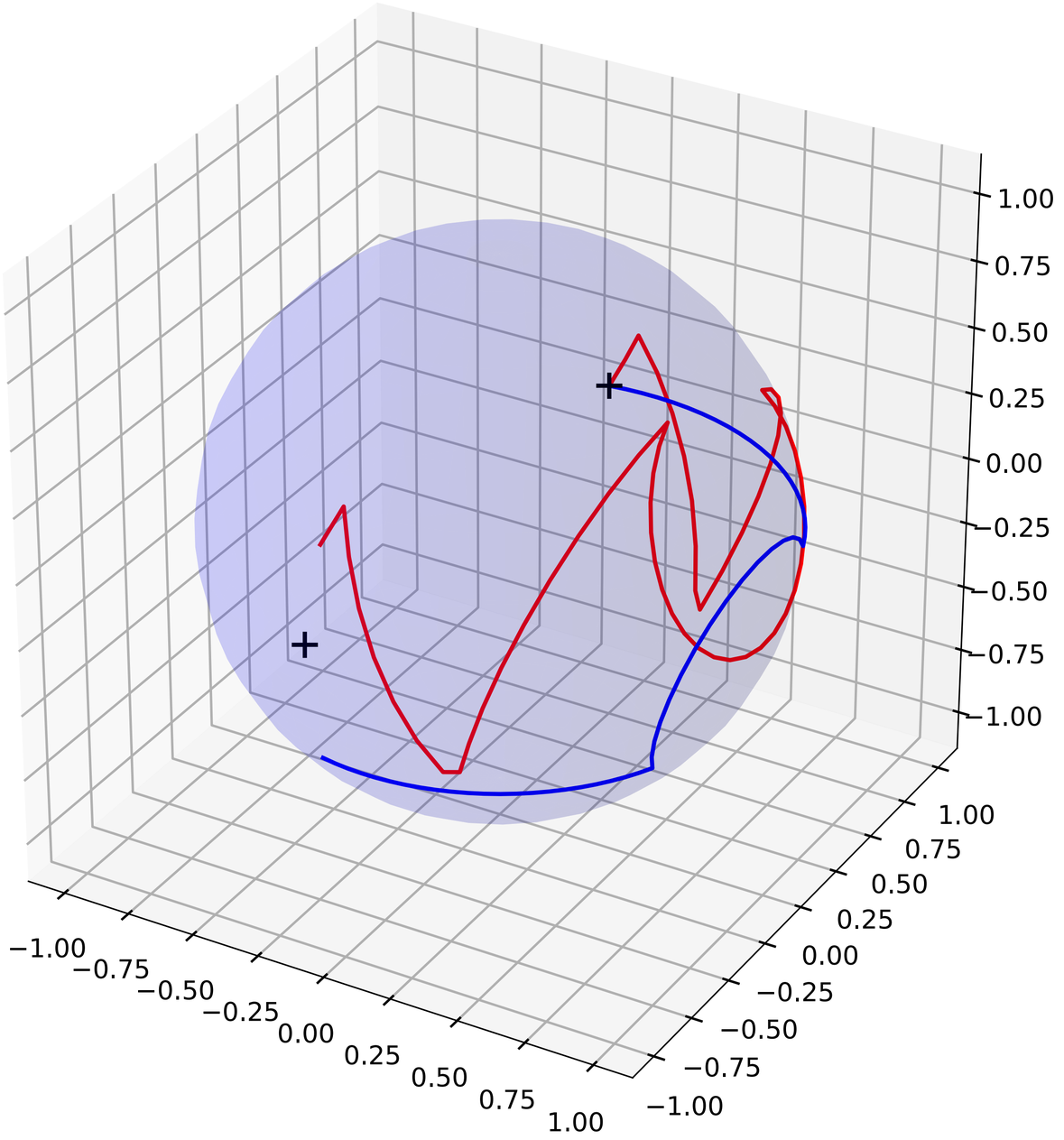
Spin echo optimization



Spin echo optimization

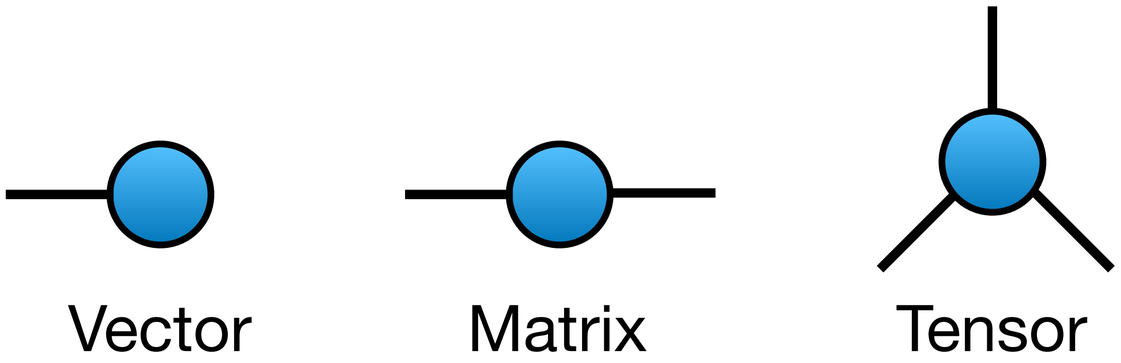
Optimized control

π pulse

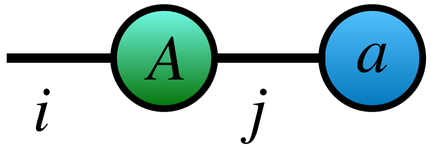


AD based optimization of a matrix product state

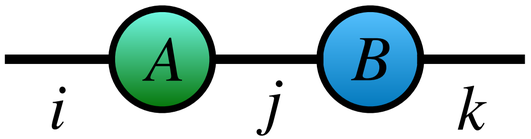
Penrose graphical notation



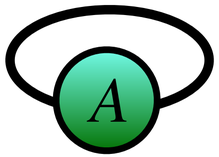
$$\sum_j A_{ij} a_j =$$



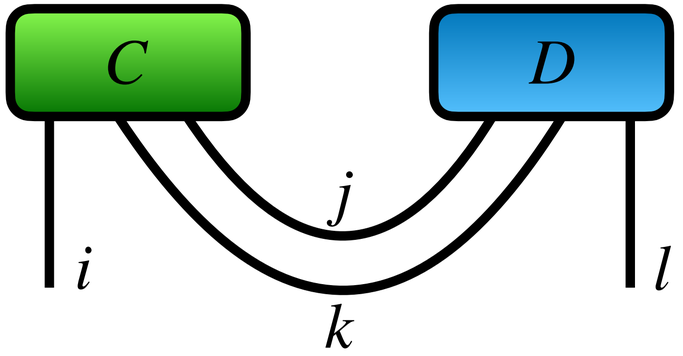
$$\sum_j A_{ij} B_{jk} =$$



$$\mathrm{Tr}(A) =$$



$$\sum_{jk} C_{ijk} D_{ljk} =$$



Matrix product state

$$|\psi\rangle = \sum_{i_1,i_2,\ldots,i_n} \psi_{i_1,i_2,\ldots,i_n} |i_1\rangle \otimes |i_2\rangle \otimes \ldots \otimes |i_n\rangle$$

Matrix product state

$$|\psi\rangle = \sum_{i_1,i_2,\ldots,i_n} \psi_{i_1,i_2,\ldots,i_n} |i_1\rangle \otimes |i_2\rangle \otimes \ldots \otimes |i_n\rangle$$

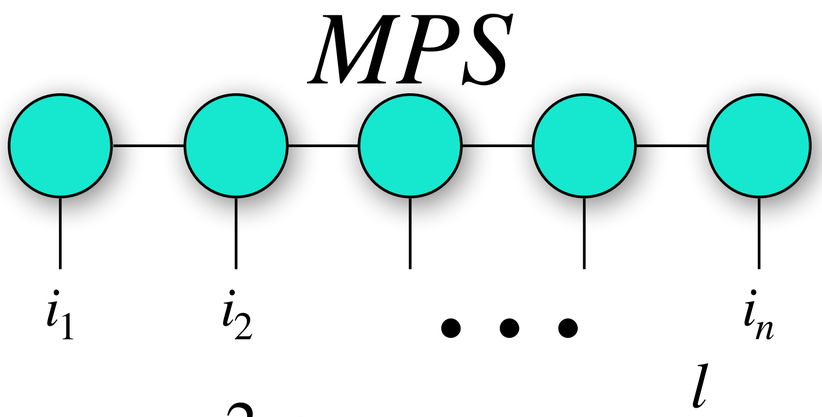
d^n coefficients

Matrix product state

$$|\psi\rangle = \sum_{i_1,i_2,\ldots,i_n} \psi_{i_1,i_2,\ldots,i_n} |i_1\rangle \otimes |i_2\rangle \otimes \ldots \otimes |i_n\rangle$$

d^n coefficients

$$\psi_{i_1,i_2,\ldots,i_n} =$$



$\approx nr^2d$ coefficients

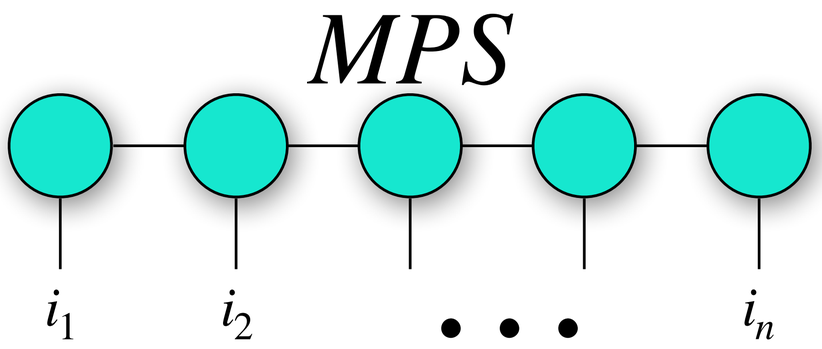
Matrix product state

$$|\psi\rangle = \sum_{i_1,i_2,\ldots,i_n} \psi_{i_1,i_2,\ldots,i_n} |i_1\rangle \otimes |i_2\rangle \otimes \ldots \otimes |i_n\rangle$$

d^n coefficients

Orús R. A practical introduction to tensor networks: Matrix product states and projected entangled pair states. Annals of Physics. 2014 Oct 1;349:117-58.

$$\psi_{i_1,i_2,\ldots,i_n} =$$



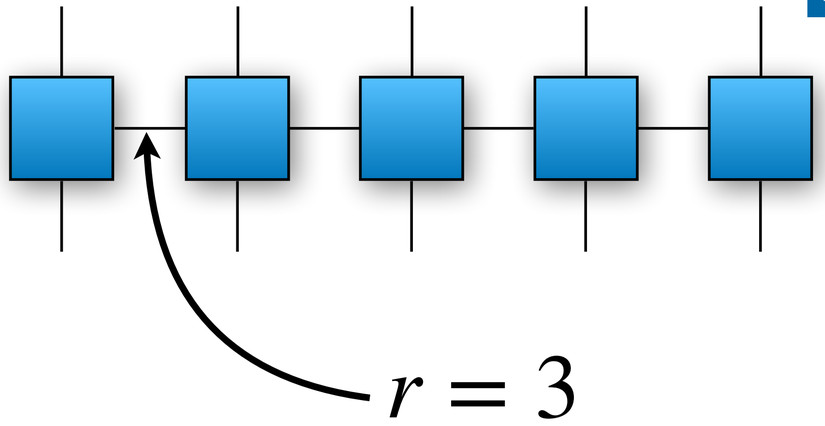
$\approx nr^2d$ coefficients

Ground state of TFI hamiltonian

$$H = \sum_{i=1}^{N-1} \sigma_z^i \sigma_z^{i+1} + \sum_{i=1}^{N} h_x \sigma_x^i + \sum_{i=1}^{N} h_z \sigma_z^i$$

Ground state of TFI hamiltonian

$$H = \sum_{i=1}^{N-1} \sigma_z^i \sigma_z^{i+1} + \sum_{i=1}^{N} h_x \sigma_x^i + \sum_{i=1}^{N} h_z \sigma_z^i =$$

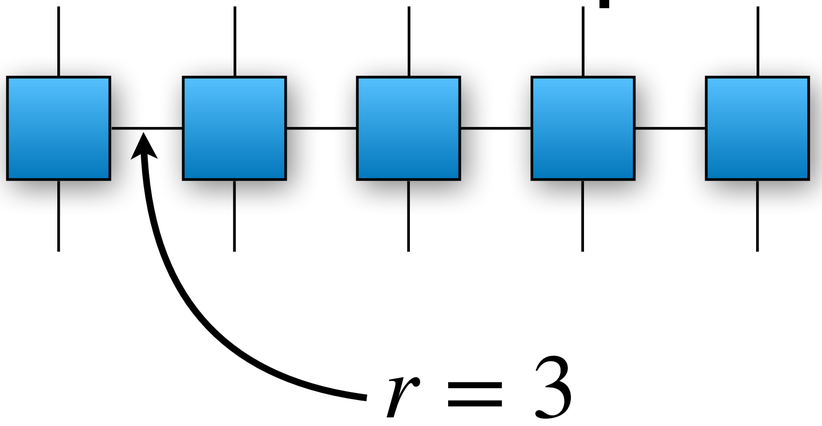


Schollwöck U. The density-matrix renormalization group in the age of matrix product states. Annals of physics. 2011 Jan 1;326(1):96-192.

Ground state of TFI hamiltonian

$$H = \sum_{i=1}^{N-1} \sigma_z^i \sigma_z^{i+1} + \sum_{i=1}^{N} h_x \sigma_x^i + \sum_{i=1}^{N} h_z \sigma_z^i =$$

Hamiltonian in MPO representation

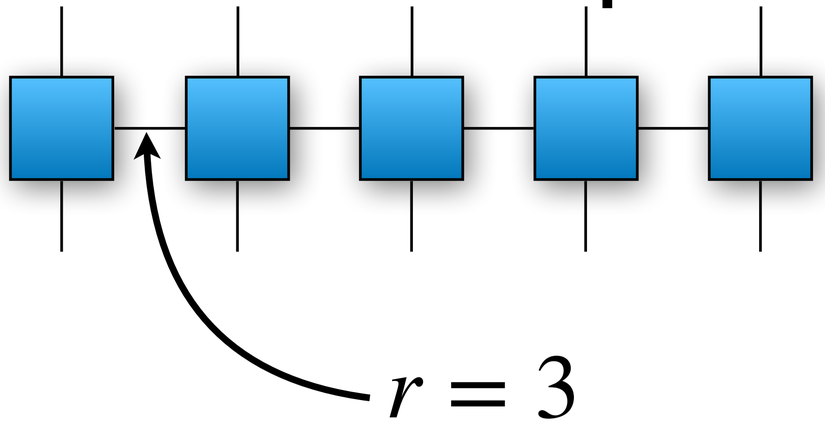


Schollwöck U. The density-matrix renormalization group in the age of matrix product states. Annals of physics. 2011 Jan 1;326(1):96-192.

Ground state of TFI hamiltonian

$$H = \sum_{i=1}^{N-1} \sigma_z^i \sigma_z^{i+1} + \sum_{i=1}^{N} h_x \sigma_x^i + \sum_{i=1}^{N} h_z \sigma_z^i =$$

Hamiltonian in MPO representation



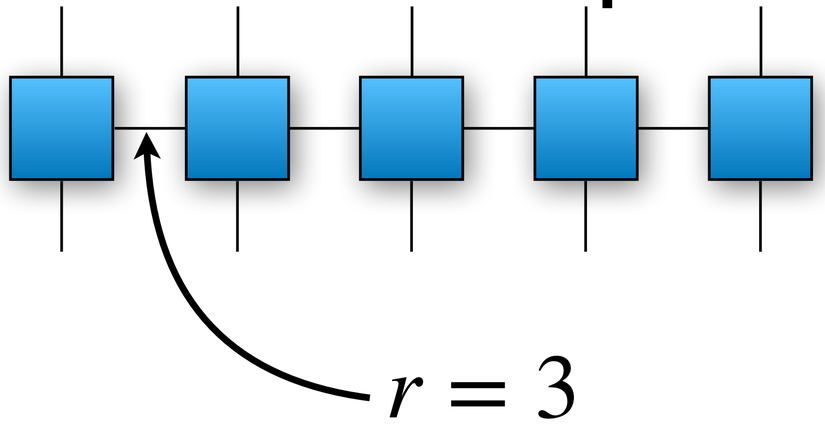
Schollwöck U. The density-matrix renormalization group in the age of matrix product states. Annals of physics. 2011 Jan 1;326(1):96-192.

$$E = \frac{\langle \text{mps} \,|\, H \,|\, \text{mps} \rangle}{\langle \text{mps} \,|\, \text{mps} \rangle}$$

Ground state of TFI hamiltonian

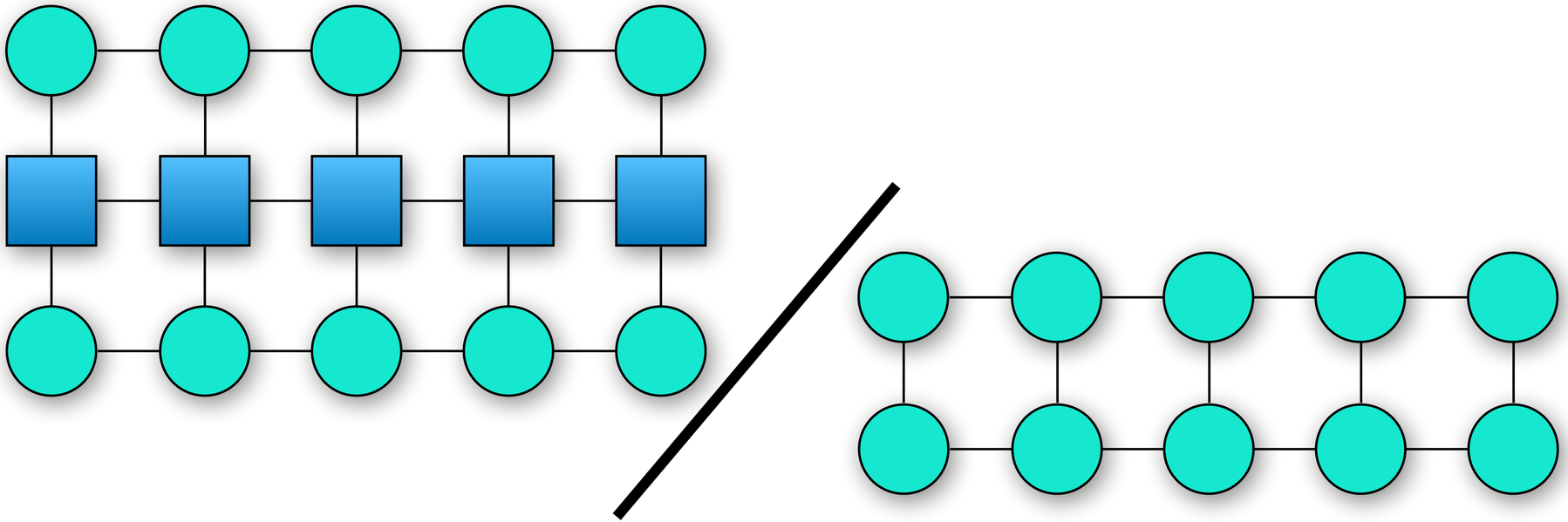
$$H = \sum_{i=1}^{N-1} \sigma_z^i \sigma_z^{i+1} + \sum_{i=1}^{N} h_x \sigma_x^i + \sum_{i=1}^{N} h_z \sigma_z^i =$$

Hamiltonian in MPO representation



Schollwöck U. The density-matrix renormalization group in the age of matrix product states. Annals of physics. 2011 Jan 1;326(1):96-192.

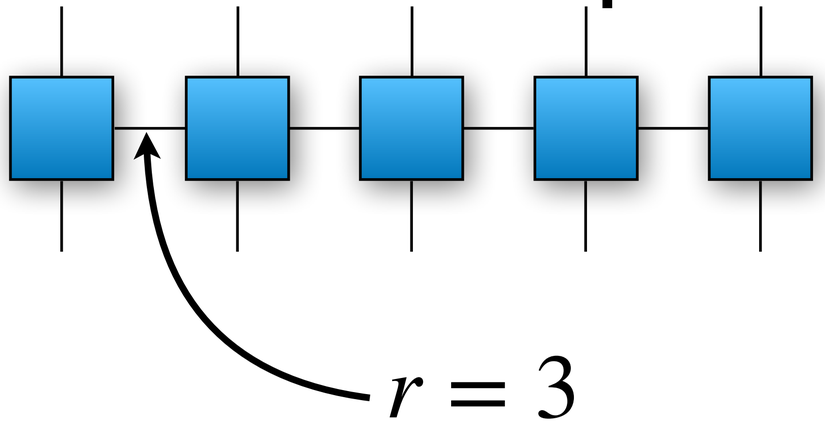
$$E = \frac{\langle \text{mps} \,|\, H \,|\, \text{mps} \rangle}{\langle \text{mps} \,|\, \text{mps} \rangle} =$$



Ground state of TFI hamiltonian

$$H = \sum_{i=1}^{N-1} \sigma_z^i \sigma_z^{i+1} + \sum_{i=1}^{N} h_x \sigma_x^i + \sum_{i=1}^{N} h_z \sigma_z^i =$$

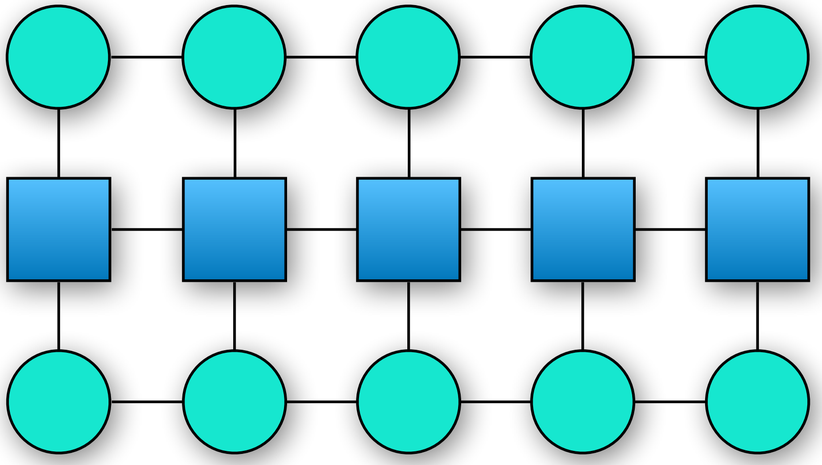
Hamiltonian in MPO representation



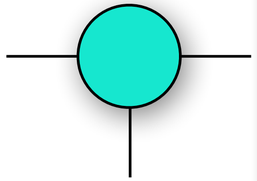
Schollwöck U. The density-matrix renormalization group in the age of matrix product states. Annals of physics. 2011 Jan 1;326(1):96-192.

$$|\Omega\rangle \approx \underset{|\text{mps}\rangle}{\text{argmin}} \frac{\langle \text{mps} | H | \text{mps} \rangle}{\langle \text{mps} | \text{mps} \rangle}$$

Programming of tensor networks

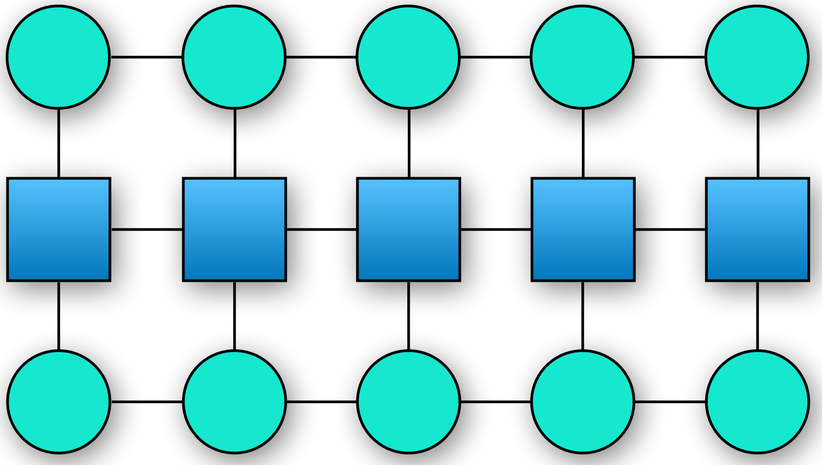


Programming of tensor networks

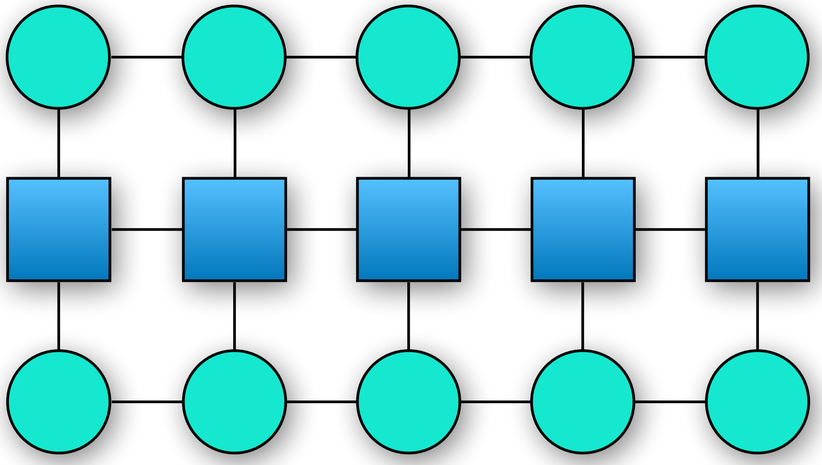


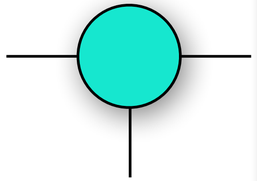
```
1   print(A.shape)
2   print(A.dtype)
```

```
(28, 2, 28)
<dtype: 'complex128'>
```



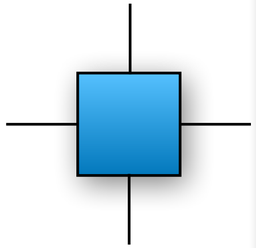
Programming of tensor networks





```
1   print(A.shape)
2   print(A.dtype)
```

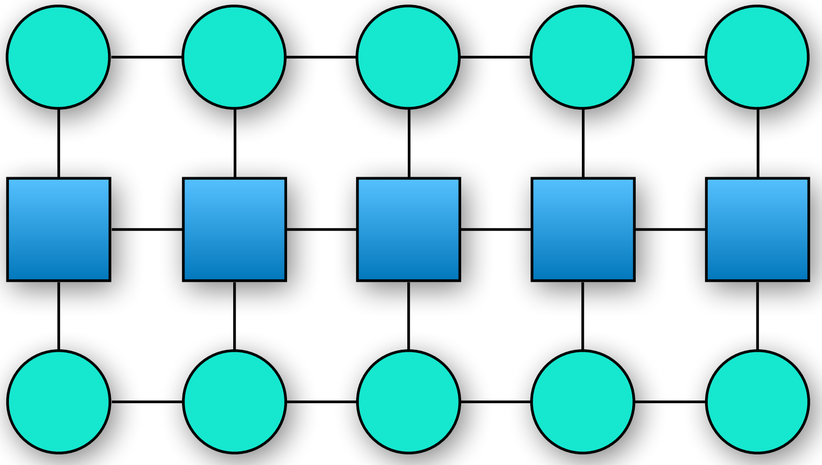
```
(28, 2, 28)
<dtype: 'complex128'>
```

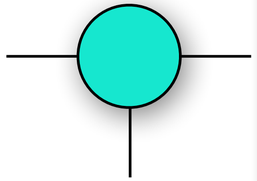


```
1   print(H.shape)
2   print(H.dtype)
```

```
(3, 2, 3, 2)
<dtype: 'complex128'>
```

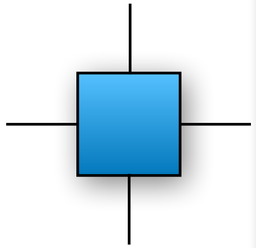
Programming of tensor networks





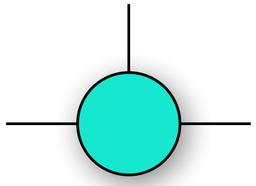
```
1   print(A.shape)
2   print(A.dtype)
```

```
(28, 2, 28)
<dtype: 'complex128'>
```



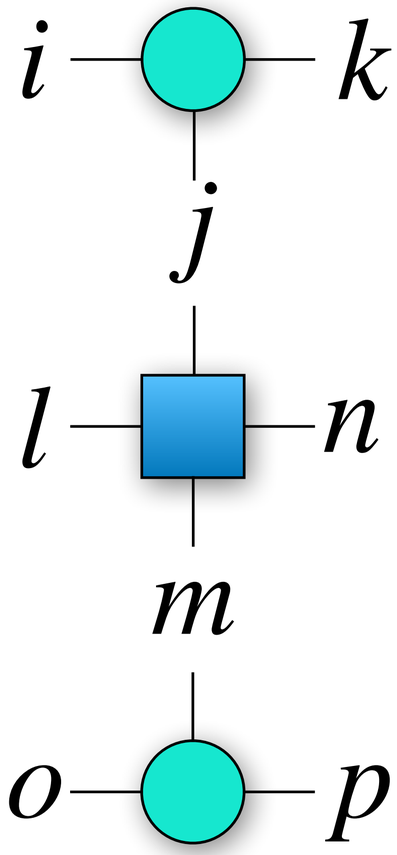
```
1   print(H.shape)
2   print(H.dtype)
```

```
(3, 2, 3, 2)
<dtype: 'complex128'>
```

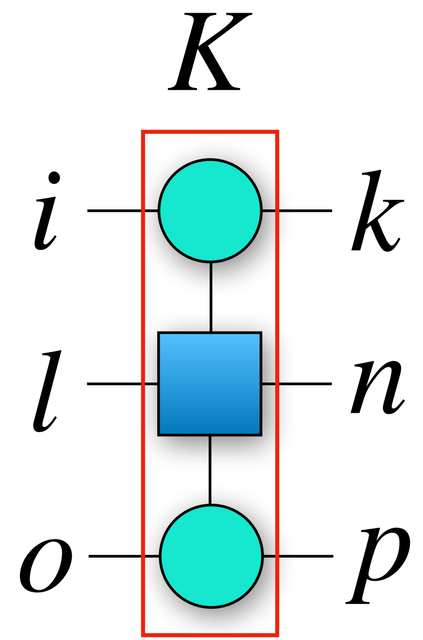


```
1   A_conj = tf.math.conj(A)
```

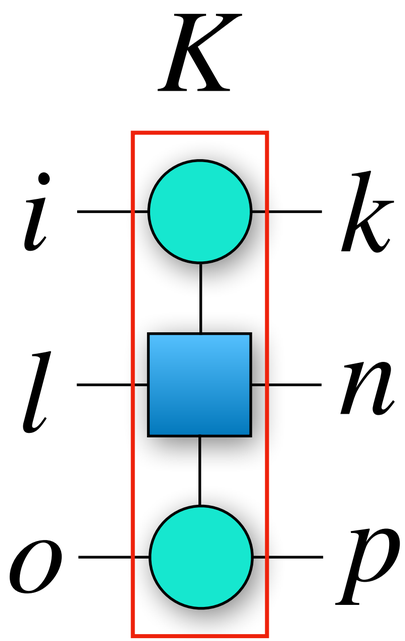
Programming of tensor networks



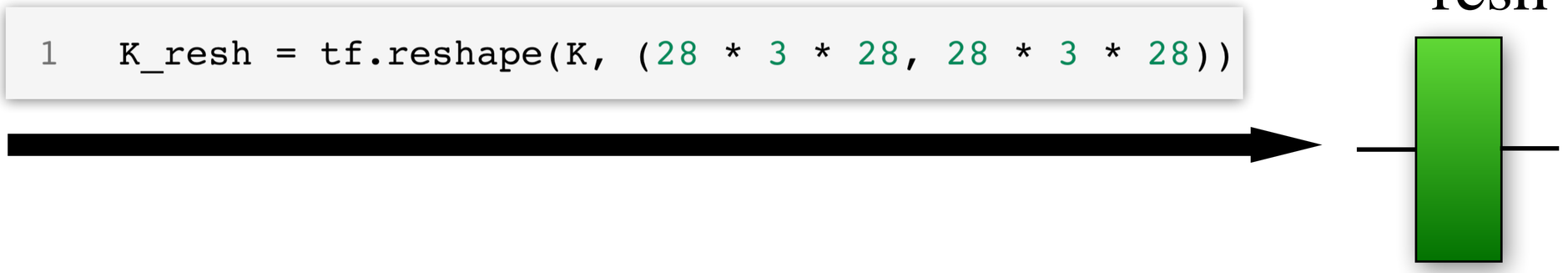

```
K = tf.einsum('ijk,lmnj,opm->iloknp', A, H, A_conj)
```



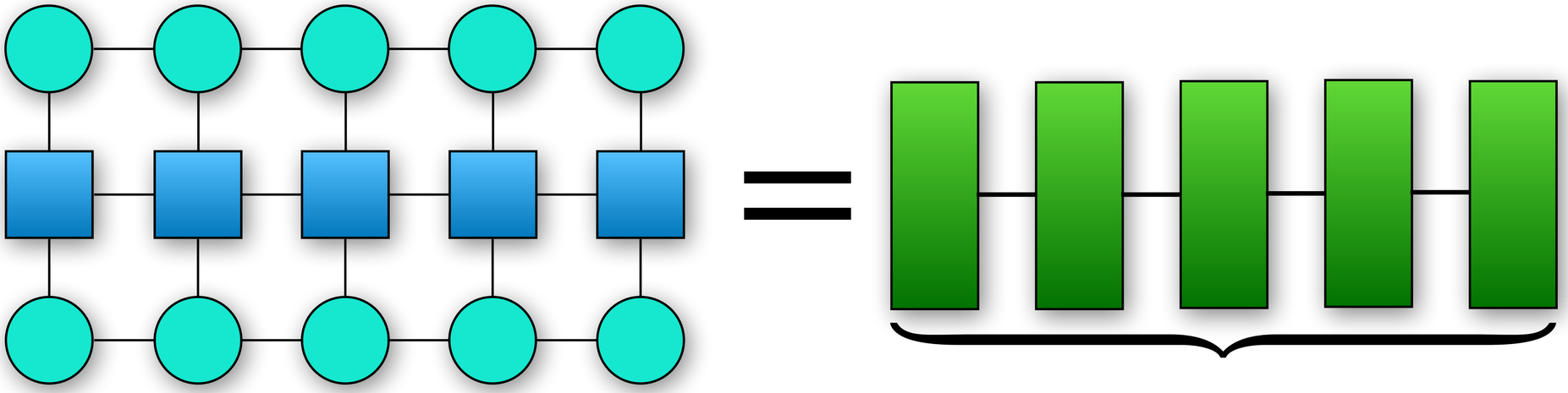
Programming of tensor networks




```
1   K_resh = tf.reshape(K, (28 * 3 * 28, 28 * 3 * 28))
```



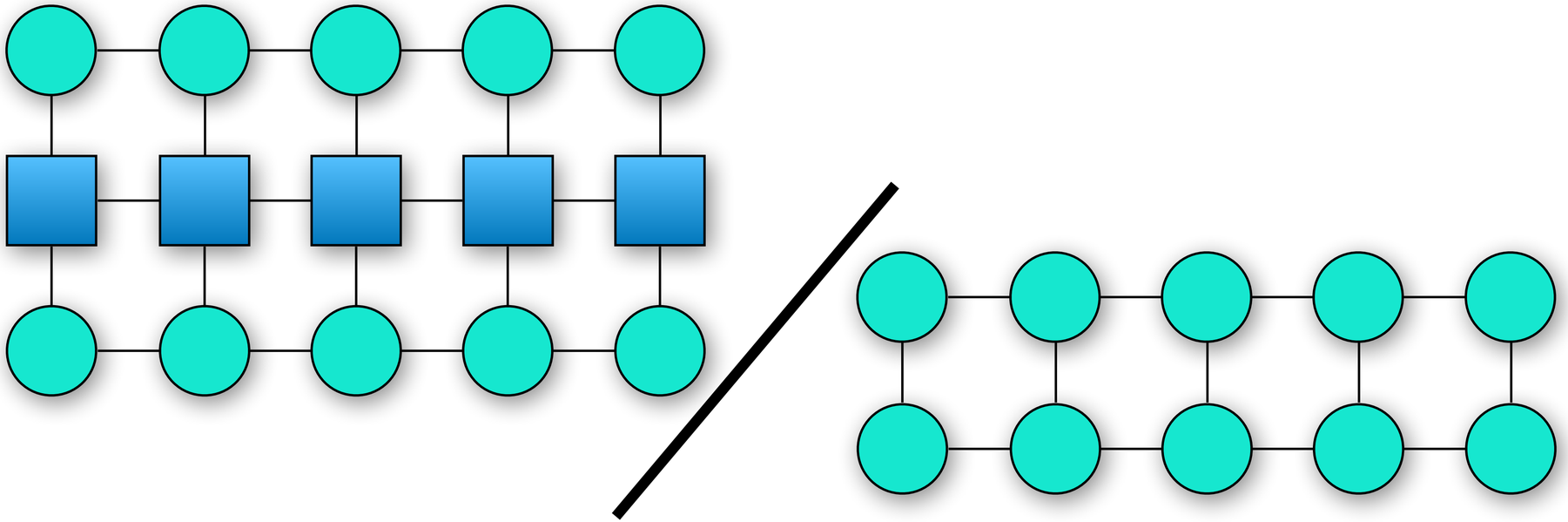
Programming of tensor networks



Many vector by matrix multiplications.
One can calculate by applying
tf.tensordot several times

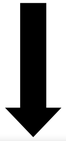
Programming of tensor networks

$$E = \frac{\langle \mathrm{mps} \,|\, H \,|\, \mathrm{mps} \rangle}{\langle \mathrm{mps} \,|\, \mathrm{mps} \rangle} =$$



Ground state of TFI hamiltonian

$$\left\{ \mathrm{Re}\left(A_i\right) \right\}_{i=1}^{N}, \quad \left\{ \mathrm{Im}\left(A_i\right) \right\}_{i=1}^{N}$$



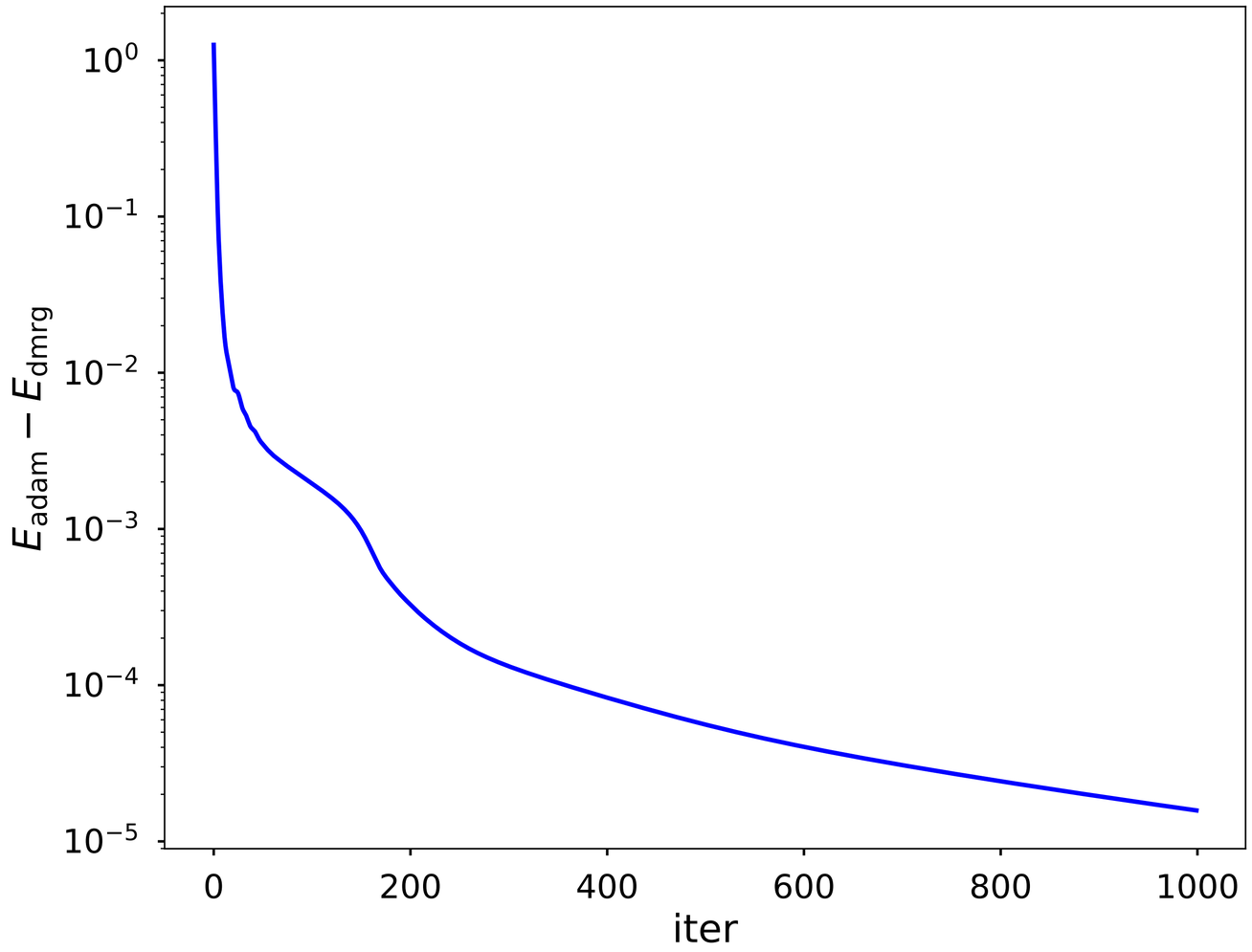
```
1   with tf.GradientTape() as tape:
```

Energy calculation

$$E$$

Ground state of TFI hamiltonian

$$H = \sum_{i=1}^{N-1} \sigma_z^i \sigma_z^{i+1} + \sum_{i=1}^{N} h_x \sigma_x^i + \sum_{i=1}^{N} h_z \sigma_z^i$$



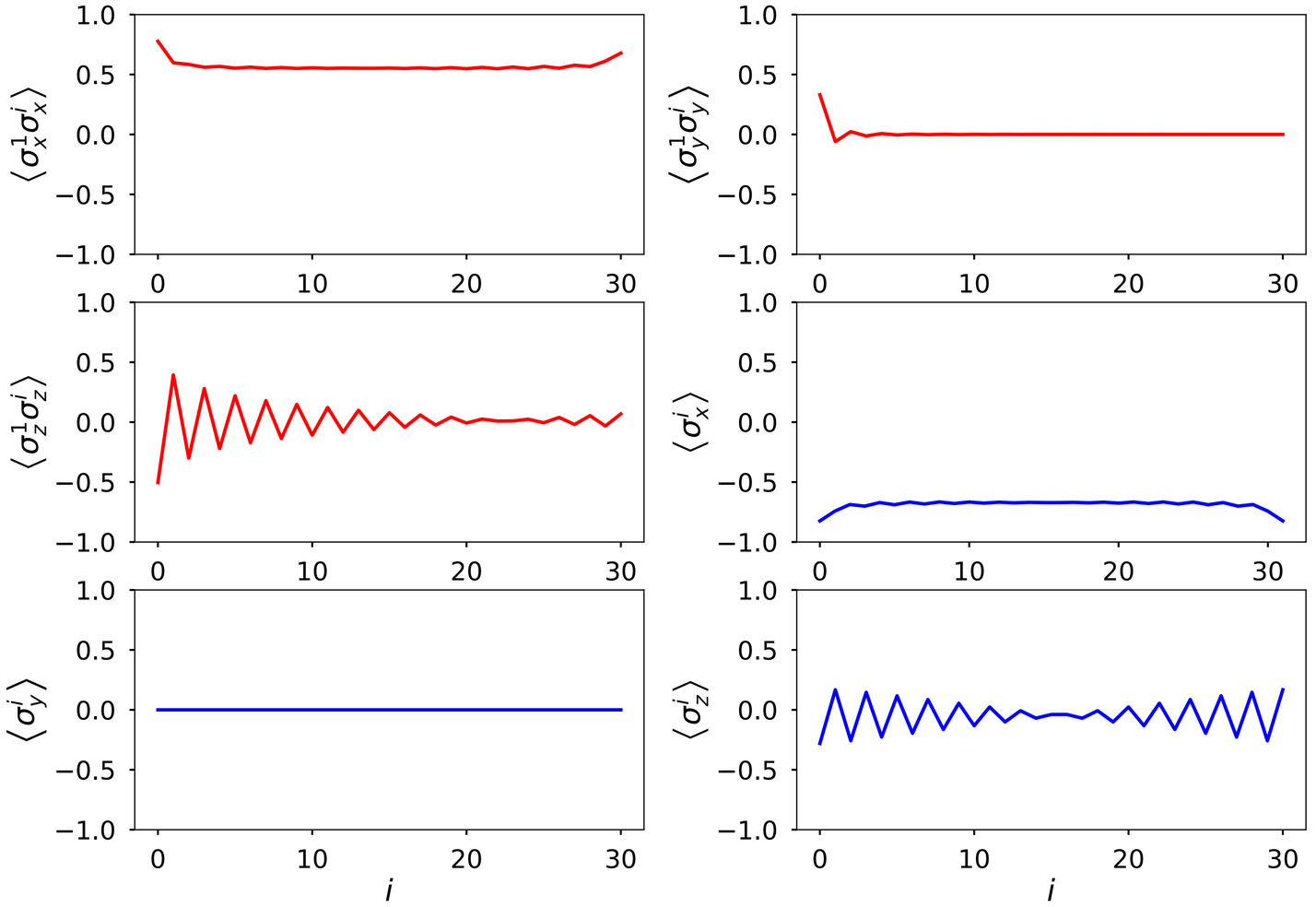
$$N = 32$$

$$h_x = 1$$

$$h_z = 0.3$$

Ground state of TFI hamiltonian

$$H = \sum_{i=1}^{N-1} \sigma_z^i \sigma_z^{i+1} + \sum_{i=1}^{N} h_x \sigma_x^i + \sum_{i=1}^{N} h_z \sigma_z^i$$

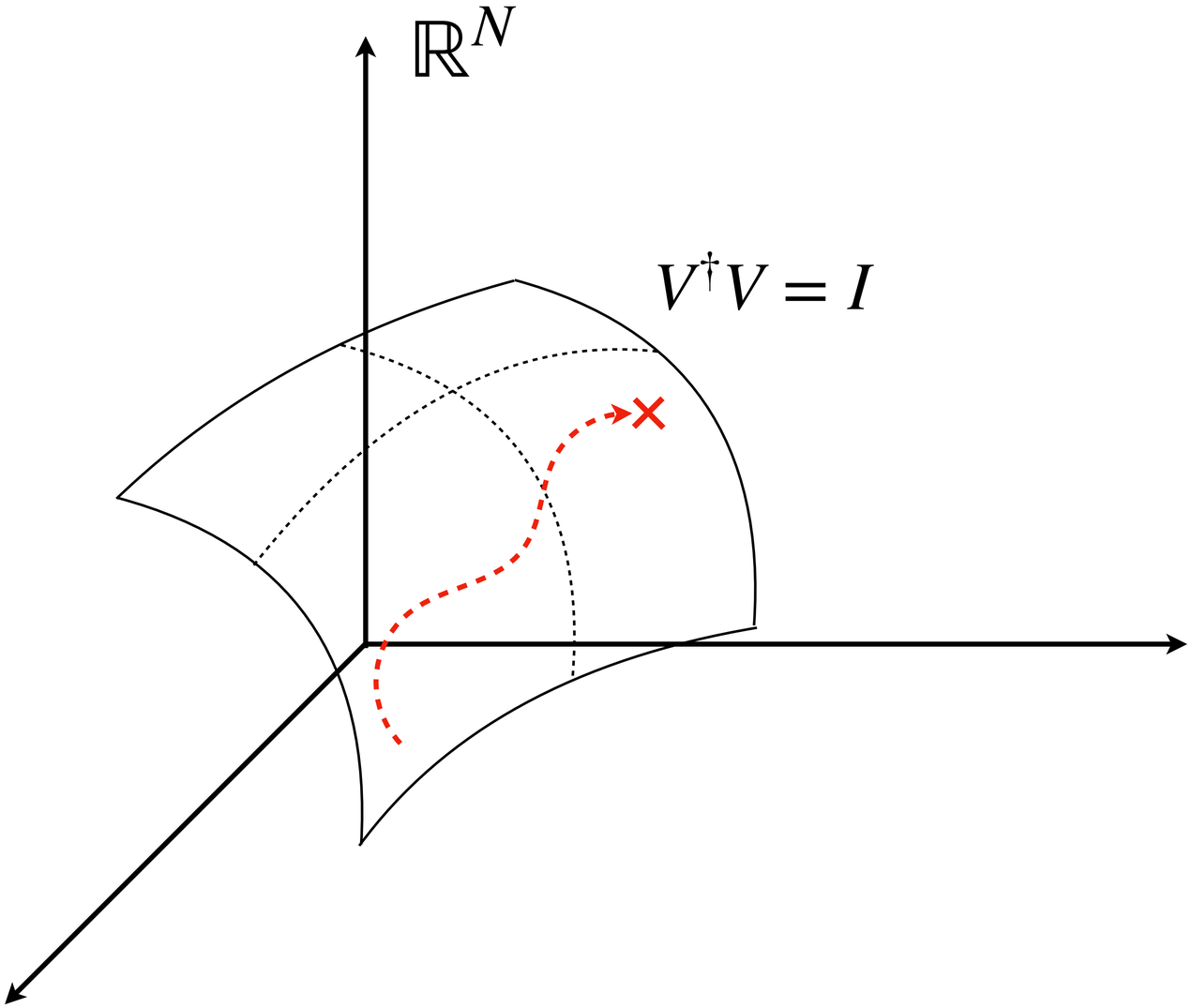


$N = 32$

$h_x = 1$

$h_z = 0.3$

Constrained optimization



QGOpt

https://qgopt.readthedocs.io/en/latest/

https://github.com/LuchnikovI/QGOpt

Luchnikov I, Krechetov M, Filippov S. Riemannian optimization and automatic differentiation for complex quantum architectures. arXiv preprint arXiv:2007.01287. 2020 Jul 2.

Constrained optimization with QGOpt

$$V_{n,\,p} = \left\{ Z \in \mathbb{C}^{n\times m} \,|\, n \geq m,\, Z^\dagger Z = I \right\} \text{ — Stiefel manifold}$$

Edelman, A., Arias, T. A., & Smith, S. T. (1998). The geometry of algorithms with orthogonality constraints. *SIAM journal on Matrix Analysis and Applications*, *20*(2), 303-353.

$$S^n_{++} = \left\{ S \in \mathbb{C}^{n\times n} \,|\, S^\dagger = S, S \succeq 0 \right\} \text{ — psd matrices}$$

$$\rho^n_{++} = \left\{ \rho \in \mathbb{C}^{n\times n} \,|\, \rho^\dagger = \rho, \rho \succeq 0,\, \mathrm{Tr}(\rho) = 1 \right\} \text{ — density matrices}$$

$$C^{n^2}_{++} = \left\{ C \in \mathbb{C}^{n^2\times n^2} \,\middle|\, C^\dagger = C, C \succeq 0,\, \mathrm{Tr}_2(C) = I \right\} \text{ — Choi matrices}$$

$$\mathrm{POVM}^{n,m} = \left\{ \{E_i\}_{i=1}^n \,\middle|\, E_i \in \mathbb{C}^{m\times m},\, E_i^\dagger = E_i,\, E_i \succeq 0 \text{ for } i = 1,\ldots,n,\, \sum_i E_i = I \right\} \text{ — POVMs}$$

$$H_n = \left\{ H \in \mathbb{C}^{n\times m} \,|\, H = H^\dagger \right\} \text{ —Hermitian matrices}$$

All of them are implemented in QGOpt!

See also https://manopt.org

Constrained optimization with QGOpt

```
stiefel = qgo.manifolds.StiefelManifold()
density_matrix = qgo.manifolds.DensityMatrix()
choi_matrix = qgo.manifolds.ChoiMatrix()
hermitian_matrix = qgo.manifolds.HermitianMatrix()
positive_cone = qgo.manifolds.PositiveCone()
povm = qgo.manifolds.POVM()
```

Constrained optimization with QGOpt

```
stiefel = qgo.manifolds.StiefelManifold()
density_matrix = qgo.manifolds.DensityMatrix()
choi_matrix = qgo.manifolds.ChoiMatrix()
hermitian_matrix = qgo.manifolds.HermitianMatrix()
positive_cone = qgo.manifolds.PositiveCone()
povm = qgo.manifolds.POVM()
```

```
opt = qgo.optimizers.Adam(m, lr)
```

Constrained optimization with QGOpt

```
1  stiefel = qgo.manifolds.StiefelManifold()
2  density_matrix = qgo.manifolds.DensityMatrix()
3  choi_matrix = qgo.manifolds.ChoiMatrix()
4  hermitian_matrix = qgo.manifolds.HermitianMatrix()
5  positive_cone = qgo.manifolds.PositiveCone()
6  povm = qgo.manifolds.POVM()
```

```
1  opt = qgo.optimizers.Adam(m, lr)
```

Constrained optimization with QGOpt

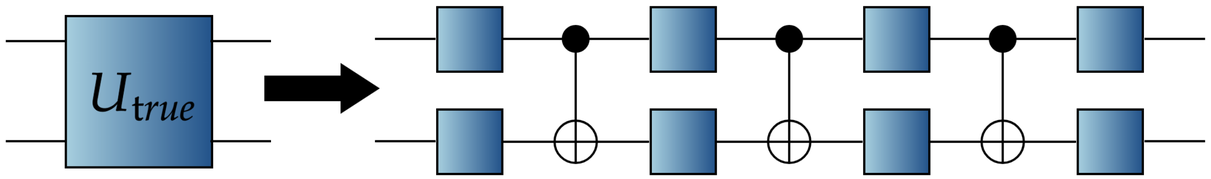
```
stiefel = qgo.manifolds.StiefelManifold()
density_matrix = qgo.manifolds.DensityMatrix()
choi_matrix = qgo.manifolds.ChoiMatrix()
hermitian_matrix = qgo.manifolds.HermitianMatrix()
positive_cone = qgo.manifolds.PositiveCone()
povm = qgo.manifolds.POVM()
```

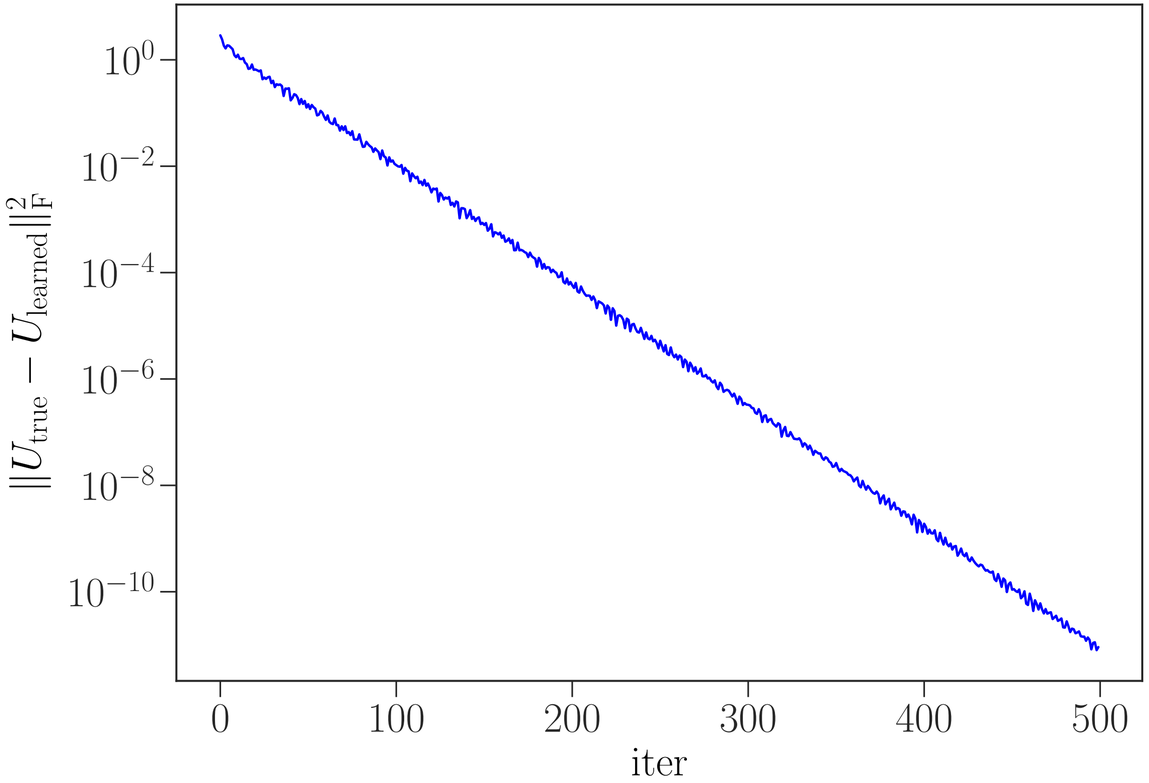
```
opt = qgo.optimizers.Adam(m, lr)
```

```
opt.apply_gradients(zip(loss, grad))
```

Gate decomposition

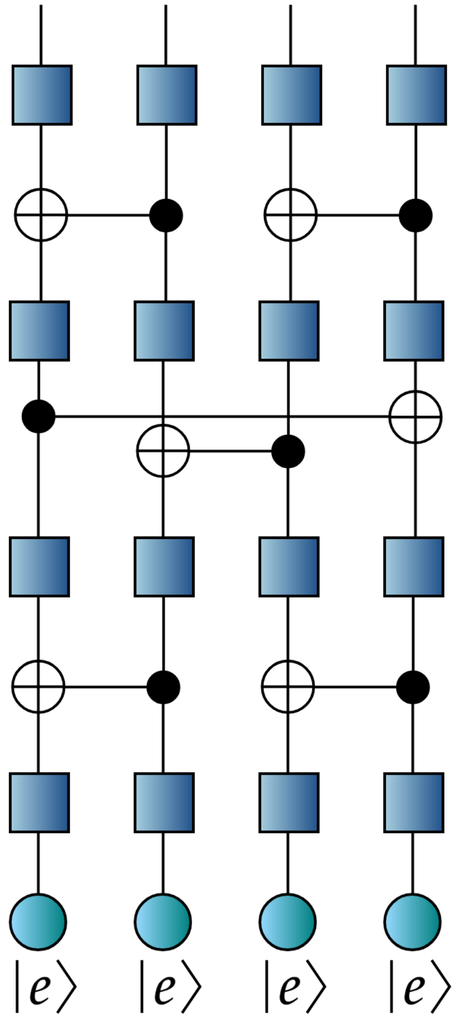
Optimization over Stiefel Manifold

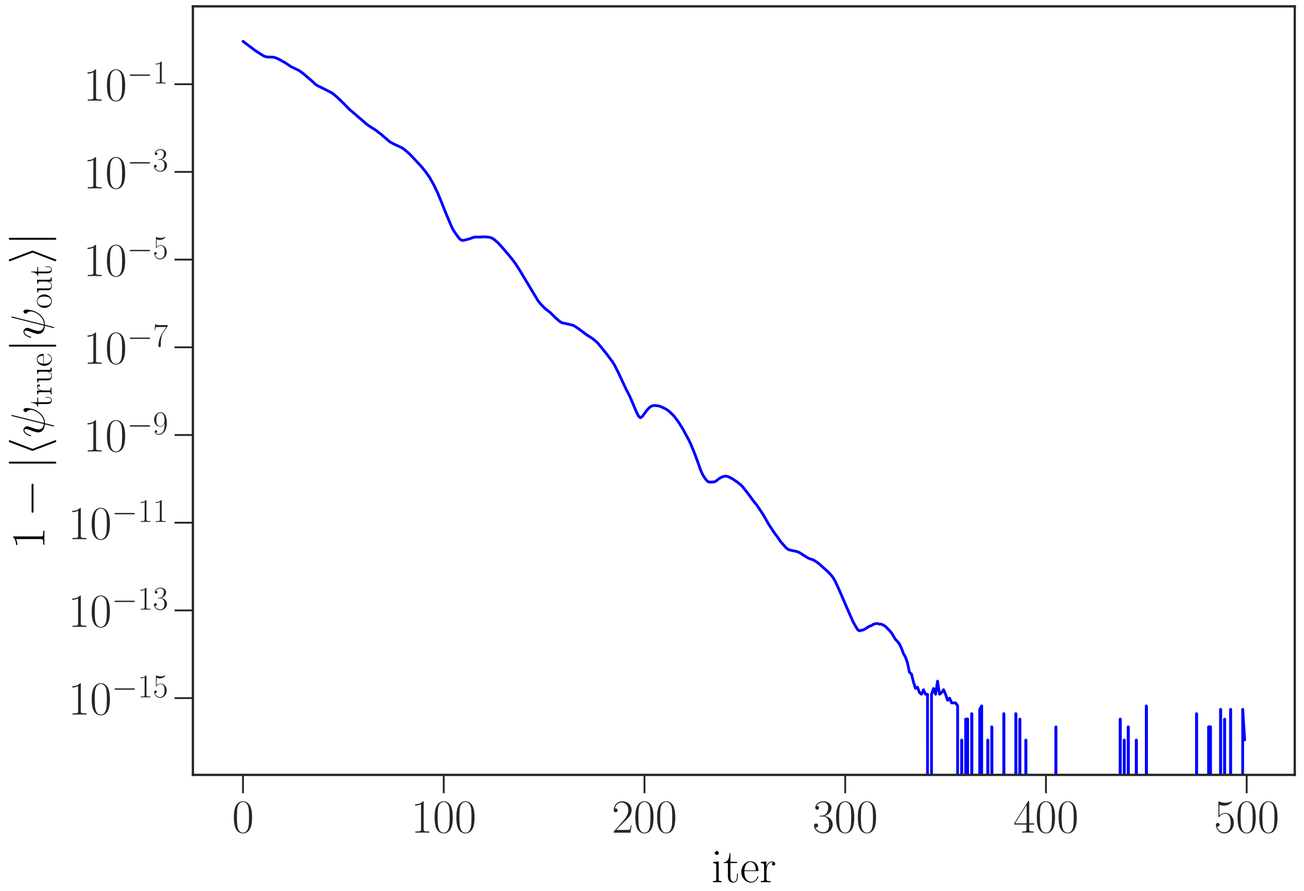




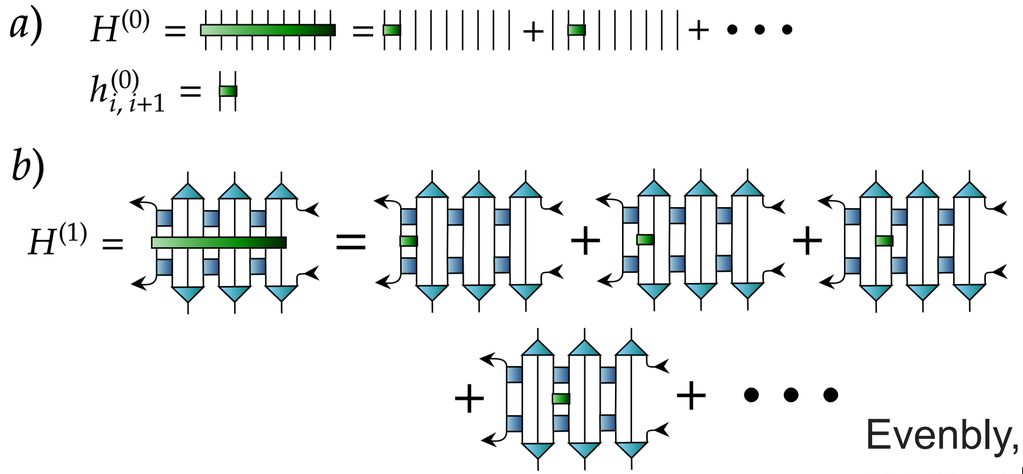
State preparation

Optimization over Stiefel Manifold





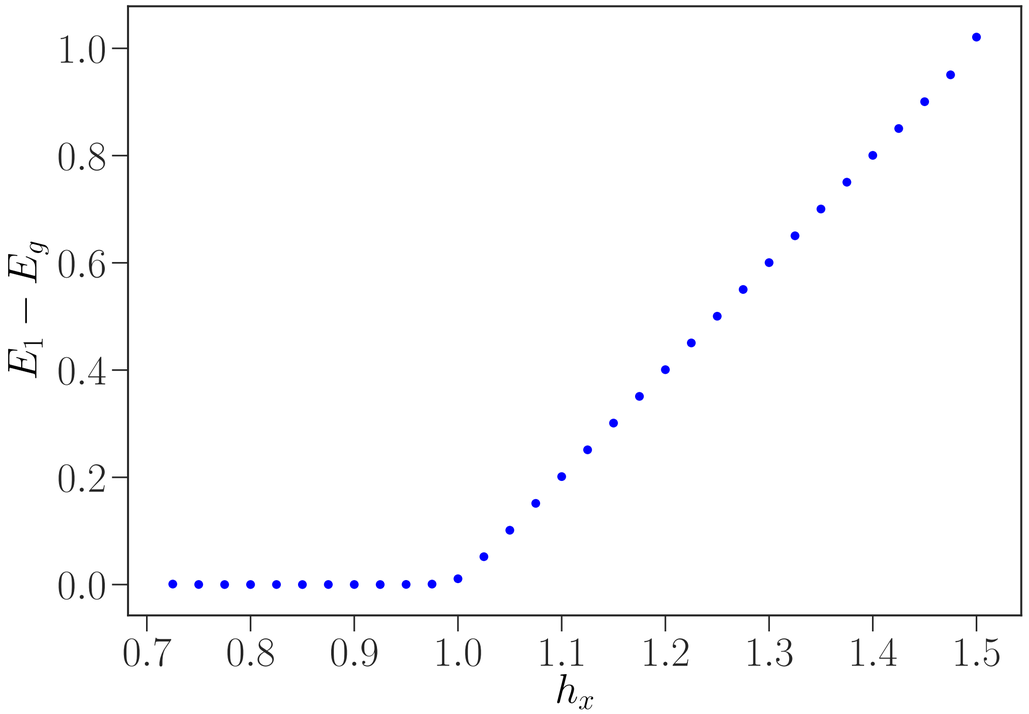
Entanglement renormalization

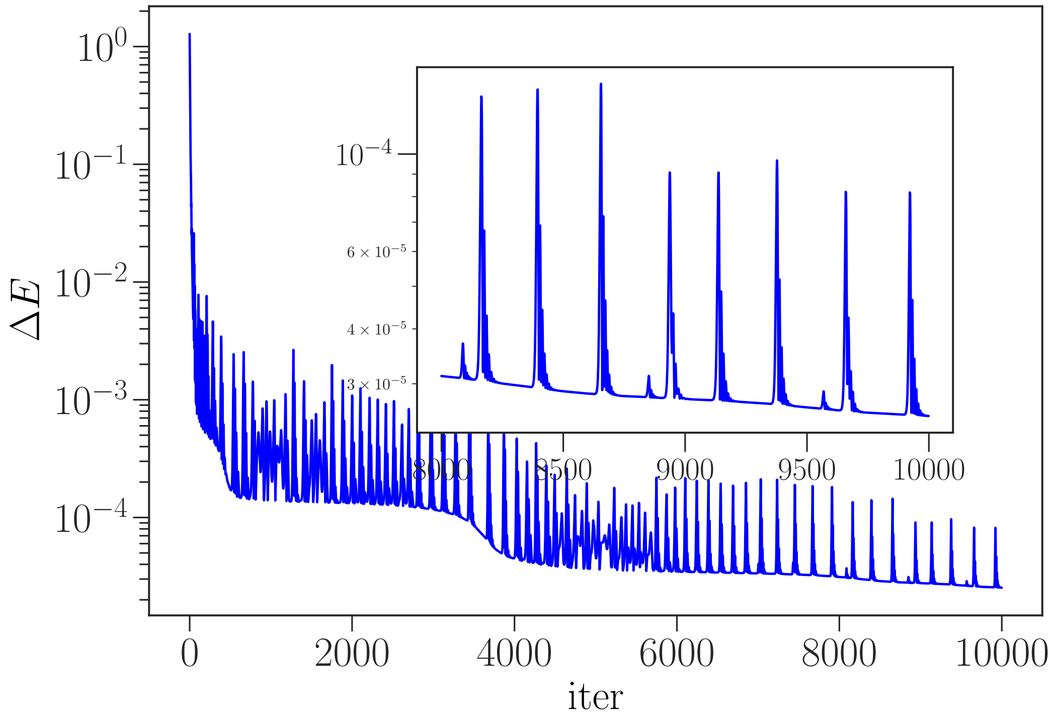


486 quantum spins TFI model

$$H = \sigma_N^z \sigma_1^z + \sum_{i=1}^{N-1} \sigma_i^z \sigma_{i+1}^z + h_x \sum_{i=1}^{N} \sigma_i^x$$

Evenbly, G., & Vidal, G. (2009). Algorithms for entanglement renormalization. *Physical Review B*, *79*(14), 144108.

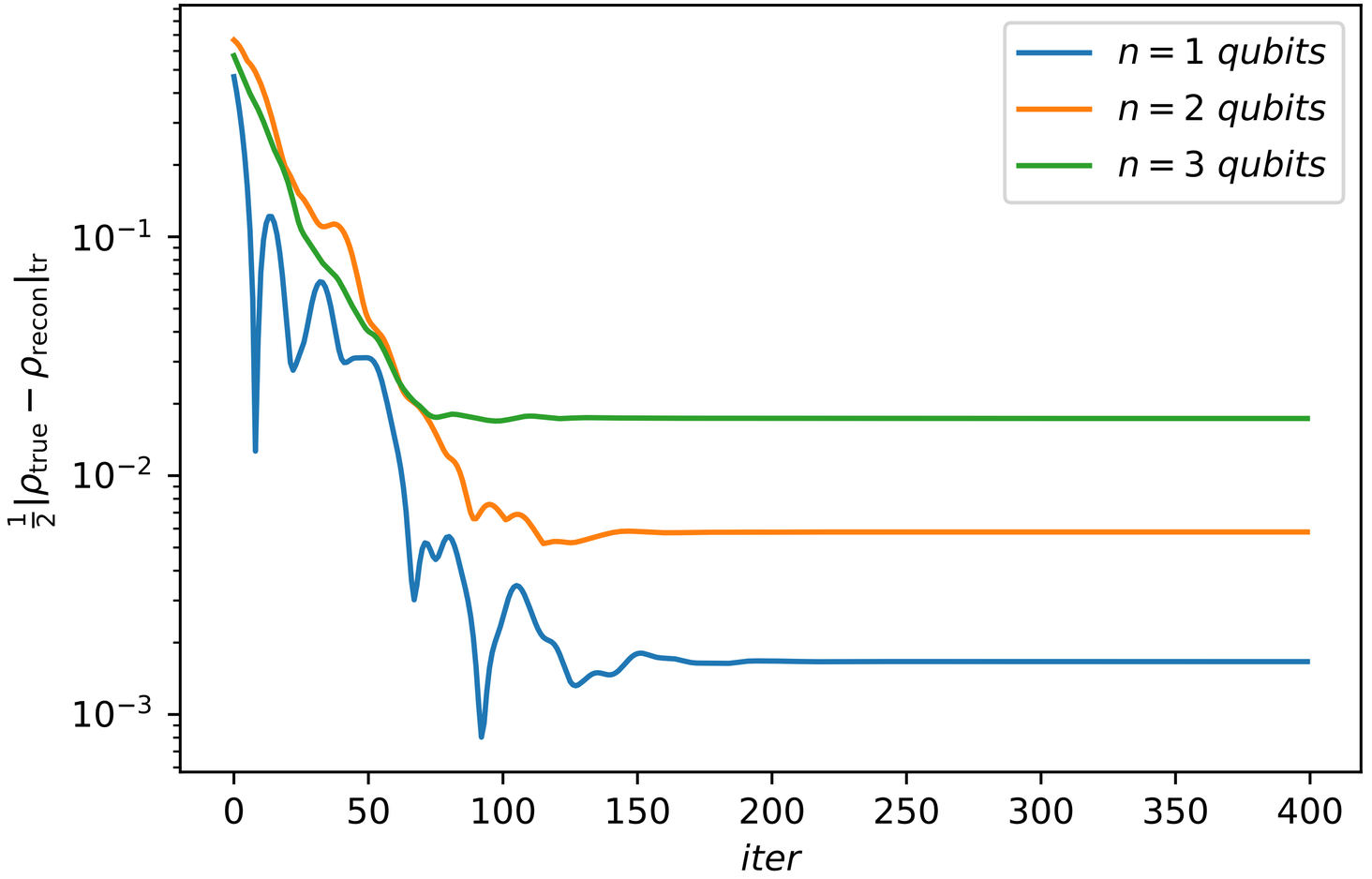




Tomography of quantum states

600 000 measurements, tetrahedral povm

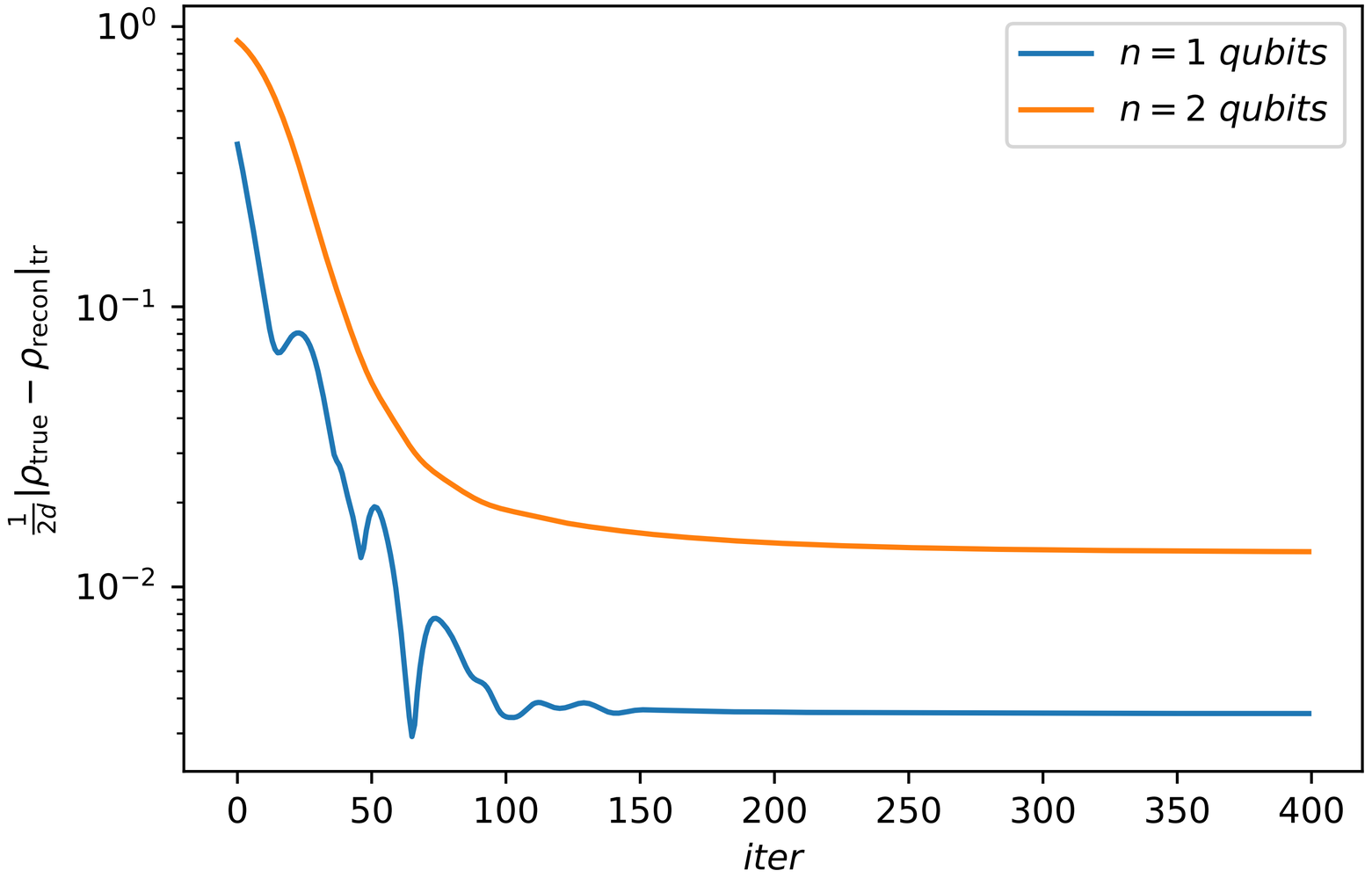
Optimization over manifold of density matrices



Tomography of quantum channels

600 000 measurements, tetrahedral povm and random input state

Optimization over manifold of Choi matrices



QGOpt tutorials

To get familiar with QGOpt API please visit

https://qgopt.readthedocs.io/en/latest/quick_start.html

Summary

- Machine learning with tensor networks
- Quantum control
- Model identification
- Tomography of states and channels
- Many body systems, quantum phase transitions
- Adaptive quantum tomography
- Combinations of approaches (like in deep learning)

Thank you for your attention!