Final Project - Impact of Error Mitigation on the Quantum Teleportation Circuit

Team members and department

- Peter Preisler (PP) - Physics (and Aeronautics)
- Daniel Krojer (DK) - Physics (Theoretical)
- Moritz Blum (MB) - Physics (Experimental)

Author Contributions

Disclaimer: All authors were intellectually involved in the theoretical discussion of all parts of the project. Here, the contribution is distributed according to the main focus and time investment of each author's work.

- Determination of the probability transfer matrix for Bob's qubit measurement (PP)
- Examination of the measurement error mitigation on the outcome of the circuit (PP)
- Introduction of the dynamical decoupling for all qubits (DK)
- Testing of the dynamical decoupling approach (MB, DK)
- Preparation of the presentation slides and main work for the visual presentation (MB)

Project Description

This project will explore the influence of some qiskit error mitigation schemes on the qubit teleportation circuit that was already discussed as a part of the lecture. Fidelity measurements between the initialized qubit state and the final state after the qubit teleportation will be used to quantify the impact of different error mitigation methods on overall success of the teleportation. Especially for the real QPU, the existing model showed very bad behavior due to the noise and time-extensive swap operations. It is expected that error mitigation will significantly improve the outcome of this circuit.

In detail, two different error mitigation schemes will be used:

- Mitigate measurement at the end of the circuit, by using the inverse of the A matrix of the respective qubit.

- Dynamic Decoupling for all qubits that sit around while the swap operation happen.

Imports

In [1]:

```
# Numercial operations
import numpy as np
from numpy import pi

# Visualization
from matplotlib import pyplot as plt
from qiskit.visualization.bloch import Bloch
import matplotlib.patches as mpatches
```

In [2]:

```
# Import quiskit libraries to define and run the quantum circuits
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister
from qiskit.circuit import Parameter # To enable parameterized circuits
from qiskit.quantum_info import SparsePauliOp # Used for example to set up observables for Estimator
from qiskit.transpiler.preset_passmanagers import generate_preset_pass_manager

# For sampling and runs on real backend
from qiskit_ibm_runtime import SamplerV2 as Sampler
from qiskit_ibm_runtime import QiskitRuntimeService
from qiskit_ibm_runtime import Session

# For dynamical decoupling
from qiskit.circuit.library import XGate, YGate, ZGate
from qiskit.circuit.library import RZGate
from qiskit.circuit import SwitchCaseOp

from qiskit.transpiler import PassManager, InstructionDurations
from qiskit.transpiler.passes import ALAPSchedule, DynamicalDecoupling
from qiskit.transpiler.passes.scheduling import (
    ALAPScheduleAnalysis,
    PadDynamicalDecoupling,
)
from qiskit.transpiler import InstructionProperties
from qiskit.circuit.equivalence_library import (
    SessionEquivalenceLibrary as sel,
)
```

```python
from qiskit.transpiler.passes import BasisTranslator
from qiskit.visualization import timeline_drawer
```

In [3]:
```python
# Set up a fake_provider backend to run locally
from qiskit_ibm_runtime.fake_provider import FakeBrisbane, FakeQuebec
#backend = FakeBrisbane()
backend = FakeQuebec()
```

Definition of Functions

Calculation of the probability transfer matrix

In [4]:
```python
def determine_prob_transfer_mat(backend, num_swaps, num_shots):
    '''
    This function will determine the probability transfer matrix for the qubit to which the B state will be swaped
    to. It returns a 2x2 numpy array.
    '''
    with Session(backend = backend) as session:

        # Set up the sampler for the measurement of P(0) and P(1)
        sampler = Sampler(mode = session)

        # Run measurement two times. Once for |0>, to retrieve p and once for |1> to get q
        for i in range(2):

            # Determine the total number of qubits involved in the teleportation circuit (A, A', B + num_swaps)
            num_qubits = 3 + num_swaps

            # Create the quantum circuit
            qc = QuantumCircuit(num_qubits, 1)

            # Prepare the |1> state in the second run
            if i:
                qc.x(num_qubits - 1)

            # Measure the qubit to which B will be swaped to
            qc.measure(num_qubits - 1, 0)

            # Create the "pass manager" that will perform the transpilation
```

```
            pm = generate_preset_pass_manager(backend = backend, optimization_level = 1)
            pm.scheduling = None

            # Run the pass manager on cicuit to transpile it to the backend
            transpiled_circuit = pm.run(qc)

            # Run the sampler N times
            job = sampler.run([transpiled_circuit], shots = num_shots)

            # This is the result from a single pub, and holds results for all measurement outcomes
            pub_results = job.result()[0]

            # Retrieves an dictonary that holds the number of samples that led to outcome 0 or 1
            counts = pub_results.data.c.get_counts()

            # Calculate the error probabilities p and q
            if not(i):
                p = counts.get('1') / num_shots
            else:
                q = counts.get('0') / num_shots

    return np.array([[1 - p, q], [p, 1 - q]])
```

Insert dynamic decoupling sequence during transpilation. All gates, which are added after this have to be translated to the backend basis gates without standard traspilation. This seems to be the only way of making it run on real QPUs.

In [5]:
```
def dynamic_decoupling(qc, backend, dd_sequence, meas_qubit):
    '''
    This function transpiles the given circuit and inserts the dynamic decoupling sequence.
    The teleportation protocol and measurements are added and translated to the backend basis gates without standard transpila
    '''

    # Get basis gates from backend
    target = backend.target
    basis_gates = list(target.operation_names)


    # Create the "pass manager" that will perform the transpilation
    pm = generate_preset_pass_manager(target=target, optimization_level=1)
```

```python
    # Run the pass manager on cicuit to finish the transpilation
    qc_transp = pm.run(qc)

    # Create specific pass manager to insert dd_sequence
    pm_dd = PassManager(
    [
        ALAPScheduleAnalysis(target=target),
        PadDynamicalDecoupling(target=target, dd_sequence=dd_sequence),
    ]
    )

    # Insert the dynamic decoupling sequence
    qc = pm_dd.run(qc_transp)

    # Perform the teleportation protocol
    qc = teleportation_protocol(qc, meas_qubit)

    # Add the measurement of the B qubit (teleported away)
    qc = create_measurement(qc, meas_qubit)

    # Change all non-basis gates to basis gates without the standard transpiler (only works for real QPU backends)
    qc = BasisTranslator(sel, basis_gates)(qc)

    return qc
```

In [6]:
```python
def dd_sequence_XX(n):
    # X-X dynamic decoupling sequence
    return [XGate()] * 2 * n

def dd_sequence_XY4(n):
    # "XY4" dynamic decoupling sequence with Z instead of X gates, because RZ is a basis gate
    return [XGate(), RZGate(pi)] * 2 * n
```

Prepare the qubit states for teleportation

In [7]:
```python
def prep_entangled_state(qc: QuantumCircuit):
    '''
    Prepares the entangled qubit state that Alice and Bob share as well as the state that Alice wants to teleport.
    Returns the prepared quantum circuit.
    '''
```

```python
    # Apply a rotation about the x axis to the A' qubit (q0) by - pi/3 to initialize the state
    qc.rx(- pi / 3, 0)

    # Apply an X gate to the A qubit (q1)
    qc.x(1)

    # Apply a Hadarmard gate to the A qubit (q1)
    qc.h(1)

    # Apply an open CNOT gate from A acting on B (q1 -> q2)
    qc.cx(1, 2, ctrl_state = 0)

    # Return the prepared qubit state of A, A' and B
    return qc
```

Perform the teleportation protocol to recover state A'

In [8]:
```python
def teleportation_protocol(qc: QuantumCircuit, meas_qubit: int):
    '''
    This function performs the quantum teleportation protocol as defined in PS2 Figure 5.1 that teleports Alice's
    quantum state on qubit A' to Bobs qubit, which might sit a certain distance away
    '''
    # Apply a CNOT gate from A' to A (aka. q0 to q1)
    qc.cx(0, 1)

    # Apply a Hadamard gate on A' (q0)
    qc.h(0)

    # Measure qubit A to classical bit 0
    qc.measure(0, 0)

    # Measure qubit A' to classical bit 1
    qc.measure(1, 1)

    # Perform the following operations depending on the classical measurements
    with qc.switch(qc.cregs[0]) as case:
        with case(0b00):
            qc.y(meas_qubit)
        with case(0b01):
```

```python
            qc.x(meas_qubit)
        with case(0b10):
            qc.z(meas_qubit)
        with case(0b11):
            qc.id(meas_qubit)

    return qc
```

Add parameter dependent rotations for expectation value measurement

In [9]:
```python
def create_measurement(qc: QuantumCircuit, meas_qubit: int):
    '''
    Performs two parameter-dependent rotations before measuring the swaped qubit, so that the expectation values in the
    X, Y, and Z direction can be determined. This will help to visualize the state on the Bloch sphere.
    '''
    # Define parameters
    phi = Parameter('$\\phi$')
    theta = Parameter('$\\theta$')

    # Add rotation about x and y axis to circuit
    qc.rx(phi, meas_qubit)
    qc.ry(theta, meas_qubit)



    qc.measure(meas_qubit, 2)

    return qc
```

Create the full teleportation circuit for a given number of swaps (optional dynamical decoupling)

In [10]:
```python
def create_for_given_distance(N: int, backend, transpile = False, dd_sequence = False, dd = False):
    '''
    Defines a quantum circuit, prepares and initial state, moves a qubit away from the rest of the circuit and
    performs the teleportation protocol to teleport the information from qubit A' to B. Optionally, a dynamical
    decoupling sequence can be added to the circuit.
    '''
    # Define the quantum and classical register
    A_prime = QuantumRegister(1, 'A\'')
```

```python
    A = QuantumRegister(1, 'A')
    B = QuantumRegister(1, 'B')
    swap_qubits = QuantumRegister(N, 'swaper')
    classical_bits = ClassicalRegister(3, 'classical')

    # Initialize the quantum circuit
    qc = QuantumCircuit(A_prime, A, B, swap_qubits, classical_bits)

    meas_qubit = 3 + N - 1

    # Prepare the etangled qubit state
    qc = prep_entangled_state(qc)

    # Swap Bobs qubit with a qubit that is N qubits away. The barriers around the swap operation are needed to
    # separate this operation from the preparation and teleportation protocol.

    if N:
        for i in range(N):
            qc.barrier()
            qc.swap(2 + i, 3 + i)
        qc.barrier()

    if dd:
        # Add the dynamical decoupling to the circuit which includes the teleportation protocol and measurement
        qc = dynamic_decoupling(qc, backend, dd_sequence, meas_qubit)
    else:
        # Perform the teleportation protocol
        qc = teleportation_protocol(qc, meas_qubit)

        # Add the measurement of the B qubit (teleported away)
        qc = create_measurement(qc, meas_qubit)

    if transpile:
        # Create the "pass manager" that will perform the transpilation and run it
        pm = generate_preset_pass_manager(backend = backend, optimization_level = 1)
        pm.scheduling = None
        qc = pm.run(qc)

    return qc
```

Calculate the X, Y, and Z expectation values and fidelity (optional error mitigation)

In [11]:
```
def calc_expt_vals_and_fidelity(pub_results, N_shots, matrix_A_inv = np.identity(2)):
    '''
    Calculation of the expectation values for X, Y and Z as well as the fidelity. Has the optionallity to correct
    the measurement with a probability transfer matrix A (function takes its inverse).
    '''
    expt_vals = []

    # Loop over the three parameter sets for X, Y and Z and sum the counts for expectation values
    for i in range(3):
        counts = pub_results.data.classical[i].get_counts()
        num_zero = 0
        num_one = 0
        for bits in ['000', '010', '001', '011']:
            count = counts.get(bits)
            if count:
                num_zero += count
        for bits in ['100', '110', '101', '111']:
            count = counts.get(bits)
            if count:
                num_one += count

        # Create column vector for the probabilities (P(0), P(1)) and calculate the initial state with inverse matrix A
        meas_prob = np.array([num_zero, num_one]).T / N_shots
        true_prob = matrix_A_inv @ meas_prob

        expt_vals.append(true_prob[0] - true_prob[1])

    # Save the expectation values for simpler calculation
    X = expt_vals[0]
    Y = expt_vals[1]
    Z = expt_vals[2]

    # Calculate the angles phi and theta from the expectation values
    phi_value = np.arctan2(Y, X)
    theta_value = np.arccos(Z / np.sqrt(X ** 2 + Y ** 2 + Z ** 2))

    # Calculate alpha and beta of the end state
    alpha_B = np.cos(theta_value / 2)
```

```
    beta_B = np.exp(1j * phi_value) * np.sin(theta_value / 2)

    # Alpha and Beta for the initial state defined for the computation
    alpha_A_prime = np.cos(np.pi / 6)
    beta_A_prime = 1j*np.sin(np.pi / 6)

    # Calculate the fidelity of the start and end state
    fidelity = (abs(alpha_A_prime.conjugate() * alpha_B + beta_A_prime.conjugate() * beta_B)) ** 2

    return expt_vals, fidelity
```

Visualize states on bloch sphere

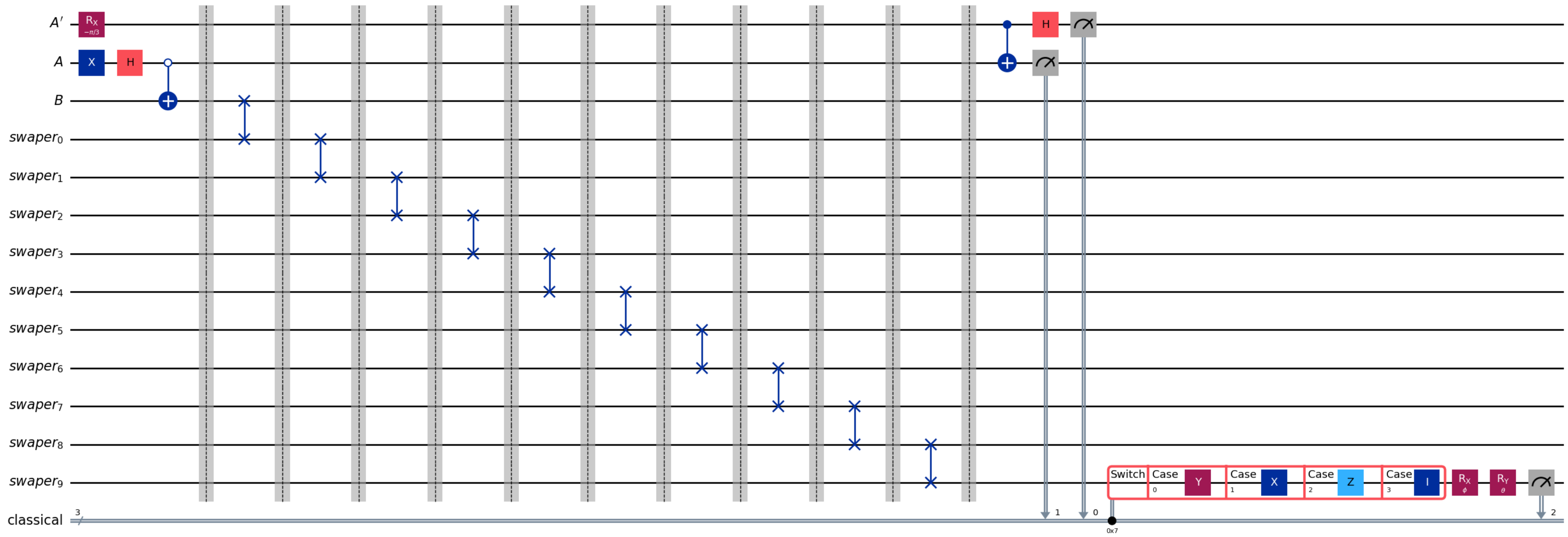
In [12]:
```
def plot_on_bloch_sphere(bloch_vectors: list):

    # Create Bloch sphere and add vectors
    bloch = Bloch()

    # Set the figure size (width, height)
    bloch.figsize = (4, 4)

    # Set vector colors
    bloch.vector_color = [color for *_, color in bloch_vectors]

    for vec, *_ in bloch_vectors:
        bloch.add_vectors(vec)

    # Render the plot
    bloch.render()

    # Add a custom legend
    legend_patches = [mpatches.Patch(color = color, label=label) for _, label, color in bloch_vectors]
    plt.legend(handles = legend_patches, loc = 'upper left')
    plt.show()
```

Initial Function Testing

Plot the untranspiled circuit

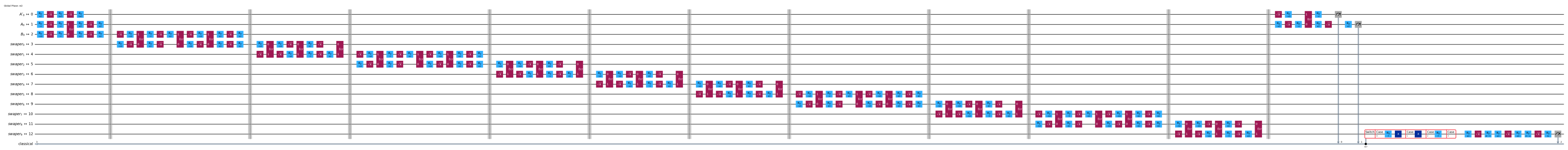
In [13]:
```python
# Create circuit with desired amount of swaps
num_swaps = 10
qc = create_for_given_distance(num_swaps, backend)

# Draw the circuit
qc.draw('mpl', fold = False)
```

Out[13]:



Show the transpiled circuit

In [14]:
```python
# Create the "pass manager" that will perform the transpilation
pm = generate_preset_pass_manager(backend = backend, optimization_level = 1)
pm.scheduling = None

# Run the pass manager on cicuit to finish the transpilation
transpiled_circuit = pm.run(qc)

# Draw the transpiled circuit
transpiled_circuit.draw('mpl', idle_wires = False, fold = False)
```

Out[14]: 

In [16]:
```
# Display circuit depth
transpiled_circuit.depth()
```

Out[16]: 132

Run the full teleportation on a fake backend

In [13]:
```
# Set up the Sampler that will run the circuit
sampler = Sampler(backend)

# Define the parameters to measrue the expectation values
param_vals= np.vstack([
    np.array([0, -pi / 2]),     # Measurement along X
    np.array([pi / 2, 0]),      # Measurement along Y
    np.array([0, 0])            # Measurement along Z
])

# Define the number of shots for each parameter combination for the sampler
N_shots = 1024
```

In [18]:
```
# Run the sampler for all values of the two parameters
job = sampler.run([(transpiled_circuit, param_vals, N_shots)])

# Store results
job_result = job.result()

# This is the result from a single pub, and holds results for all observables estimated (if using Estimator)
pub_results = job.result()[0]
```

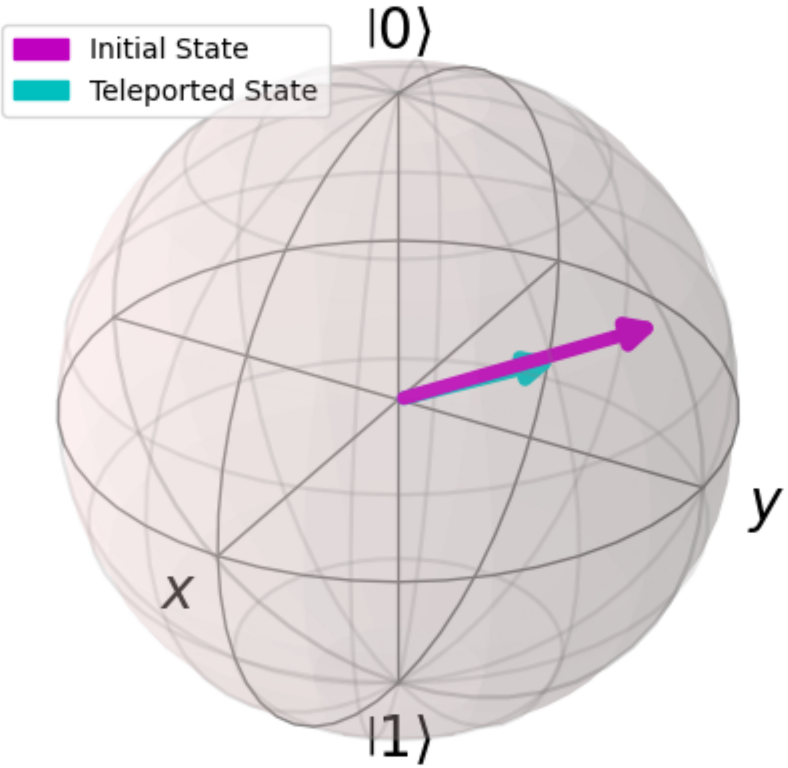
In [191…
```
# Calculate expectation values and fidelity
expt_vals, fidelity = calc_expt_vals_and_fidelity(pub_results, N_shots)

print(fidelity)
```

```
0.9997605514350726
```

Figure 1: Bloch sphere plot of the initial and teleported state for the evaluation on the FakeBrisbane backend.

In [192...

```
# Define the initial and the teleported bloch vector and plot them on the sphere
bloch_vectors = [
    ([0, np.sin(pi/3), np.cos(pi/3)], 'Initial State', 'm'),
    (expt_vals, 'Teleported State', 'c')
]

plot_on_bloch_sphere(bloch_vectors)
```



Examining the Influence of Readout Error Mitigation

In [ ]:

```
# Defines the QPU service to which the run order will be routed. Using the IBM Quantum Platform with 10 free computation
# minutes to save on course credits.

service = QiskitRuntimeService(
    channel='ibm_quantum',
    instance='ibm-q/open/main',
    token=''
)

# Set the physical backend
#backend = service.backend('ibm_brisbane')

print(backend)
```

```
<IBMBackend('ibm_brisbane')>
```

In [30]:

```
num_swaps = 10
num_shots = 2048

# Set up the Sampler that will run the circuit
sampler = Sampler(backend)

# Define the parameters to measure the expectation values
param_vals= np.vstack([
    np.array([0, -pi / 2]),      # Measurement along X
    np.array([pi / 2, 0]),       # Measurement along Y
    np.array([0, 0])             # Measurement along Z
])
```

In [ ]:

```
# Calculate the probability transfer matrix just before the teleportation protocol is performed (100k shots)
#A = determine_prob_transfer_mat(backend, num_swaps, int(1e5))

# Create the transpiled circuit
transpiled_circuit = create_for_given_distance(num_swaps, backend, transpile = True)

# Run the sampler for all values of the two parameters
job = sampler.run([(transpiled_circuit, param_vals, num_shots)])

# Saves the results from the PUB (for all three parameter combinations)
pub_results = job.result()[0]
```

In [ ]:
```
# Save the matrix that was just calculated
prob_trans_mat = np.array([ [0.99243, 0.02752],
                            [0.00757, 0.97248]])
```

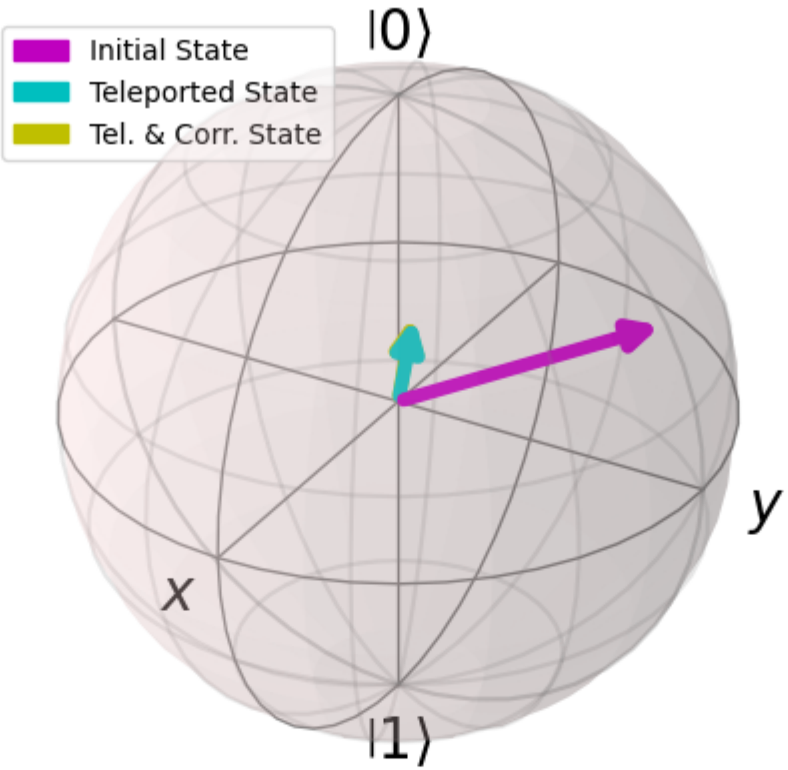
In [ ]:
```
# Calculate expectation values and fidelity without correction
expt_vals, fidelity = calc_expt_vals_and_fidelity(pub_results, num_shots)

# Calculate expectation values and fidelity with correction
expt_vals_corr, fidelity_corr = calc_expt_vals_and_fidelity(pub_results, num_shots, np.linalg.inv(prob_trans_mat))
```

In [209…
```
print('\t\tUncorrected\t\tCorrected\t\tDifference')
print(f'Fidelity\t{fidelity}\t{fidelity_corr}\t{np.abs(fidelity-fidelity_corr)}')
```

```
                Uncorrected             Corrected               Difference
Fidelity        0.8312134632495023      0.8122747170396597      0.018938746209842594
```

Figure 2: Bloch sphere plot for the initial and teleported state ran on the real QPU. Additionally, a readout error corrected version of the teleported state is depicted. It only deviates insignificantly from the uncorrected state.

In [210…
```
# Define the initial and the teleported bloch vector and plot them on the sphere
bloch_vectors = [
    ([0, np.sin(pi/3), np.cos(pi/3)], 'Initial State', 'm'),
    (expt_vals, 'Teleported State', 'c'),
    (expt_vals_corr, 'Tel. & Corr. State', 'y')
]

plot_on_bloch_sphere(bloch_vectors)
```



This result shows that the low fidelity of the final state is due to a large dephasing of the state rather than a measurement error. The corrected, and uncorrected state are almost equivalent to each other. If the measurement errors were significant, one would expect the fidelity to improve for the corrected state. However, since the percentages are so low, the uncorrected state is only unsignificantly shifted from its true position. It is pure chance that the fidelity of the uncorrected state is lower.

In [214…

```
# Change the number of swaps to 9 since the 11th qubit has a large readout error
num_swaps = 9
```

In [ ]:

```
# Calculate the probability transfer matrix just before the teleportation protocol is performed (100k shots)
# A = determine_prob_transfer_mat(backend, num_swaps, int(1e5))

# Create the transpiled circuit
transpiled_circuit = create_for_given_distance(num_swaps, backend, transpile = True)
```

```python
# Run the sampler for all values of the two parameters
job = sampler.run([(transpiled_circuit, param_vals, num_shots)])

# Saves the results from the PUB (for all three parameter combinations)
pub_results = job.result()[0]
```

In [221…

```python
# Save the matrix that was just calculated
prob_trans_mat = np.array([ [0.97343, 0.0384 ],
                            [0.02657, 0.9616 ]])
```

In [217…

```python
# Calculate expectation values and fidelity without correction
expt_vals, fidelity = calc_expt_vals_and_fidelity(pub_results, num_shots)

# Calculate expectation values and fidelity with correction
expt_vals_corr, fidelity_corr = calc_expt_vals_and_fidelity(pub_results, num_shots, np.linalg.inv(prob_trans_mat))
```

In [218…

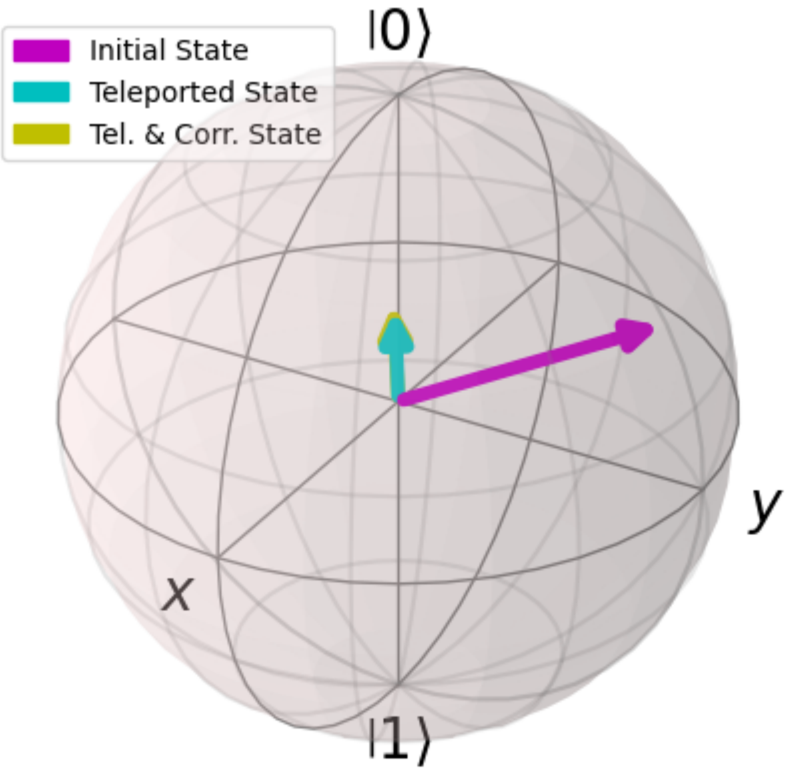
```python
print('\t\tUncorrected\t\tCorrected\t\tDifference')
print(f'Fidelity\t{fidelity}\t{fidelity_corr}\t{np.abs(fidelity-fidelity_corr)}')
```

```
                Uncorrected             Corrected               Difference
Fidelity        0.7552119907519368      0.7405224818470345      0.014689508904902282
```

Figure 3: Same plot as in Figure 2 but for 9 swaps and a newly calculated A matrix. This number of swaps was chosen deliberately to test a different qubit and especially one that had many readout assignment errors (see explanation below).

In [219…

```python
# Define the initial and the teleported bloch vector and plot them on the sphere
bloch_vectors = [
    ([0, np.sin(pi/3), np.cos(pi/3)], 'Initial State', 'm'),
    (expt_vals, 'Teleported State', 'c'),
    (expt_vals_corr, 'Tel. & Corr. State', 'y')
]

plot_on_bloch_sphere(bloch_vectors)
```

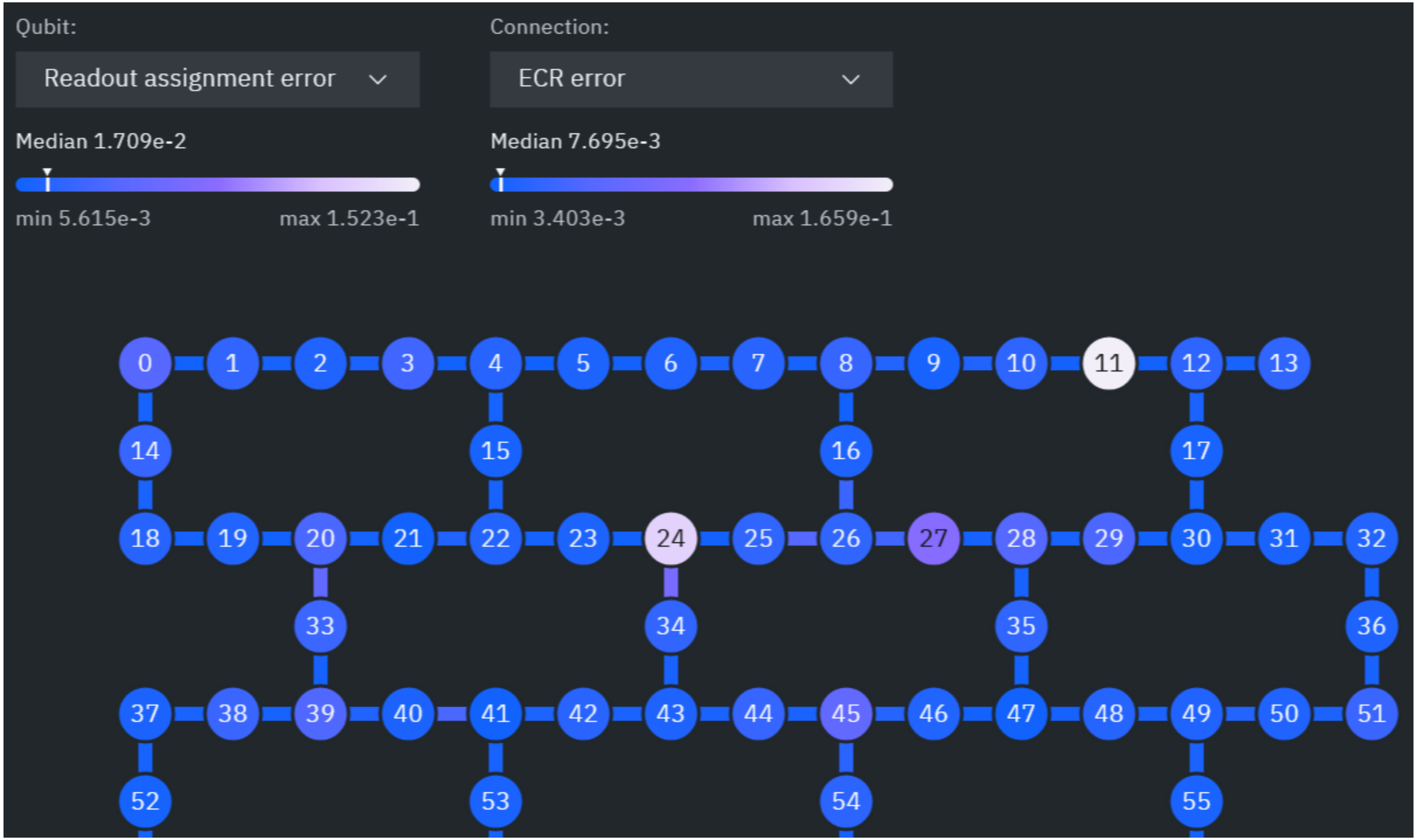


For the second run, the uncorrected fidelity is again lower than the one of the corrected state. From the probabilities in the A matrix it can be seen that the error probabilities of p and q are actually higher than in the run before. This correlates to IBMs real time readout assignment error (152.3 out of 1000 readouts fail):

In [222…

```
from IPython.display import Image
Image('Brisbane_Readout_Error.png')
```

Out[222…



Dynamical Decoupling for Mitigating Phase Error

Since the measurement error correction showed that the dephasing of the qubit state is much more problematic, this is now tried to be mitigated using dynamical decoupling.

In [59]:
```
# Set up a fake_provider backend to run locally
```

```python
from qiskit_ibm_runtime.fake_provider import FakeBrisbane, FakeQuebec, FakeKyiv, FakeSherbrooke
backend = FakeKyiv()

print(backend)
```

```
<qiskit_ibm_runtime.fake_provider.backends.kyiv.fake_kyiv.FakeKyiv object at 0x0000023C2EA5EED0>
```

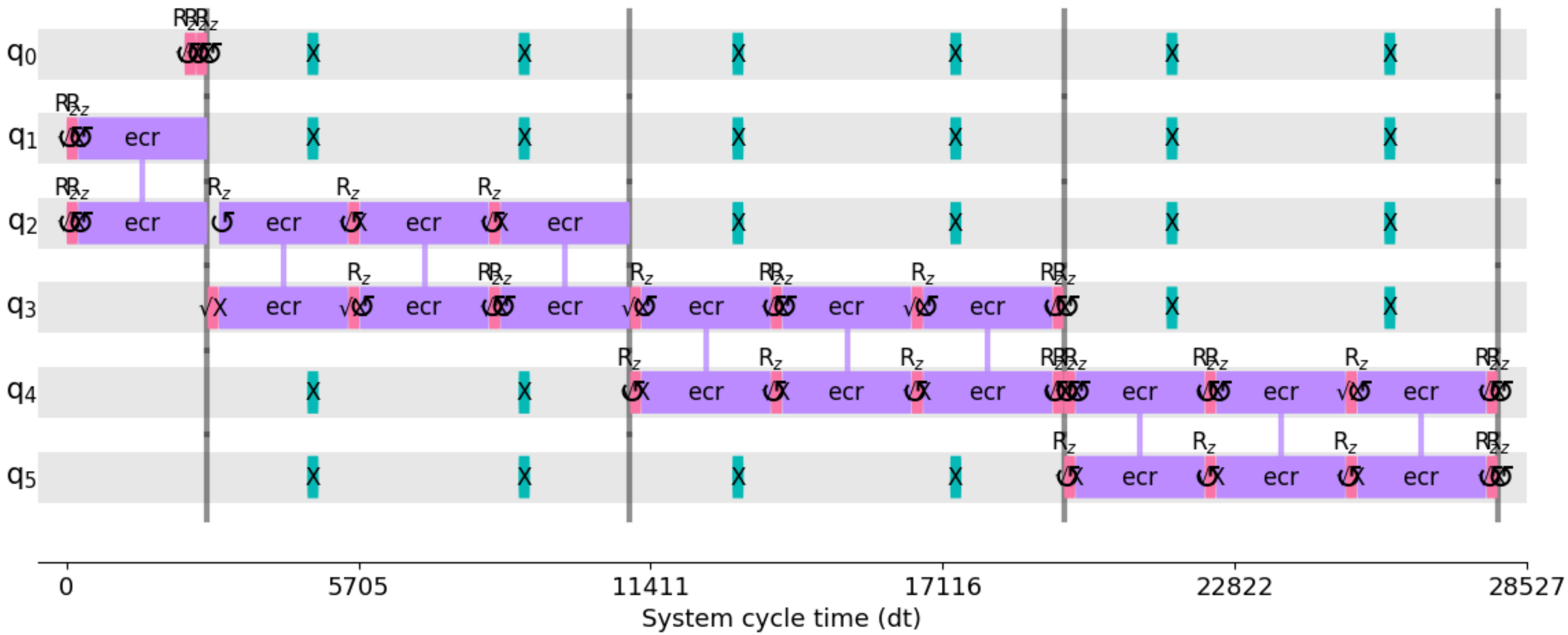
Timing Plots

In [ ]:
```python
# function that draws a timeline plot for the circuit

def draw(circuit):
    from qiskit import transpile

    scheduled = transpile(
        circuit,
        optimization_level=0,
        instruction_durations=InstructionDurations(),
        #scheduling_method="alap",
    )
    return timeline_drawer(scheduled, idle_wires=False)
```

In [ ]:
```python
# timeline for a circuit with three swaps and one XX sequence

num_swaps = 3

dd_sequence = dd_sequence_XX(1)
qc_plot_XX = create_for_given_distance(num_swaps, backend, transpile = False, dd_sequence = dd_sequence, dd = True)
plot = draw(qc_plot_XX)

plot.savefig("1_XX_timeline.png", dpi=300, bbox_inches="tight")

plot
```

```
C:\Users\danie\AppData\Local\Temp\ipykernel_25056\593719193.py:4: DeprecationWarning: ``qiskit.compiler.transpiler.transpile()`
`'s argument ``instruction_durations`` is deprecated as of Qiskit 1.3. It will be removed in Qiskit 2.0. The `target` parameter
should be used instead. You can build a `Target` instance with defined instruction durations with `Target.from_configuration
(..., instruction_durations=...)`
  scheduled = transpile(
```

Out[ ]:

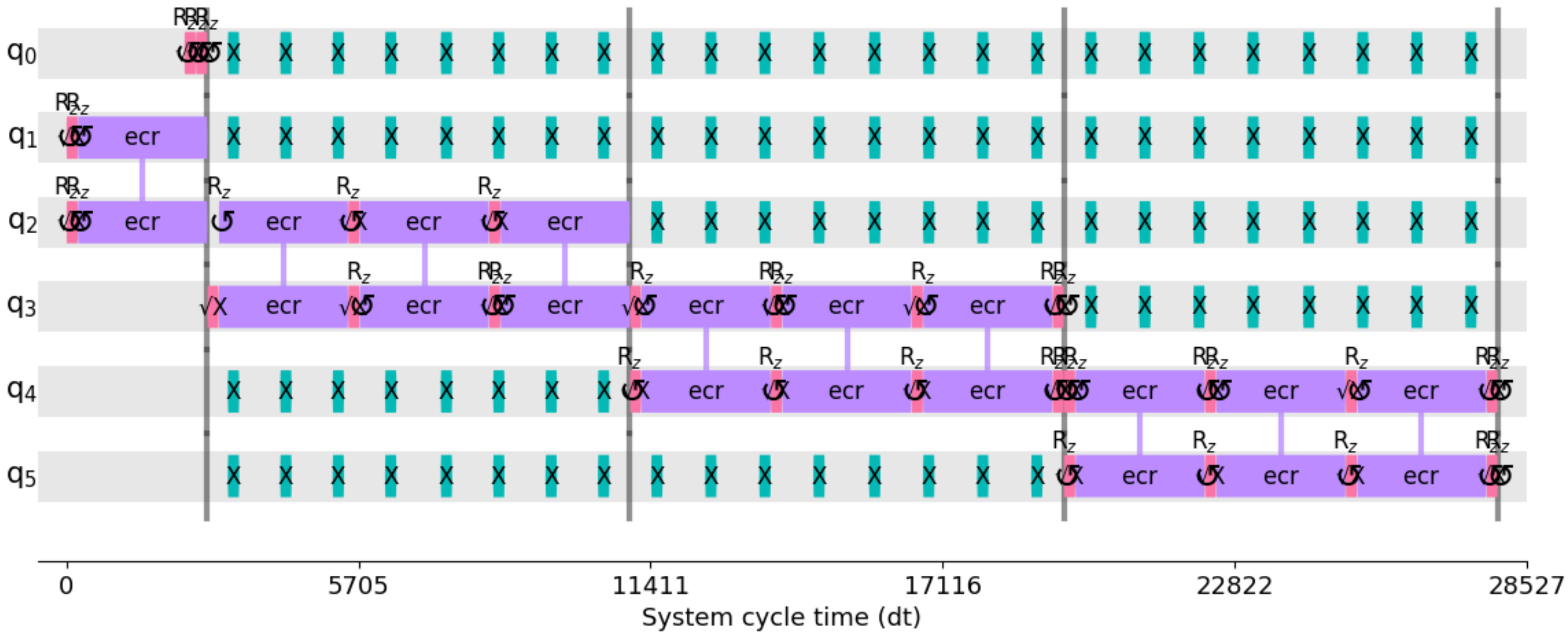


In [ ]:

```
# timeline for a circuit with three swaps and 4 XX sequence

num_swaps = 3

dd_sequence = dd_sequence_XX(4)
qc_plot_XX = create_for_given_distance(num_swaps, backend, transpile = False, dd_sequence = dd_sequence, dd = True)
plot = draw(qc_plot_XX)

plot.savefig("3_XX_timeline.png", dpi=300, bbox_inches="tight")

plot
```

```
C:\Users\danie\AppData\Local\Temp\ipykernel_25056\593719193.py:4: DeprecationWarning: ``qiskit.compiler.transpiler.transpile()`
``'s argument ``instruction_durations`` is deprecated as of Qiskit 1.3. It will be removed in Qiskit 2.0. The `target` parameter
should be used instead. You can build a `Target` instance with defined instruction durations with `Target.from_configuration
(..., instruction_durations=...)`
  scheduled = transpile(
```

Out[ ]:



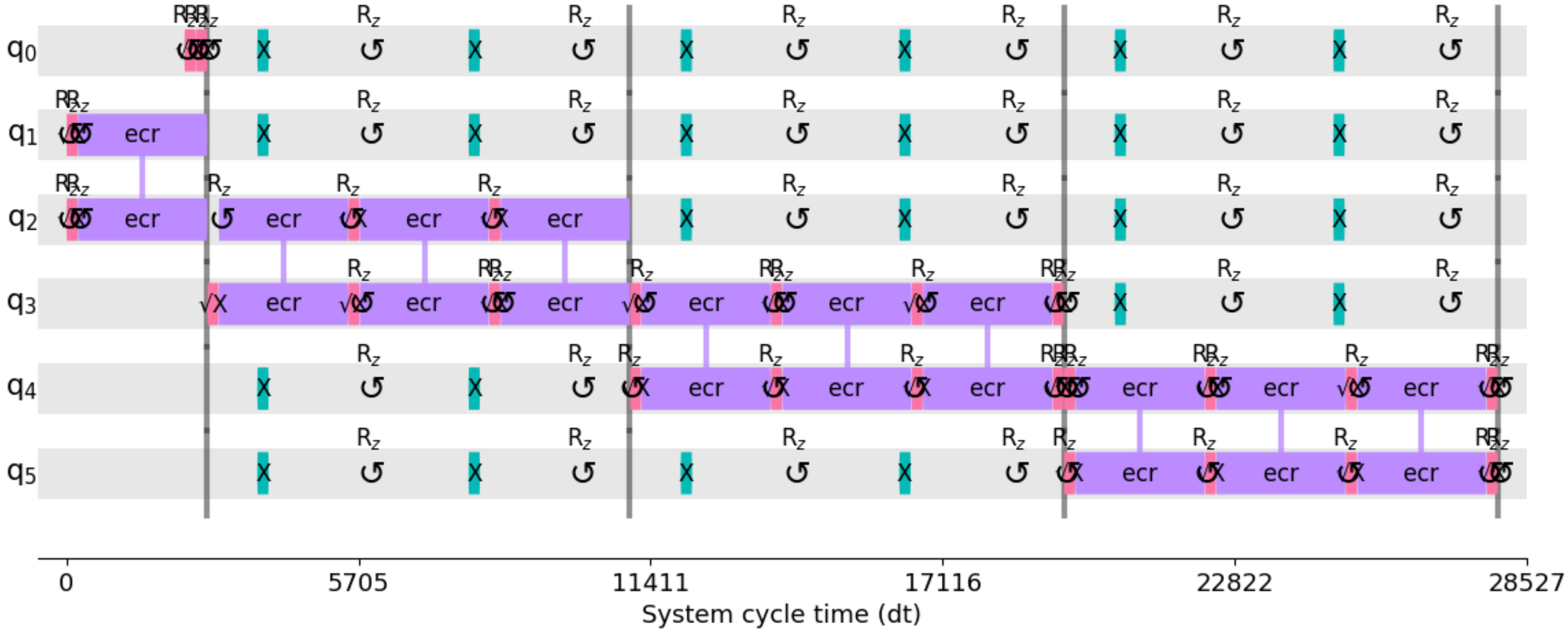
In [ ]:

```python
# timeline for a circuit with three swaps and one XY4 sequence

num_swaps = 3

dd_sequence = dd_sequence_XY4(1)
qc_plot_XY4 = create_for_given_distance(num_swaps, backend, transpile = False, dd_sequence = dd_sequence, dd = True)

plot = draw(qc_plot_XY4)

plot.savefig("3_XY4_timeline.png", dpi=300, bbox_inches="tight")
```

```
plot
```

```
C:\Users\danie\AppData\Local\Temp\ipykernel_25056\593719193.py:4: DeprecationWarning: ``qiskit.compiler.transpiler.transpile()`
``'s argument ``instruction_durations`` is deprecated as of Qiskit 1.3. It will be removed in Qiskit 2.0. The `target` parameter
should be used instead. You can build a `Target` instance with defined instruction durations with `Target.from_configuration
(..., instruction_durations=...)`
  scheduled = transpile(
```

Out[ ]:



In [30]:

```
# Check if backend has invalid relaxation times, T2 should be smaller than 2 * T1
# Fix: Change backend
backend_properties = backend.properties()
for qubit in range(len(backend_properties.qubits)):
    T1 = backend_properties.t1(qubit)
    T2 = backend_properties.t2(qubit)
    if T2 > 2 * T1: print(f"Qubit {qubit}: T1 = {T1}, T2 = {T2}")
```

In [ ]:

```
#real backend
```

```
service = QiskitRuntimeService(
    channel='ibm_quantum',
    instance='ibm-q/open/main',
    token=''
)

# Set the physical backend
#backend = service.backend('ibm_brisbane')
backend = service.backend('ibm_kyiv')
#backend = service.backend('ibm_sherbrooke')

print(backend)
```

```
<IBMBackend('ibm_kyiv')>
```

XX Sequence

testing the dependence on the number of XX sequences

In [ ]:

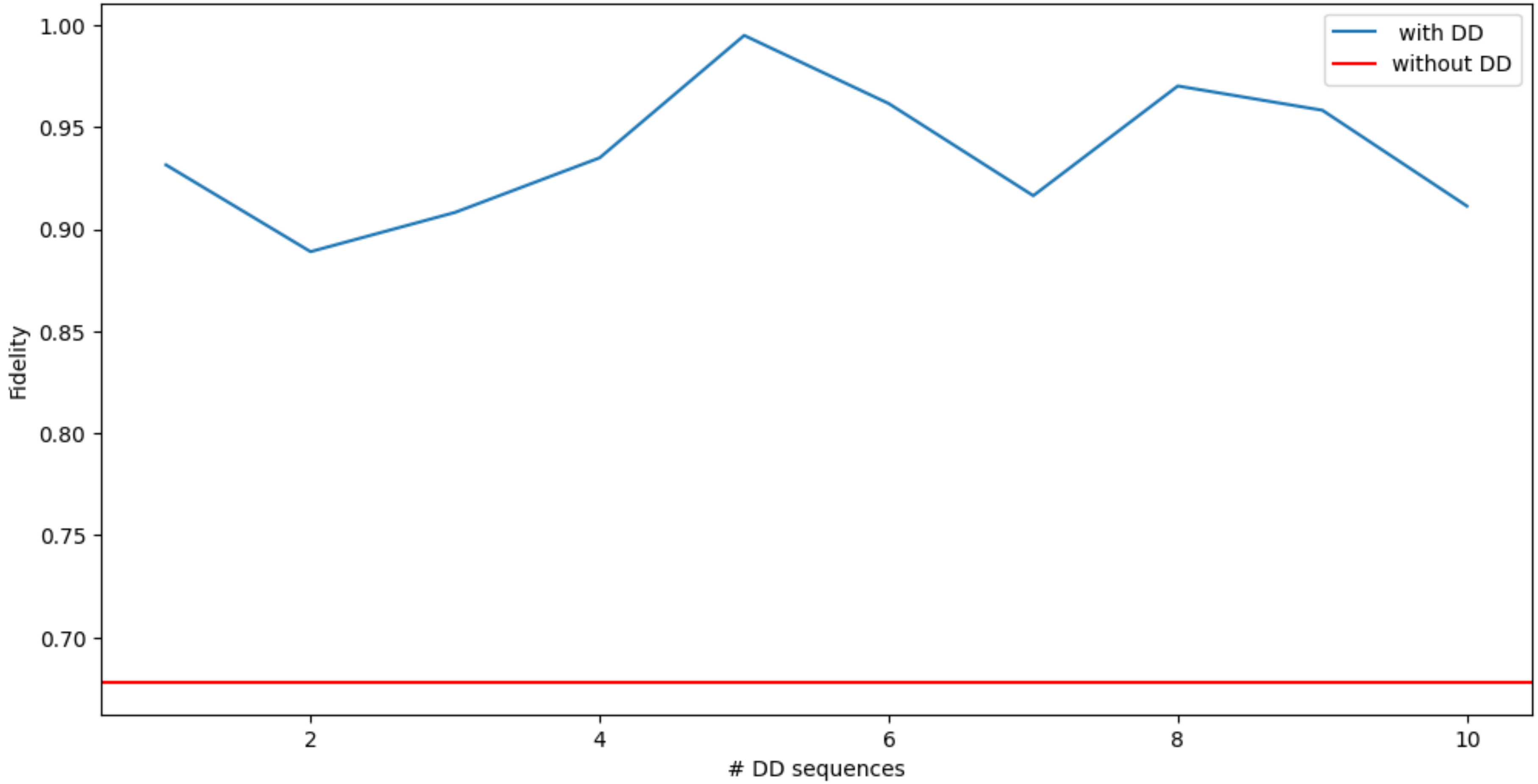
```
# Define the number of shots for each parameter combination for the sampler
N_shots = 1024

n = 10
num_swaps = 5

fidelities_dd_xx = []
fidelity_without = 0

with Session(backend=backend) as session:
    # Set up the Sampler that will run the circuit
    sampler = Sampler(mode=session)

    for i in range(n):

        # Set up the Sampler that will run the circuit
        #sampler = Sampler(backend)

        # creating circuit with dynamical decoupling
        dd_sequence = dd_sequence_XX(i+1)
```

```
        transpiled_circuit = create_for_given_distance(num_swaps, backend, transpile = False, dd_sequence = dd_sequence, dd =

        # Run the sampler for all values of the two parameters
        job = sampler.run([(transpiled_circuit, param_vals, N_shots)])

        # Saves the results from the PUB (for all three parameter combinations)
        pub_results = job.result()[0]

        # Calculate expectation values and fidelity
        expt_vals, fidelity = calc_expt_vals_and_fidelity(pub_results, N_shots)

        fidelities_dd_xx.append(fidelity)

    # creating circuit without dynamical decoupling
    transpiled_circuit = create_for_given_distance(num_swaps, backend, transpile = True, dd = False)

    # Run the sampler for all values of the two parameters
    job = sampler.run([(transpiled_circuit, param_vals, N_shots)])

    # Saves the results from the PUB (for all three parameter combinations)
    pub_results = job.result()[0]

    # Calculate expectation values and fidelity
    expt_vals, fidelity = calc_expt_vals_and_fidelity(pub_results, N_shots)
    fidelity_without = fidelity
```

In [61]:
```
# save values in extern file

with open("number_of_XX_fidelities.txt", "w") as f:
    for value in fidelities_dd_xx:
        f.write(f"{value}\n")

    f.write(f"\n")
    f.write(f"{fidelity_without}\n")
```

In [59]:
```
# Create figure
plt.figure(figsize=(12, 6))
plt.plot(np.arange(1, 11, 1), fidelities_dd_xx, label=' with DD')
plt.axhline(y=fidelity_without, color='red', label='without DD')
```

```
# Label axes
plt.xlabel("# DD sequences")
plt.ylabel("Fidelity")

plt.legend()

plt.savefig("number_of_XX.png", dpi=300, bbox_inches="tight")

# Display plot
plt.show()
```



The differences in the fidelities for the different numbers of XX sequences are probably random.

Run for a fixed number of XX sequences

In [ ]:

```
# Define the number of shots for each parameter combination for the sampler
N_shots = 2048

num_swaps = 5
num_XX = 5
num_XY4 = 3

fidelity_XX = 0
expt_vals_XX = 0

fidelity_XY4 = 0
expt_vals_XY4 = 0

fidelity_without = 0
expt_vals_without = 0

with Session(backend=backend) as session:

    # XX


    # Set up the Sampler that will run the circuit
    sampler = Sampler(mode=session)

    # creating circuit with dynamical decoupling
    dd_sequence = dd_sequence_XX(num_XX)
    transpiled_circuit = create_for_given_distance(num_swaps, backend, transpile = False, dd_sequence = dd_sequence, dd = True

    # Run the sampler for all values of the two parameters
    job = sampler.run([(transpiled_circuit, param_vals, N_shots)])

    # Saves the results from the PUB (for all three parameter combinations)
    pub_results = job.result()[0]

    # Calculate expectation values and fidelity
```

```python
    expt_vals_XX, fidelity_XX = calc_expt_vals_and_fidelity(pub_results, N_shots)


    # XY4



    # creating circuit with dynamical decoupling
    dd_sequence = dd_sequence_XY4(num_XY4)
    transpiled_circuit = create_for_given_distance(num_swaps, backend, transpile = False, dd_sequence = dd_sequence, dd = True

    # Run the sampler for all values of the two parameters
    job = sampler.run([(transpiled_circuit, param_vals, N_shots)])

    # Saves the results from the PUB (for all three parameter combinations)
    pub_results = job.result()[0]

    # Calculate expectation values and fidelity
    expt_vals_XY4, fidelity_XY4 = calc_expt_vals_and_fidelity(pub_results, N_shots)



    # without dynamical decoupling



    # creating circuit without dynamical decoupling
    transpiled_circuit = create_for_given_distance(num_swaps, backend, transpile = True, dd = False)

    # Run the sampler for all values of the two parameters
    job = sampler.run([(transpiled_circuit, param_vals, N_shots)])

    # Saves the results from the PUB (for all three parameter combinations)
    pub_results = job.result()[0]

    # Calculate expectation values and fidelity
    expt_vals_without, fidelity_without = calc_expt_vals_and_fidelity(pub_results, N_shots)
```

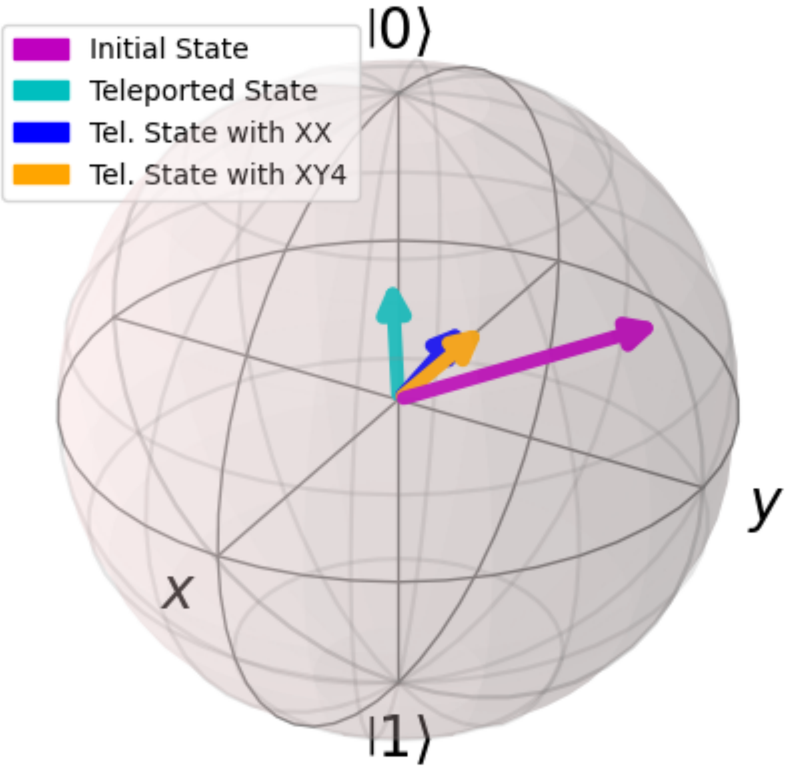
In [16]:

```python
print(fidelity_without)
print(fidelity_XX)
```

```
print(fidelity_XY4)
```

```
0.7251862630554938
0.9526118362832163
0.973274490906637
```

In [22]:
```
# Define the initial and the teleported bloch vector and plot them on the sphere
bloch_vectors = [
    ([0, np.sin(pi/3), np.cos(pi/3)], 'Initial State', 'm'),
    (expt_vals_without, 'Teleported State', 'c'),
    (expt_vals_XX, 'Tel. State with XX', 'b'),
    (expt_vals_XY4, 'Tel. State with XY4', 'orange')
]

plot_on_bloch_sphere(bloch_vectors)
```



Conclusion

The first approach of mitigating the readout measurement error did not help in increasing the fidelity between the initial and the teleported state. However, the determination of the probability transfer matrix gave an interesting look at the actual readout errors of the real QPUs. It was found that the readout error for the $|1\rangle$ state was consistently higher than the one for the $|0\rangle$ state. However, this can not lead directly to the assumption that the implemented measurement is worse for the $|1\rangle$ state because the higher error probability could also be a result of the differently prepared states. In fact, for the $|0\rangle$ state measurement the circuit was simply empty, only consisting of one measurement. For the $|1\rangle$ state, however, an X gate had to be applied first which could introduce additional errors that are represented in the value for q. We have no way of differentiating between the additional gate errors and the actual readout error. All in all, both errors were so small that the readout error correction did not help in reducing the fidelity. Instead, the dephasing was tackled.

Since we have long idle times in our circuit, the states on our qubits will dephase over time. One way of dealing with this error is Dynamic Decoupling, which we implemented by using PadDynamicalDecoupling class from qiskit. Because this class needs the duration of all the used gates and we couldn't get the duration of the switch gate, we had to apply the switch case after inserting the Dynamic Decoupling sequences. This caused another problem, because we would have to transpile the whole circuit again, but this changes more than just the gates so the Dynamical Decoupling gives us an error. Our solution to this problem was using the BasisTranslator class from qiskit, which just replaces the gates with basis gates but leaves the dynamic decoupling intact. However using the BasisTranslator gave us an error on all the backends except kyiv, but we couldn't figure out what caused the problem, so we just used kyiv for our Dynamical Decoupling runs. Before we did our final runs, we wanted to test if the number of Dynamical Decoupling sequences we inserted had an influence on how well the error mitigation works. In our plot of the fidelities we couldn't identify a clear dependency on the number of sequences, so we just used one constant number of sequences for our final run. For our final runs we calculated the fidelity once without Dynamical Decoupling, once with XX-sequences and once with XY4-sequences inserted. We implemented the XY4 sequence with a RZ-Gate instead of a Y-Gate, because the Y-Gate is not a basis gate for our backend. The fidelities improved significantly by using Dynamical Decoupling. It can also be seen that the XY4-sequence produces a slightly highter fidelity than the XX-sequence.