Tackling systematic off-resonant errors with CORPSE pulses: Theory and Demonstration using Qiskit

Summary: To rotate qubits on a quantum computer, one might drive the qubit with an electromagnetic pulse of a given driving frequency for a specific duration. However, when the driving frequency of this pulse differs from the resonant frequency of the qubit, the actually realized rotation is imperfect, and whose angle and axis of rotation differ from those intended. This is a major problem for quantum hardware, because it is very hard to avoid systematic errors in the driving pulses' frequency. However, by choosing to implement a single rotation as a carefully chosen series of rotations, called a composite rotation, we may be able to substantially reduce the degree of error. This project examines one such example: CORPSE.

The variables and notation in this project differ from one or more of the resources mentioned below, and so the results may also be slightly different.

Sources:

- [1] https://arxiv.org/pdf/1209.4247.pdf
- [2] https://arxiv.org/pdf/quant-ph/0208092.pdf

(This seemingly empty cell is for markdown formatting.)

1. Introduction

An ideal single-qubit rotation with rotation axis in the xy-plane takes the form:

$$R(\theta, \phi) = \exp\left[-i\theta \mathbf{n(\phi)} \ \cdot \sigma/2\right]$$

- θ: rotation angle
- ϕ: azimuthal angle, specifying the rotation axis within the xy-plane $\mathbf{n(\phi)} = (\cos\phi, \sin\phi, 0)$
- $\sigma = (\sigma_x, \sigma_y, \sigma_z)$: Pauli vector operator

We will call pulses of this form **elementary pulses**.

In reality, such elementary rotations are imperfectly implemented on real quantum backends due to inevitable errors in the driving freqency, and the actually realized rotation is:

$$R'(\theta, \phi) = R(\theta, \phi) + O(\epsilon)$$

- ϵ quantifies the degree of error.
  - $O(\epsilon)$ is the total error term.

Suppose we have a sequence of N elementary pulses $\{R(\theta_i, \phi_i)\}$ each with a degree of error $O(\epsilon)$:

$$U(\theta, \phi) = R(\theta_N, \phi_N) R(\theta_{N-1}, \phi_{N-1}) \ldots R(\theta_1, \phi_1)$$

where $U(\theta, \phi)$ is constructed to be as close to $R(\theta, \phi)$ as possible. Then it is possible to show by simply multiplying all the elementary pulses together that:

$$U'(\theta, \phi) = R(\theta, \phi) - i\epsilon\delta U + O(\epsilon^2)$$

- $U'(\theta, \phi)$: actually realized $U(\theta, \phi)$ in the prescence of error
- δU: functional form of the error term that is first order in ϵ
- $O(\epsilon^2)$: error term second-order in ϵ and higher

The N elementary pulses $\{R(\theta_i, \phi_i)\}$ compose of a **composite pulse** if they are chosen to make δU=0, so that the degree of error in U' is only of second order or higher:

$$U'(\theta, \phi) = R(\theta, \phi) + O(\epsilon^2)$$

Such composite pulses effectively mitigate the total error term, since the term first-order in ϵ is the most substantial contribution.

2. Off-resonant Errors (ORE)

An ideal elementary pulse has a rotation axis in the xy-plane. However, when an ORE is present, the actual rotation axis has a component in the z-axis. Then the actually realized elementary pulse is:

$$R'_f(\theta, \phi) = \exp\left[-i\theta(\mathbf{n}(\phi) \cdot \sigma + f\sigma_z)/2\right] \approx R(\theta, \phi) - if\sin(\theta/2)\sigma_z$$

- f is some constant (unknown to the experimenter), quantifying the strength of the ORE

As discussed in references [1], [2], The **CORPSE pulse sequence (Concatenated Composite Pulses Compensating Simultaneous Systematic Errors)** is a composite pulse designed to suppress the deleterious effects of ORE.

For a target rotation with parameters θ, ϕ, a CORPSE pulse sequence consists of three elementary pulses indexed by $1, 2, 3$, the order in which they are applied to the qubit:

Rotation Index	Rotation Angle	Azimuthal Angle
1	$2n_1\pi + \theta/2 - k$	ϕ
2	$2n_2\pi - 2k$	$\phi + \pi$
3	$2n_3\pi + \theta/2 - k$	ϕ

- $k = \arcsin[\sin(\theta/2)/2]$
- $n_i \in \mathbb{Z}$

CORPSE usually takes $n_1 = 1, n_2 = 1, n_3 = 0$ (choosing different values for these integers yields slight variations of CORPSE). Then our CORPSE pulse sequence is summarized as:

Rotation Index	Rotation Angle	Azimuthal Angle
1	$2\pi + \theta/2 - k$	ϕ
2	$2\pi - 2k$	$\phi + \pi$
3	$\theta/2 - k$	ϕ

Where do these values come from? In short,the authors of reference [2] derive these results by taylor expanding the composite rotation as a function of the off-resonant error, and solving for the conditions which cause the first-degree term to vanish.

3. An Interactive example

a) Theory

For the reminder of this project, we assume that the ideal goal is to implement an elementary θ = $180^\circ = \pi$ rotation—for example, if we want to rotate a qubit from $|0\rangle$ to $|1\rangle$.

Also, **for the reminder of this project, suppose that our ideal rotation axis is the +x-axis**, corresponding to $\mathbf{n}(\phi) = (\cos\phi, \sin\phi, 0) = (1, 0, 0)$. Hence, $\phi = 0$.

From the table in section 2, the CORPSE pulse sequence corresponding to that target rotation can be calculated:

In [176…

```
import numpy as np

print('\033[1m' + "Target rotation" + '\033[0m')
theta = np.pi
print(f"theta:", theta*(180/np.pi), "degrees")

print('\033[1m' + "CORPSE sequence" + '\033[0m')
theta1 = 2*np.pi + theta/2 - np.arcsin(np.sin(theta/2)/2)
print(f"theta1:", theta1*(180/np.pi), "degrees, along the x axis")

theta2 = 2*np.pi - 2*np.arcsin(np.sin(theta/2)/2)
print(f"theta2:", theta2*(180/np.pi), "degrees, along the -x axis")

theta3 = theta/2 - np.arcsin(np.sin(theta/2)/2)
print(f"theta3:", round(theta3*(180/np.pi), 2), "degrees, along the x axis")
```

```
Target rotation
theta: 180.0 degrees
CORPSE sequence
theta1: 420.0 degrees, along the x axis
theta2: 300.0 degrees, along the -x axis
theta3: 60.0 degrees, along the x axis
```

Lets demonstrate the performance of this CORPSE pulse. Suppose $f = 0.1$. Then the actually realized elementary pulse is:

$$R'_f(\theta, \phi) = \exp\left[-i\theta(\mathbf{n}(\phi) \cdot \sigma + f\sigma_z)/2\right]$$

$$\approx R(\theta, \phi) - if \sin(\theta/2)\sigma_z$$

$$= R(\theta, \phi) - i(0.1) \sin(\theta/2)\sigma_z$$

$$= \exp\left[-i\theta\sigma_x/2\right] - i(0.1) \sin(\theta/2)\sigma_z$$

$$= \cos(\theta/2)I - i \sin(\theta/2)\sigma_x - i(0.1) \sin(\theta/2)\sigma_z$$

$$= \cos(\pi/2)I - i \sin(\pi/2)\sigma_x - i(0.1) \sin(\pi/2)\sigma_z$$

$$= -i\sigma_x - 0.1i\sigma_z$$

$$= -i(\sigma_x - 0.1\sigma_z)$$

Although this approximate expression is great for us to get an intuitive feel for what the off-resonant error looks like (since it is a linear combination of the sum of the desired term, σ_x, and the error term, σ_z, up to a global phase of $-i$), we cannot actually plot this on the bloch sphere because this expression is not unitary.

We employ a convenient fix of this problem.

Define $\bar{\sigma} = (\frac{1}{\sqrt{1+\gamma^2}}, 0, \frac{\gamma}{\sqrt{1+\gamma^2}}) \cdot \sigma$

$$= \frac{\gamma}{\sqrt{1+\gamma^2}}\sigma_z + \frac{1}{\sqrt{1+\gamma^2}}\sigma_x$$

- γ: alternate parameter that quantifies the degree of ORE
  - γ can be any real number
- σ: Pauli vector $(\sigma_x, \sigma_y, \sigma_z)$

$\bar{\sigma}$ is a unitary operator because the vector $(\frac{1}{\sqrt{1+\gamma^2}}, 0, \frac{\gamma}{\sqrt{1+\gamma^2}})$ has unit length: $(\frac{1}{\sqrt{1+\gamma^2}})^2 + 0^2 + (\frac{\gamma}{\sqrt{1+\gamma^2}})^2 = 1.$

Following the above assumptions, the actually realized elementary pulse is:

$$U = \exp\left[-i\theta\bar{\sigma})/2\right]$$

$$= \cos(\theta/2)I - i \sin(\theta/2)\bar{\sigma}$$

$$= \cos(\theta/2)I - i \sin(\theta/2)(\frac{\gamma}{\sqrt{1+\gamma^2}}\sigma_z + \frac{1}{\sqrt{1+\gamma^2}}\sigma_x)$$

As a sanity check, let's check what happens in the case when there is no error, $\gamma = 0$. Then

$= \cos(\theta/2)I - i \sin(\theta/2)(\frac{0}{\sqrt{1+0^2}}\sigma_z + \frac{1}{\sqrt{1+0^2}}\sigma_x) = \cos(\theta/2)I - i \sin(\theta/2)\sigma_x$, as desired.

b) Play around!

In the following subsection, we will visualize all this using Qiskit!

Let us first find the ideal elementary rotation from $|0\rangle$ to $|1\rangle$, and the corresponding resulting state.

In [177…

```
from qiskit.quantum_info.operators import Operator
from qiskit.opflow import I, X, Y, Z

theta = np.pi

ideal = np.cos(theta/2) * I - (1j) * np.sin(theta/2) * X
ideal_op = Operator(ideal)

from qiskit.visualization import array_to_latex
array_to_latex(ideal_op, prefix = "\\text{Ideal Rotation:}")
```

Out[177…

$$\text{Ideal Rotation:} \begin{bmatrix} 0 & -i \\ -i & 0 \end{bmatrix}$$

In [178…

```
array_to_latex([0, 1], prefix = "\\text{Ideal Resulting Vector:}")
```

Out[178…

$$\text{Ideal Resulting Vector:} \begin{bmatrix} 0 & 1 \end{bmatrix}$$

How about an actually realized rotation from $|0\rangle$ to $|1\rangle$ in the prescence of ORE? Play around with the value of γ below, and run the subsequent cells.

Although γ can be any real number, we set the slider over a representative range of 0 to 20.

In [179…

```
import ipywidgets as widgets
from IPython.display import display

gamma_slider = widgets.FloatSlider(
    value = 0.1,
    min = 0,
    max = 20,
    step = 0.1,
    description = r'$\gamma$=',
    continuous_update = True
)

widgets.VBox([gamma_slider],
             layout = widgets.Layout(align_items = 'center'))
```

In [180…

```
gamma = gamma_slider.value

actual = np.cos(theta/2) * I - (1j) * np.sin(theta/2) * (
        (gamma / np.sqrt(1 + gamma ** 2)) * Z + (1 / np.sqrt(1 + gamma ** 2) * X
    )
actual_op = Operator(actual)
```

```python
array_to_latex(actual_op, prefix = "\\text{Actually Realized Rotation:}")
```

Out[180…

$$\text{Actually Realized Rotation:} \begin{bmatrix} -0.0995i & -0.99504i \\ -0.99504i & 0.0995i \end{bmatrix}$$

In [181…

```python
# create quantum circuit
from qiskit import QuantumCircuit
from qiskit.circuit import ClassicalRegister, QuantumRegister

cr = ClassicalRegister(1, name = "cr")
qr = QuantumRegister(1, name = "qr")
qc = QuantumCircuit(cr, qr)

qc.unitary(actual_op, 0, label = "Actually Realized Rotation")
qc.draw()
```

Out[181…

```
      ┌────────────────────────────┐
qr_0: ┤ Actually Realized Rotation ├
      └────────────────────────────┘
cr: 1/══════════════════════════════
```

In [182…

```python
# execute circuit
from qiskit import Aer, execute

statevec_sim = Aer.get_backend('statevector_simulator')
job = execute(experiments = qc, backend = statevec_sim)
result_vector = job.result().get_statevector()
result_vector_adjustedphase = np.array(result_vector)

array_to_latex(result_vector_adjustedphase,
               prefix = "\\text{Actual Resulting Statevector:}")
```

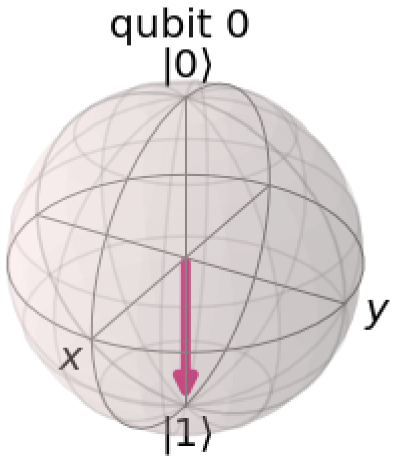
Out[182…

$$\text{Actual Resulting Statevector:} \begin{bmatrix} -0.0995i & -0.99504i \end{bmatrix}$$

In [183…

```python
# visualize results
from qiskit.visualization import plot_bloch_multivector
from matplotlib.pyplot import plot as plt
%matplotlib inline

idealrot_fig = plot_bloch_multivector([0,1])
idealrot_fig.suptitle('Ideal resulting vector in the absence of ORE', y = 0.1)
idealrot_fig
```

Out[183…

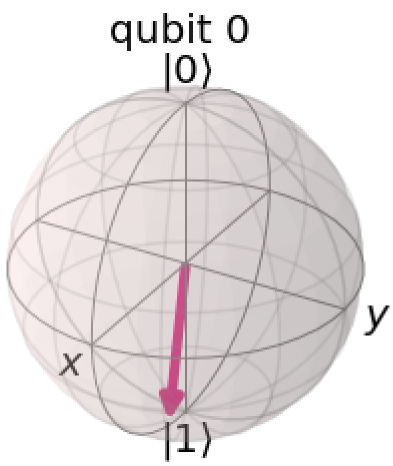


Ideal resulting vector in the absence of ORE

In [184…

```
actualrot_fig = plot_bloch_multivector(result_vector_adjustedphase)
actualrot_fig.suptitle(r'Actually realized resulting vector for $\gamma$ =' + f'
actualrot_fig
```

Out[184…



Actually realized resulting vector for γ =0.1

To quantify how good or bad this result is, we can measure the state fidelity between the ideal and actual state vectors.

In [185…

```
from qiskit.quantum_info import state_fidelity
print('\033[1m' + 'State Fidelity:'+ '\033[0m', str(state_fidelity(result_vector
```

State Fidelity: 0.9900990099009903

4. A deeper dive

To get a better feel for the impact of this ORE on the qubit's rotation, let us plot the state fidelity as a function of γ from 0 to 20.

In [186…

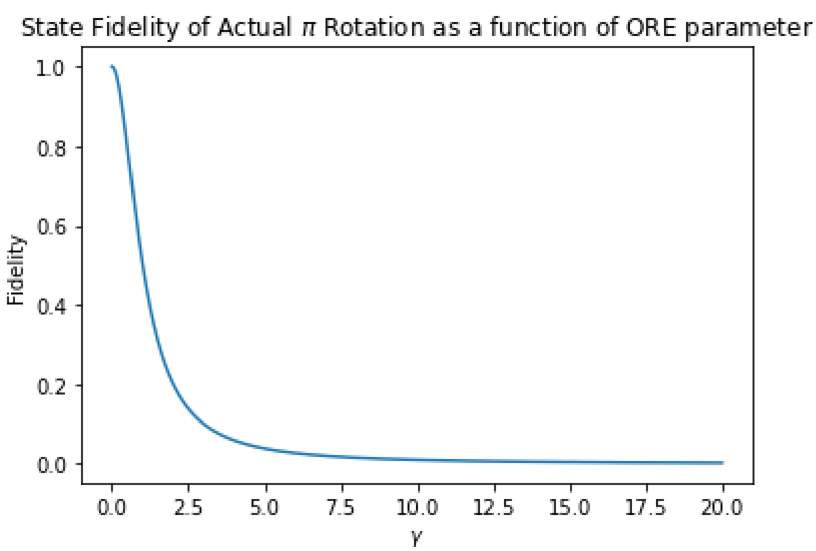
```
gamma_array = np.linspace(start = 0, stop = 20, num = 500)
fidelity_array = []
for gamma in gamma_array:
```

```
    # create actual rot op
    actual = np.cos(theta/2) * I - (1j) * np.sin(theta/2) * (
        (gamma / np.sqrt(1 + gamma ** 2)) * Z + (1 / np.sqrt(1 + gamma ** 2) * X
    )
    actual_op = Operator(actual)
    # create qc
    cr = ClassicalRegister(1, name = "cr")
    qr = QuantumRegister(1, name = "qr")
    qc = QuantumCircuit(cr, qr)
    qc.unitary(actual_op, 0, label = "Actually Realized Rotation")
    # run qc
    statevec_sim = Aer.get_backend('statevector_simulator')
    job = execute(experiments = qc, backend = statevec_sim)
    # get resulting statevector
    result_vector = job.result().get_statevector()
    result_vector_adjustedphase = np.array(result_vector)*1j # times i to cancel
    # calculate fidelity
    fidelity_value = state_fidelity(result_vector_adjustedphase, [0 , 1])
    fidelity_array.append(fidelity_value)
```

In [187…

```
# plot results
import matplotlib.pyplot as plt
plt.plot(gamma_array, fidelity_array)
plt.title("State Fidelity of Actual $\pi$ Rotation as a function of ORE paramete
plt.xlabel(r"$\gamma$")
plt.ylabel("Fidelity")
```

Out[187… Text(0, 0.5, 'Fidelity')



As expected, the fidelity is 1 when there is no ORE. When γ increases, the fidelity monotonically decreases.

Let's see what happens with a CORPSE pulse. Lets again start off with an interactive example. Again, choose the value of the ORE parameter below. We name the slider a differently to avoid conflicting with the previous interactive example's slider.

In [188…

```
gamma2_slider = widgets.FloatSlider(
    value = 0.1,
    min = 0,
```

```
    max = 20,
    step = 0.1,
    description = r'$\gamma$=',
    continuous_update = True
)

widgets.VBox([gamma2_slider],
             layout = widgets.Layout(align_items = 'center'))
```

In [189…

```
# calculate CORPSE rotation angles
gamma = gamma2_slider.value
theta = np.pi

CORPSE1_theta = 2*np.pi + theta/2 - np.arcsin(np.sin(theta/2)/2)
CORPSE2_theta = 2*np.pi - 2* np.arcsin(np.sin(theta/2)/2)
CORPSE3_theta = theta/2 - np.arcsin(np.sin(theta/2)/2)

print("CORPSE rotation 1 degrees:", CORPSE1_theta * 180/np.pi, " about +X axis")
print("CORPSE rotation 2 degrees:", CORPSE2_theta * 180/np.pi, " about -X axis")
print("CORPSE rotation 3 degrees:", CORPSE3_theta * 180/np.pi, " about +X axis")
```

```
CORPSE rotation 1 degrees: 419.99999999999994  about +X axis
CORPSE rotation 2 degrees: 300.00000000000006  about -X axis
CORPSE rotation 3 degrees: 60.00000000000001  about +X axis
```

In [190…

```
# create CORPSE operators
CORPSE1 = np.cos(CORPSE1_theta/2) * I - (1j) * np.sin(CORPSE1_theta/2) * (
        (gamma / np.sqrt(1 + gamma ** 2)) * Z + (1 / np.sqrt(1 + gamma ** 2) * X
    )

CORPSE2 = np.cos(CORPSE2_theta/2) * I - (1j) * np.sin(CORPSE2_theta/2) * (
        (gamma / np.sqrt(1 + gamma ** 2)) * Z + (1 / np.sqrt(1 + gamma ** 2) * (
    )

CORPSE3 = np.cos(CORPSE3_theta/2) * I - (1j) * np.sin(CORPSE3_theta/2) * (
        (gamma / np.sqrt(1 + gamma ** 2)) * Z + (1 / np.sqrt(1 + gamma ** 2) * X
    )

CORPSE1_op = Operator(CORPSE1)
CORPSE2_op = Operator(CORPSE2)
CORPSE3_op = Operator(CORPSE3)
```

In [191…

```
# create quantum circuit
cr = ClassicalRegister(1, name = "cr")
qr = QuantumRegister(1, name = "qr")
qc = QuantumCircuit(cr, qr)
qc.unitary(CORPSE1_op, 0, label = "CORPSE rotation 1")
qc.unitary(CORPSE2_op, 0, label = "CORPSE rotation 2")
qc.unitary(CORPSE3_op, 0, label = "CORPSE rotation 3")
qc.draw()
```

Out[191…

```
qr_0: ┤ CORPSE rotation 1 ├┤ CORPSE rotation 2 ├┤ CORPSE rotation 3 ├
cr: 1/═══════════════════════════════════════════════════════════════
```

In [192…

```
# execute quantum circuit
statevec_sim = Aer.get_backend('statevector_simulator')
job = execute(experiments = qc, backend = statevec_sim)
result_vector = job.result().get_statevector()
result_vector_adjustedphase = np.array(result_vector)
result_vector_adjustedphase

print('\033[1m' + 'State Fidelity:'+ '\033[0m', str(state_fidelity(result_vector
```

State Fidelity: 0.9999262351487573

Depending on what you chose for γ, the fidelity might have been much worse or much better than you expected. Again, let's plot the state fidelity as a function over the representative rang of γ and see what happens for CORPSE.

In [193…

```
gamma_array_2 = np.linspace(start = 0, stop = 20, num = 500)
fidelity_array_2 = []
for gamma in gamma_array_2:
    # create actual rot ops
    CORPSE1_theta = 2*np.pi + theta/2 - np.arcsin(np.sin(theta/2)/2)
    CORPSE2_theta = 2*np.pi - 2* np.arcsin(np.sin(theta/2)/2)
    CORPSE3_theta = theta/2 - np.arcsin(np.sin(theta/2)/2)

    CORPSE1 = np.cos(CORPSE1_theta/2) * I - (1j) * np.sin(CORPSE1_theta/2) * (
            (gamma / np.sqrt(1 + gamma ** 2)) * Z + (1 / np.sqrt(1 + gamma ** 2)
        )

    CORPSE2 = np.cos(CORPSE2_theta/2) * I - (1j) * np.sin(CORPSE2_theta/2) * (
            (gamma / np.sqrt(1 + gamma ** 2)) * Z + (1 / np.sqrt(1 + gamma ** 2)
        )

    CORPSE3 = np.cos(CORPSE3_theta/2) * I - (1j) * np.sin(CORPSE3_theta/2) * (
            (gamma / np.sqrt(1 + gamma ** 2)) * Z + (1 / np.sqrt(1 + gamma ** 2)
        )

    CORPSE1_op = Operator(CORPSE1)
    CORPSE2_op = Operator(CORPSE2)
    CORPSE3_op = Operator(CORPSE3)
    # create qc
    cr = ClassicalRegister(1, name = "cr")
    qr = QuantumRegister(1, name = "qr")
    qc = QuantumCircuit(cr, qr)
    qc.unitary(CORPSE1_op, 0, label = "CORPSE rotation 1")
    qc.unitary(CORPSE2_op, 0, label = "CORPSE rotation 2")
    qc.unitary(CORPSE3_op, 0, label = "CORPSE rotation 3")
    # run qc
    statevec_sim = Aer.get_backend('statevector_simulator')
    job = execute(experiments = qc, backend = statevec_sim)
    # get resulting statevector
    result_vector = job.result().get_statevector()
    result_vector_adjustedphase = np.array(result_vector)*1j # times i to cancel
    # calculate fidelity
    fidelity_value = state_fidelity(result_vector_adjustedphase, [0 , 1])
    fidelity_array_2.append(fidelity_value)
```

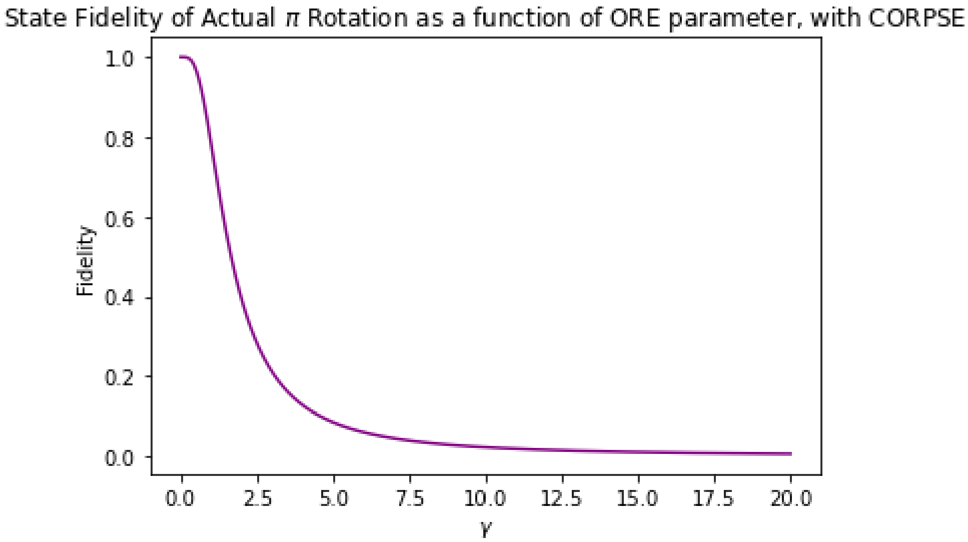
In [194…

```
# plot results
import matplotlib.pyplot as plt
plt.plot(gamma_array_2, fidelity_array_2, color = "purple")
```

```
plt.title("State Fidelity of Actual $\pi$ Rotation as a function of ORE paramete
plt.xlabel(r"$\gamma$")
plt.ylabel("Fidelity")
```

Out[194…

```
Text(0, 0.5, 'Fidelity')
```



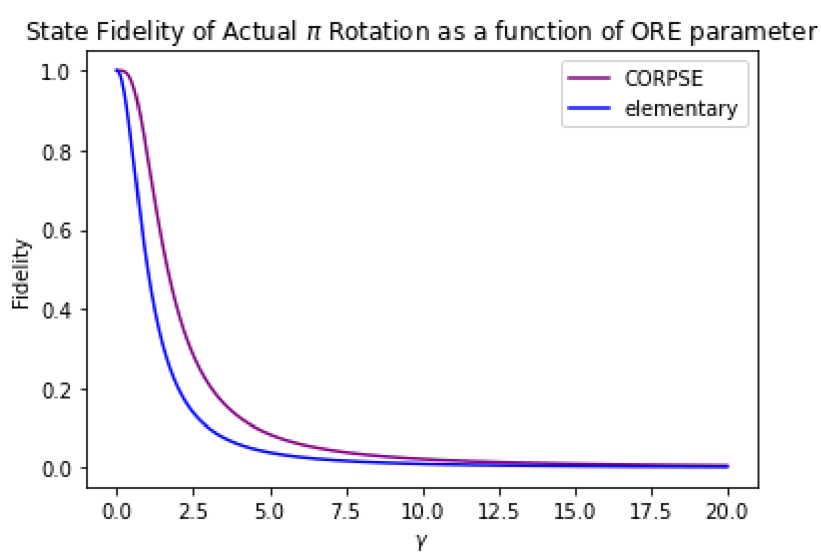
Let's plot the two cases together for direct comparison.

In [195…

```
from matplotlib.lines import Line2D

# compare CORPSE with previous case
plt.plot(gamma_array_2, fidelity_array_2, color = "purple")
plt.plot(gamma_array, fidelity_array, color = "blue")
plt.title(r"State Fidelity of Actual $\pi$ Rotation as a function of ORE paramet
plt.xlabel(r"$\gamma$")
plt.ylabel("Fidelity")

legend_elements = [Line2D([0], [0], color = 'purple', label = 'CORPSE'),
                   Line2D([0], [0], color = 'blue', label = 'elementary')]
plt.legend(handles = legend_elements)
```

Out[195…

```
<matplotlib.legend.Legend at 0x12f9762e0>
```

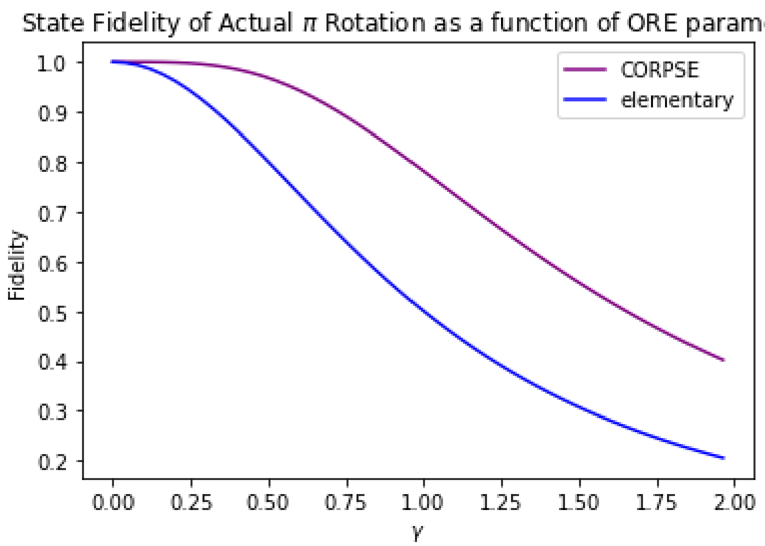


Since the fidelity drops asympotically towards zero quickly, let's compare the two plots again but within a much narrower range.

In [196…

```
# compare CORPSE with previous case
plt.plot(gamma_array_2[0:50], fidelity_array_2[0:50], color = "purple")
plt.plot(gamma_array[0:50], fidelity_array[0:50], color = "blue")
plt.title(r"State Fidelity of Actual $\pi$ Rotation as a function of ORE paramet
plt.xlabel(r"$\gamma$")
plt.ylabel("Fidelity")

legend_elements = [Line2D([0], [0], color = 'purple', label = 'CORPSE'),
                   Line2D([0], [0], color = 'blue', label = 'elementary')]
plt.legend(handles = legend_elements)
```

Out[196… <matplotlib.legend.Legend at 0x12f38e190>

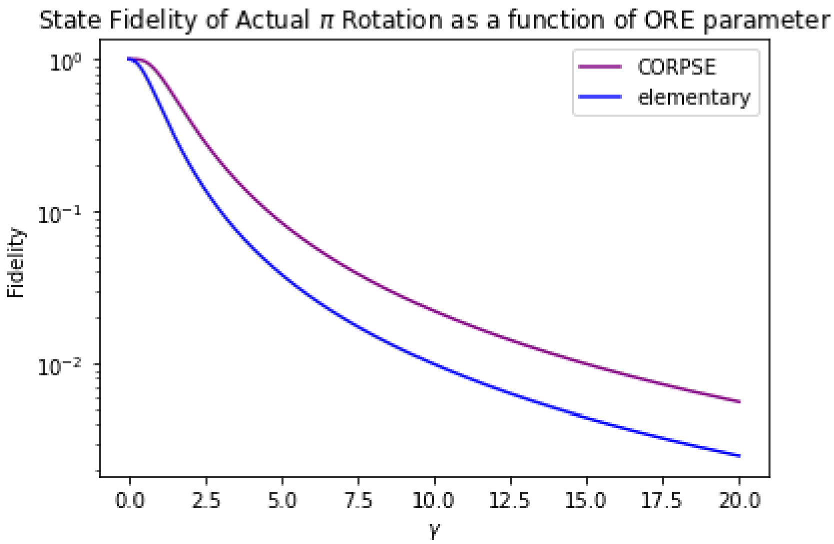


Alternatively, we can plot the fidelity on a log scale.

In [197…

```
# compare CORPSE with previous case, but now with log scale
plt.plot(gamma_array_2, fidelity_array_2, color = "purple")
plt.plot(gamma_array, fidelity_array, color = "blue")
plt.title(r"State Fidelity of Actual $\pi$ Rotation as a function of ORE paramet
plt.xlabel(r"$\gamma$")
plt.ylabel("Fidelity")
plt.yscale("log")

legend_elements = [Line2D([0], [0], color = 'purple', label = 'CORPSE'),
                   Line2D([0], [0], color = 'blue', label = 'elementary')]
plt.legend(handles = legend_elements)
```

Out[197… <matplotlib.legend.Legend at 0x130382e50>



Takeaways

This is incredible! Under the assumptions and definitions above, we see that the CORPSE pulse mitigates the effects of ORE across all values of γ, and is especially robust when the ORE is small (i.e. in the regime where $\gamma \approx 0$).

5. An Even Deeper Dive

For the remainder of this section, we will redefine the parameter of ORE as $\gamma = \frac{\Delta}{\Omega}$ and call it the **detuning ratio**.

- Δ: detuning
- Ω: Rabi frequency of qubit

Via the derivation from "Notes on Quantum Information", an imperfect elementary rotation operator is represented by the matrix:

$$U_\theta = \cos\frac{\theta}{2}\sqrt{1+\gamma^2}I - i\sin\frac{\theta}{2}\sqrt{1+\gamma^2}\bar{\sigma}$$

- $\bar{\sigma}$: $\frac{\gamma}{\sqrt{1+\gamma^2}}\sigma_z + \frac{1}{\sqrt{1+\gamma^2}}\sigma_x$

Play around with the widget below and see what happens.

In [198…

```
## choose gamma
gamma3_slider = widgets.FloatSlider(
    value = 0.1,
    min = 0,
    max = 1,
    step = 0.1,
    description = r'$\gamma$=',
    continuous_update = True
)
```

```
widgets.VBox([gamma3_slider],
             layout = widgets.Layout(align_items = 'center'))
```

In [199…

```
# create operator
gamma = gamma3_slider.value
barsigma = gamma / np.sqrt(1 + gamma ** 2) * Z + \
           1 / np.sqrt(1 + gamma ** 2) * X

actual = np.cos((theta/2) * np.sqrt(1 + gamma **2)) * I - \
         (1j) * np.sin((theta/2) * np.sqrt(1 +gamma **2)) * barsigma
actual_op = Operator(actual)
```

We can run a sanity check on our derivation by verifying that our operator is indeed unitary, and hence a valid rotation in Hilbert space.

In [200…

```
print("actual_op is unitary? " + str(actual_op.is_unitary()) + '.')
```

```
actual_op is unitary? True.
```

In [201…

```
# create circuit
cr = ClassicalRegister(1, name = "cr")
qr = QuantumRegister(1, name = "qr")
qc = QuantumCircuit(cr, qr)

qc.unitary(actual_op, 0, label = "Actually Realized Rotation")
qc.draw()
```

Out[201…

```
      ┌────────────────────────────┐
qr_0: ┤ Actually Realized Rotation ├
      └────────────────────────────┘
cr: 1/══════════════════════════════
```

In [202…

```
# execute circuit
statevec_sim = Aer.get_backend('statevector_simulator')
job = execute(experiments = qc, backend = statevec_sim)
result_vector = job.result().get_statevector()
result_vector_adjustedphase = np.array(result_vector)*1j # times i to cancel out
result_vector_adjustedphase
```

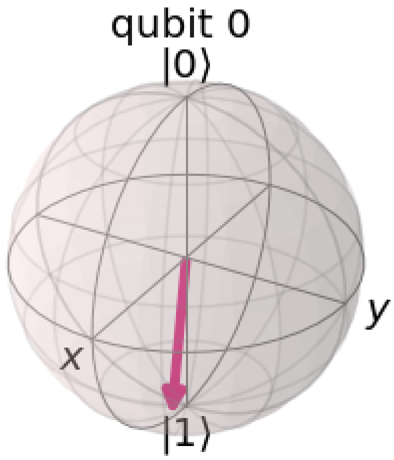
Out[202…

```
array([0.09950067-0.00783436j, 0.99500665+0.j        ])
```

In [203…

```
# plot result
actualrot_fig = plot_bloch_multivector(result_vector_adjustedphase)
actualrot_fig.suptitle(r'Actually Realized Rotation for $\gamma$ =' + f'{gamma}'
actualrot_fig
```

Out[203…



Actually Realized Rotation for γ =0.1

Again, we plot the fidelity over the full representative range of ORE and see what happens.

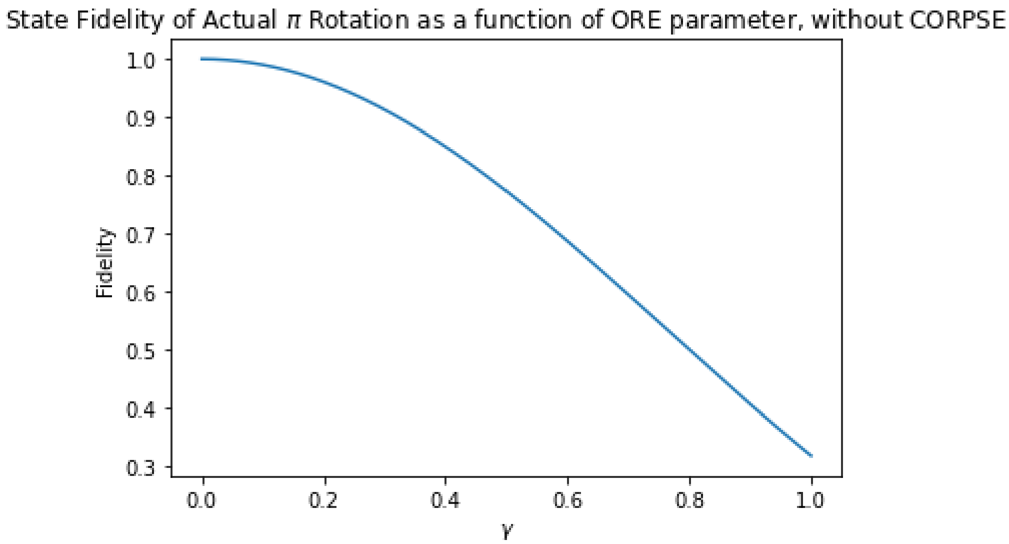
In [204…

```
gamma_array_3 = np.linspace(start = 0, stop = 1, num = 500)
fidelity_array_3 = []
for gamma in gamma_array_3:
    # create actual rot op
    barsigma = gamma / np.sqrt(1 + gamma ** 2) * Z + \
               1 / np.sqrt(1 + gamma ** 2) * X

    actual = np.cos((theta/2) * np.sqrt(1 + gamma **2)) * I - \
             (1j) * np.sin((theta/2) * np.sqrt(1 + gamma **2)) * barsigma
    actual_op = Operator(actual)
    # create qc
    cr = ClassicalRegister(1, name = "cr")
    qr = QuantumRegister(1, name = "qr")
    qc = QuantumCircuit(cr, qr)
    qc.unitary(actual_op, 0, label = "Actually Realized Rotation")
    # run qc
    statevec_sim = Aer.get_backend('statevector_simulator')
    job = execute(experiments = qc, backend = statevec_sim)
    # get resulting statevector
    result_vector = job.result().get_statevector()
    result_vector_adjustedphase = np.array(result_vector)*1j # times i to cancel
    # calculate fidelity
    fidelity_value = state_fidelity(result_vector_adjustedphase, [0 , 1])
    fidelity_array_3.append(fidelity_value)
```

In [205…

```
# plot results
plt.plot(gamma_array_3, fidelity_array_3)
plt.title(r"State Fidelity of Actual $\pi$ Rotation as a function of ORE paramet
plt.xlabel(r"$\gamma$")
plt.ylabel("Fidelity")
```

Out[205… Text(0, 0.5, 'Fidelity')



What about CORPSE?

In [206…

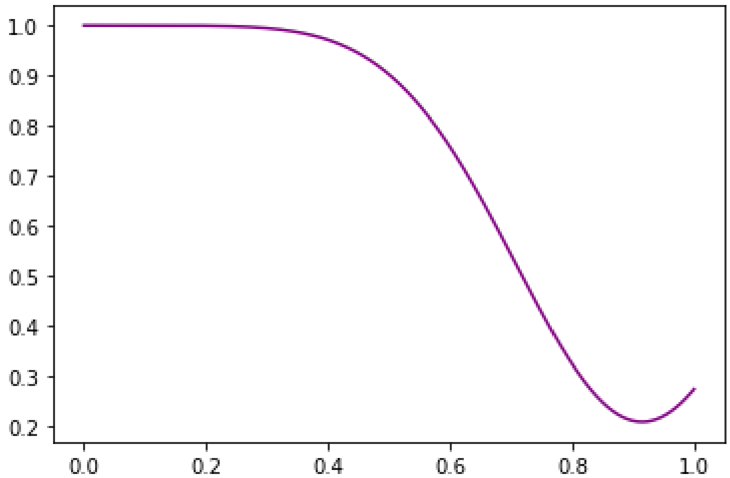
```
gamma_array_4 = np.linspace(start = 0, stop = 1, num = 500)
fidelity_array_4 = []
for gamma in gamma_array_4:
    # create actual rot ops
    CORPSE1_theta = 2*np.pi + theta/2 - np.arcsin(np.sin(theta/2)/2)
    CORPSE2_theta = 2*np.pi - 2* np.arcsin(np.sin(theta/2)/2)
    CORPSE3_theta = theta/2 - np.arcsin(np.sin(theta/2)/2)

    barsigma = gamma / np.sqrt(1 + gamma ** 2) * Z + \
               1 / np.sqrt(1 + gamma ** 2) * X

    barsigma_minusx = gamma / np.sqrt(1 + gamma ** 2) * Z + \
               1 / np.sqrt(1 + gamma ** 2) * (-X)

    CORPSE1 = np.cos((CORPSE1_theta/2) * np.sqrt(1 + gamma **2)) * I - \
             (1j) * np.sin((CORPSE1_theta/2) * np.sqrt(1 + gamma **2)) * barsigm

    CORPSE2 = np.cos((CORPSE2_theta/2) * np.sqrt(1 + gamma **2)) * I - \
             (1j) * np.sin((CORPSE2_theta/2) * np.sqrt(1 + gamma **2)) * barsigm

    CORPSE3 = np.cos((CORPSE3_theta/2) * np.sqrt(1 + gamma **2)) * I - \
             (1j) * np.sin((CORPSE3_theta/2) * np.sqrt(1 + gamma **2)) * barsigm

    CORPSE1_op = Operator(CORPSE1)
    CORPSE2_op = Operator(CORPSE2)
    CORPSE3_op = Operator(CORPSE3)
    # create qc
    cr = ClassicalRegister(1, name = "cr")
    qr = QuantumRegister(1, name = "qr")
    qc = QuantumCircuit(cr, qr)
    qc.unitary(CORPSE1_op, 0, label = "CORPSE rotation 1")
    qc.unitary(CORPSE2_op, 0, label = "CORPSE rotation 2")
    qc.unitary(CORPSE3_op, 0, label = "CORPSE rotation 3")
    # run qc
    statevec_sim = Aer.get_backend('statevector_simulator')
    job = execute(experiments = qc, backend = statevec_sim)
    # get resulting statevector
    result_vector = job.result().get_statevector()
    result_vector_adjustedphase = np.array(result_vector)*1j # times i to cancel
    # calculate fidelity
```

```
    fidelity_value = state_fidelity(result_vector_adjustedphase, [0 , 1])
    fidelity_array_4.append(fidelity_value)
```

In [207…

```
plt.plot(gamma_array_4, fidelity_array_4, color  = "purple")
```

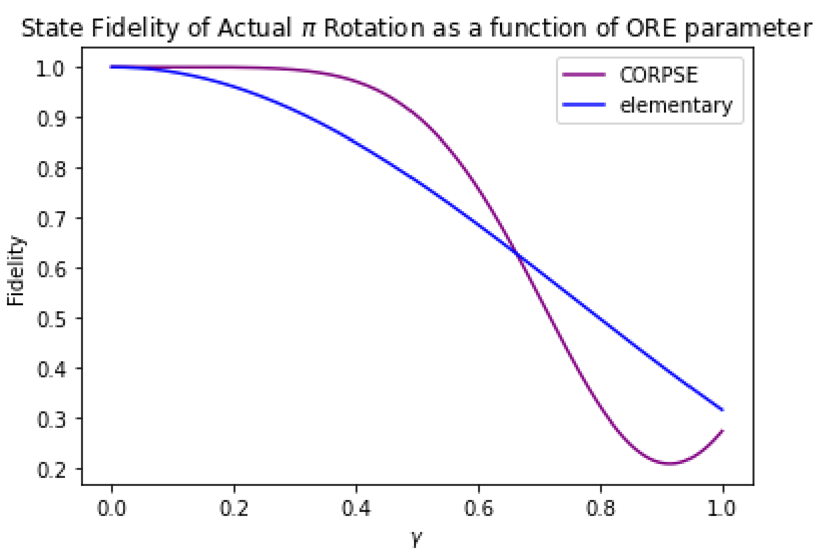
Out[207… [<matplotlib.lines.Line2D at 0x1307f52e0>]



In [208…

```
# compare CORPSE with previous case
plt.plot(gamma_array_4, fidelity_array_4, color = "purple")
plt.plot(gamma_array_3, fidelity_array_3, color = "blue")
plt.title(r"State Fidelity of Actual $\pi$ Rotation as a function of ORE paramet
plt.xlabel(r"$\gamma$")
plt.ylabel("Fidelity")

legend_elements = [Line2D([0], [0], color = 'purple', label = 'CORPSE'),
                   Line2D([0], [0], color = 'blue', label = 'elementary')]
plt.legend(handles = legend_elements)
```

Out[208… <matplotlib.legend.Legend at 0x13084ee50>



Conclusion

We see that as long as γ is less than roughly 0.65, using a CORPSE pulse will allow as to mitigate the effects of an ORE! Moreover, for γ less than roughly 0.3, this effect is very substantial! This graph very close to the results of reference [1], where they use a slightly different variable to parameterize the ORE.

An actual quantum computer may employ CORPSE to substantially reduce the effects of small off-resonant errors, which may precede the usage of quantum-error correcting protocols. Furthermore, CORPSE is the simplest type of composite rotation with only three rotations and thus creates minimal circuit depth, and thus poses minimal overhead on the quantum hardware.

Next steps for this project

- Develop notebook to include arbitrary azimuthal angle ϕ, instead of just along x-axis
- Develop interactive calculator that customizes any CORPSE pulse pased on user input, and simulates result in circuit
- Improve notebook formatting

In [ ]:

```

```