Research Update and Goals 25F

September 15, 2025

Aryan Bawa
Dartmouth College
Department of Physics and Astronomy

1. Research Update

1.1. Relaxation Method QSP Approximation

My main achievement over the last term of research was a working model for QSP approximation using a relaxation method approach outlined in this Pennylane notebook. I retained the PyTorch gradient descent approach used in the notebook while changing other aspects to approximate a whole gate rather than just a polynomial. The most significant change was refitting the model by using fidelity as a distance metric. As the model learned to minimize fidelity, the output phi values are those that yield a matrix closest to the input unitary for a set a value. When approximating a Hadamard gate, the model achieved a fidelity of 0.99991608 with $a \approx -\frac{\pi}{5}$ and $d = 5$.

1.2. QSP/QSVT Theory Document

To cement my understanding of the theory behind both QSP and QSVT, I started a document outlining the theory behind each process in a succinct, digestible way. This proved useful for developing my QSP approximation model, as an understanding of the theory allowed me to fix issues in my code that resultied in an inaccurate initial fidelity. I believe this document will also prove useful for future research presentations/write-ups. I have currently completed the QSP section of the document and have started the QSVT section, although I expect both sections to expand as I learn more and continue updating my models.

1.3. Cline and Dartmouth AI API Integration

Over the brief summer break, I began looking into integrating cline.ai with the Dartmouth AI API. I have made decent progress in understanding the approach and limitations. The way I have approached the problem so far is to use cline's MCP integration capabilities. I was able to use the Dartmouth AI langchain module cookbook to start creating the MCP. Currently, it requires further testing and debugging, as I have not been able to get it to work fully yet. However, I believe I am on the right track and will continue working on this project over the fall term.

2. Fall Term Goals

2.1. Research Progress

In terms of progress I would like to make during fall term, the main focus will be having a working QSVT approximation. Of course, this means that my current QSP approximation code will have to be tweaked until it is sufficiently efficent. To accomplish this, I plan to confer with other group members who have worked more with QML and ML in general. I will explain my current approach and see if there are any improvements I can make. After thoroughly tweaking with my current model, I will move onto QSVT approximation. Thorughout this process, I will continue to update my theory document with important information about QSP, QSVT, and my approximation models. I may continue work on the Cline project occasionally thorughout the term, but my main focus will be on the approximation.

2.2. Communication and Organization

My biggest shortcomings over the last few terms have been a lack of communication and organization. I believe both of these problems can be solved primarily through pre-scheduled weekly meetings with Professor Whitfield and Linta. These will provide an environment for me to express what difficulties I may be having, as this can be more difficult during lab meeting. In this meetings, I will also have a discernable goal week-to-week so that my progress is measurable between meetings. Additionally, I will do a majority of my work on the project in the actual lab room (if there is space), so I can quickly access help if I need it and be held accountable for my work.

3. Appendix

3.1. QSP Approximation Code

```
# have to do this because some libraries using OpenMP cause issues with multiple threads
import os
os.environ['KMP_DUPLICATE_LIB_OK'] = 'TRUE'

import torch
import pennylane as qml
import math
import numpy as np
import matplotlib.pyplot as plt

# defining important functions
def rotation_mat(a):
    """Given a fixed value 'a', compute the signal rotation matrix W(a).
    (requires -1 <= 'a' <= 1)
    """
    diag = a
    off_diag = (1 - a**2) ** (1 / 2) * 1j
    W = [[diag, off_diag], [off_diag, diag]]

    return W

def generate_many_sro(a_vals):
    """Given a tensor of possible 'a' vals, return a tensor of W(a)"""
    w_array = []
    for a in a_vals:
        w = rotation_mat(a)
        w_array.append(w)

    return torch.tensor(w_array, dtype=torch.complex64, requires_grad=False)

def target_hadamard():
    """Return the target Hadamard matrix"""
    return torch.tensor([[1/np.sqrt(2), 1/np.sqrt(2)],
                         [1/np.sqrt(2), -1/np.sqrt(2)]],
                        dtype=torch.complex64)

# fidelity-based loss function for the QSP unitary fitting
def fidelity_loss(pred_unitaries, target_op):
    batch_size = pred_unitaries.shape[0]
    fidelities = []

    for i in range(batch_size):
        U_pred = pred_unitaries[i]
        U_target = target_op

        product = torch.matmul(torch.conj(U_pred).T, U_target)
        trace = torch.trace(product)
        fidelity = torch.abs(trace)**2 / 4.0
        fidelities.append(fidelity)

    avg_fidelity = torch.mean(torch.stack(fidelities))
    return 1.0 - avg_fidelity
```

```
# below is the QSP circuit itself - we don't change the basis this time since we don't
need to use the relaxed third condition for QSP
@qml.qnode(qml.device("default.qubit", wires=1))
def QSP_circ(phi, W):
    for angle in phi[:-1]:
        qml.RZ(angle, wires=0)
        qml.QubitUnitary(W, wires=0)
    qml.RZ(phi[-1], wires=0)
    return qml.state()

    torch_pi = torch.Tensor([math.pi])

class QSP_Unitary_Fit(torch.nn.Module):
    def __init__(self, degree, random_seed=None):
        """Given the degree and number of samples, this method randomly
        initializes the parameter vector (randomness can be set by random_seed)
        """
        super().__init__()
        if random_seed is None:
            self.phi = torch_pi * torch.rand(degree + 1, requires_grad=True)

        else:
            gen = torch.Generator()
            gen.manual_seed(random_seed)
            self.phi = torch_pi * torch.rand(degree + 1, requires_grad=True,
generator=gen)

        self.phi = torch.nn.Parameter(self.phi)
        self.num_phi = degree + 1

    def forward(self, omega_mats):
        """PennyLane forward implementation"""
        unitary_pred = []
        generate_qsp_mat = qml.matrix(QSP_circ, wire_order=[0])

        for w in omega_mats:
            u_qsp = generate_qsp_mat(self.phi, w)
            unitary_pred.append(u_qsp)

        return torch.stack(unitary_pred, 0)

        # cleaned this up and added verbose output with Claude AI's help
class Model_Runner:
    def __init__(self, degree, num_samples):
        self.degree = degree
        self.num_samples = num_samples
        # generating a values and corresponding W(a) matrices
        self.a_vals = torch.linspace(-0.99, 0.99, num_samples)
        self.omega_mats = generate_many_sro(self.a_vals)
        self.target_op = target_hadamard()

        print(f"Initialized model runner")
```

```python
    def execute(self, max_iterations=5000, lr=1e-3, random_seed=42, verbose=True):
        # starting up the model
        self.model = QSP_Unitary_Fit(self.degree, random_seed=random_seed)
        optimizer = torch.optim.Adam(self.model.parameters(), lr=lr)

        # storing the initial for comparison
        with torch.no_grad():
            self.init_pred = self.model(self.omega_mats)

        # Training tracking
        self.losses = []
        self.fidelities = []

        if verbose:
            print(f"\nTraining QSP to approximate Hadamard gate...")
            print(f"Learning rate: {lr}")
            print("-" * 50)

        for t in range(max_iterations):
            pred_unitaries = self.model(self.omega_mats)
            loss = fidelity_loss(pred_unitaries, self.target_op)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            current_loss = loss.item()
            current_fidelity = 1.0 - current_loss

            self.losses.append(current_loss)
            self.fidelities.append(current_fidelity)

            if verbose and (t % 500 == 0 or t < 10):
                print(f"Iteration {t:5d}: Loss = {current_loss:.8f}, Avg Fidelity =
{current_fidelity:.8f}")

            # Early stopping for very high fidelity
            if current_fidelity > 0.9999:
                if verbose:
                    print(f"High fidelity achieved! Stopping at iteration {t}")
                break

            # Early stopping for convergence
            if current_loss < 1e-8:
                if verbose:
                    print(f"Converged at iteration {t}")
                break

        # Store final results
        self.final_loss = current_loss
        self.final_fidelity = current_fidelity
        self.training_iterations = len(self.losses)

        if verbose:
            print(f"\nTraining completed after {self.training_iterations} iterations")
            print(f"Final loss: {self.final_loss:.8f}")
            print(f"Final average fidelity: {self.final_fidelity:.8f}")

    def evaluate(self, verbose=True):
        """
        Evaluate the quality of the approximation
        """
        if not hasattr(self, 'model'):
            raise ValueError("Must run execute() before evaluate()")

        with torch.no_grad():
            self.final_pred = self.model(self.omega_mats)

        # Compute fidelities and errors for each 'a' value
        self.fidelities_per_a = []
        self.errors_per_a = []

        for i, a in enumerate(self.a_vals):
            U_pred = self.final_pred[i]

            # Use the SAME fidelity calculation as in fidelity_loss function
            product = torch.matmul(torch.conj(U_pred).T, self.target_op)
            trace = torch.trace(product)
            fidelity = (torch.abs(trace)**2 / 4.0).item()
            self.fidelities_per_a.append(fidelity)

            # Frobenius error for additional analysis
            diff = U_pred - self.target_op
            frobenius_error = torch.sqrt(torch.sum(torch.abs(diff)**2)).item()
            self.errors_per_a.append(frobenius_error)

        # Compute statistics
        self.avg_fidelity = np.mean(self.fidelities_per_a)
        self.min_fidelity = np.min(self.fidelities_per_a)
        self.max_fidelity = np.max(self.fidelities_per_a)
        self.std_fidelity = np.std(self.fidelities_per_a)


        if verbose:
            print(f"\n=== Hadamard Gate Approximation Quality ===")
            print(f"Average Fidelity:     {self.avg_fidelity:.8f}")
            print(f"Minimum Fidelity:     {self.min_fidelity:.8f}")
            print(f"Maximum Fidelity:     {self.max_fidelity:.8f}")
            print(f"Fidelity Std Dev:     {self.std_fidelity:.8f}")


    def plot_results(self, show=True, figsize=(15, 5)):
        """
        Plot comprehensive training and evaluation results
        """
        if not hasattr(self, 'losses'):
            raise ValueError("Must run execute() before plotting")

        if not hasattr(self, 'fidelities_per_a'):
            self.evaluate(verbose=False)
        plt.figure(figsize=figsize)

        # Plot 1: Training progress
        plt.subplot(1, 3, 1)
        plt.plot(self.losses, label='Loss', color='red', alpha=0.7)
        plt.plot(self.fidelities, label='Avg Fidelity', color='blue', alpha=0.7)
        plt.xlabel('Iteration')
        plt.ylabel('Value')
        plt.title('Training Progress')
        plt.legend()
        plt.yscale('log')
        plt.grid(True)

        if show:
            plt.show()

    def get_learned_parameters(self):
        """
        Get the learned φ parameters
        """
        if not hasattr(self, 'model'):
            raise ValueError("Must run execute() before getting parameters")
        return self.model.phi.detach().numpy()

        # Initialize the runner
runner = Model_Runner(degree=5, num_samples=10)

# Execute training
runner.execute(max_iterations=1000, lr=1e-2, random_seed=42)

# Evaluate the results
runner.evaluate()

# Plot comprehensive results
runner.plot_results()

# Display learned parameters and additional analysis
print(f"\n=== Final Results ===")
learned_params = runner.get_learned_parameters()
print(f"Learned φ parameters: {learned_params}")
print(f"Parameter range: [{learned_params.min():.4f}, {learned_params.max():.4f}]")
print(f"Training iterations: {runner.training_iterations}")

# converting learned values for a
phi_learned = torch.tensor(learned_params, dtype=torch.float32)

test_a = -0.64314 #  any value in [-0.99, 0.99] - initially was 0.98 but manually fit it
to this value (about -pi/5)
test_W = torch.tensor(rotation_mat(test_a), dtype=torch.complex64)


qsp_matrix_func = qml.matrix(QSP_circ, wire_order=[0])
learned_unitary = qsp_matrix_func(phi_learned, test_W)
```

```python
print(f"\n=== QSP Output Unitary (a={test_a}) ===")
print("Learned QSP Unitary:")
print(learned_unitary)
print("\nTarget Hadamard:")
print(target_hadamard())

# fidelity for this phi
product = torch.matmul(torch.conj(learned_unitary).T, target_hadamard())
trace = torch.trace(product)
fidelity = (torch.abs(trace)**2 / 4.0).item()
print(f"\nFidelity with Hadamard: {fidelity:.8f}")
```

3.1.1. QSP Approximation Output

```
=== Final Results ===
Learned φ parameters: [3.565692  3.241349  2.070773  2.8037655 2.2006905 1.8083417]
Parameter range: [1.8083, 3.5657]
Training iterations: 1000

=== QSP Output Unitary (a=-0.64314) ===
Learned QSP Unitary:
tensor([[ 0.0054+0.7073j, -0.0074+0.7069j],
        [ 0.0074+0.7069j,  0.0054-0.7073j]])

Target Hadamard:
tensor([[ 0.7071+0.j,  0.7071+0.j],
        [ 0.7071+0.j, -0.7071+0.j]])

Fidelity with Hadamard: 0.99991608
```

3.2. QSP/QSVT Theory Document

Begins on following page.

Dequantization with Quantum Signal Processing and Quantum Singular Value Transform

August 13, 2025

Aryan Bawa
Dartmouth College
Department of Physics and Astronomy

1. Quantum Signal Processing

Quantum signal processing (QSP) is derived from an attempt to characterize pulse sequences used in NMR. It is built upon the idea of interleaving two single-qubit rotations—a signal rotation operator W and a signal processing rotation operator S. The rotation always rotates through the same angle θ, while the processing operator rotates through a variable angle depending on a predetermined sequence. Mathematically, the QSP operator is defined as below for a vector of phases $\vec{\varphi} = (\varphi_0, \varphi_1, ...\varphi_d) \in \mathbb{R}^{d+1}$:

$$U_{\text{QSP}}(\vec{\varphi}) = S(\varphi_0) \prod_{k=1}^{d} S(\varphi_k) W(a) \tag{1.1}$$

Here, d is a free parameter which represents the number of repeated applications of $W(a)$. In general, the matrix element of the QSP unitary $\langle 0 | U_{\text{QSP}(\vec{\varphi})} | 0 \rangle = P(a)$ is polynomial in a with degree d. The inverse is also true: given a polynomial $P(a)$ under certian constraints, there exists a set of phases $\vec{\varphi}$ such that $P(a) = \langle 0 | U_{\text{QSP}(\vec{\varphi})} | 0 \rangle$. This is the core principle behind QSP.

As an example, using the W_x convention of QSP, we can express the signal rotation and signal processing rotation operators as:

$$W(a) = \begin{pmatrix} a & i\sqrt{1-a^2} \\ i\sqrt{1-a^2} & a \end{pmatrix} \tag{1.2}$$

$$S(\varphi) = e^{-i\varphi Z} \tag{1.3}$$

where Z is the Pauli-Z operator and W(a) is a rotation by angle $\theta = -2\arccos(a)$ around the x-axis.

1.1. Qubitization

1.2. Dequantizing QSP

2. Quantum Singular Value Transform

2.1. Dequantizing QSVT