Quantum Computing Applications in Nuclear Physics

Jingyi Li, Alexandra Semposki, Pablo Giuliani, Kyle Godbey

Jun 30, 2022

CONTENTS

This book of background, tutorials, and applications of aspects of quantum computing was born out of the 2022 FRIB quantum computing summer school and attempts to explore tools and techniques of interest to nuclear physicists.

Through this virtual book, we will go through three main topic :

1. The Variational Quantum Eigensolver, a technique that uses a hybrid approach between a classical and a quantum computer to calculate the ground state configuration of a system. We showcase the technique by applying it to the computation of the ground state of the deuteron.
2. The noise challenges that arise from running a non-idealized quantum circuit, including techniques for mitigation such as the Zero Noise Extrapolation. We will give an overview on the origins and structure of quantum noise, as well as present how it can be applied to the deuteron problem.
3. Dimensionality reduction techniques, in particular the Reduced Basis Method. These techniques could be crucial to reduce the (usually houndreds, thousands, or milliomns) effective degrees of freedom of a system to a point where it becomes accessible to modern few-qbits computers. In particular, we show how the Quantum Harmonic Oscillator in coordinate representation can be reduced to a two body system of interacting "particles" in the configuration space constructed by the reduced basis.

The tools we develop and showcase in these three topics will come to play together to tackle a challenging problem in the final part of this book: using a quantum computer we will solve the one dimensional Gross-Pitaevskii equation. This non-linear Schrödinger equation approximately describes the low-energy properties of dilute Bose-Einstein condensates, and is one of the simplest models describing many-body quantum systems. We will use the Variational Quantum Eigensolver to find the ground state of the reduced system for different Hamiltonian parameters, while the Zero Noise Extrapolation will be exploited to produce accurate results even in the presence of noise.

List of contributors can be found *here*.

CHAPTER

ONE

INTRODUCTION

Author: Alexandra Semposki

Many of the most intriguing problems known by the physics community are presently unreachable by classical computation. Diagonalization of high-dimensional matrices cannot be performed beyond a dimension of 10^{11} currently (see Tsunoda, N., Otsuka, T., Takayanagi, K. et al. The impact of nuclear shape on the emergence of the neutron dripline. Nature 587, 66–71 (2020), and the fermion sign problem pervades Quantum Monte Carlo (QMC) calculations. Instead of classical computers, we now look to quantum computers to provide aid on these frustrating limits of modern classical technology. The hope is that, in the near future—with the fast developments in the quantum computing (QC) world—we may be able to mitigate the such issues with quantum computers, reducing time to run computationally challenging physics problems from exponential time to polynomial time. With this improvement, we might get results -during our lifespam- on problems that would have, on a classical computer, taken longer than the age of the universe to complete!

Quantum computers run in a similar way to classical computers: they use a form of bit called a qubit, and have quantum gates that perform transformations on these qubits, much like classical gates do on classical bits. However, there are a couple of stark differences between the two systems. One, a qubit can take many more possible values than a classical bit owing to their ability to be in a superposition of states, and two, quantum gates are reversible—hence they are unitary operators, a condition not necessary for gates in a classical circuit. Of course, with qubits and quantum gates comes the question: what about entanglement? Entanglement can occur and be induced between qubits on purpose using the gates, but qubits can also interact with the environment, causing a fair amount of error to arise. Is this the end for qubits? Not to worry! Error correction and mitigation are also fast advancing, with algorithms to correct for bit and phase flips developing and improving year by year.

There are many techniques used in QC that are worth investigating, and many have been and will be tremendously useful in the fields of few-body and many-body systems in physics. One that has been used quite often is the Variational Quantum Eigensolver (VQE), which can determine the ground state of an atomic system. It is very popular in Quantum Chemistry and there have been many applications of it in recent years—for example, this PRL paper from 2018, where it is applied to the deuteron.

VQE will be discussed in detail in the next chapter, but right now we will say that it is an application of the Variational Principle you most likely learned in your first ever Quantum Mechanics class in undergrad, with more bells and whistles and qubits of course! The next two chapters deal with circuit noise and dimensionality reduction techniques, respectively. We tackle the challenge problem of the one dimensional Gross-Pitaevskii in the final part of this journey, and then close with some conclusions and remarks.

CHAPTER

TWO

VARIATIONAL QUANTUM EIGENSOLVER

Author: Alexandra Semposki

The variational quantum eigensolver (VQE) has been used prominently in QC in the past few years as a way to find the ground state energy of a system. This is done using the variational method from quantum mechanics, which is able to provide an upper bound for this ground state energy. We start with a Hermitian Hamiltonian, H, and write the eigenvalue equation down as

$$H|\psi_i\rangle = \lambda_i|\psi_i\rangle.$$

Say, however, that we cannot directly solve this and must instead use states $|\psi(\theta)\rangle$, which have some variable angle θ dependence, and can be expanded in a basis of the original states such that

$$|\psi(\theta)\rangle = \sum_i c_i|\psi_i\rangle.$$

Now we can find the expectation value.

$$\langle\psi(\theta)|H|\psi(\theta)\rangle = \sum_{i,j} c_j^* c_i \langle\psi_j|H|\psi_i\rangle = \sum_{i,j} \delta_{ij} c_j^* c_i E_i,$$

and finally

$$\langle\psi(\theta)|H|\psi(\theta)\rangle = \sum_i |c_i|^2 E_i,$$

and we know that

$$\langle\psi(\theta)|H|\psi(\theta)\rangle = E \geq E_0,$$

so we can minimize the angles θ and get

$$\min\langle\psi(\theta)|H|\psi(\theta)\rangle = E_0.$$

VQE will attempt to do this angle minimization to obtain a reasonable approximation to the ground state energy of the deuteron in this chapter. (To see the notebook this was adapted from, and an excellent tutorial using a very simple Hamiltonian, click here).

When it comes to implementing this technique, there are a few main ingredients in the recipe:

1. The Hamiltonian in question transformed into the Pauli basis using the **Jordan-Wigner transformation**;
2. A suitable **ansatz** for the wave function;
3. A classical **optimization routine** to be used to continously optimize the angles θ for each run of the VQE circuit.

In the next chapter, we will see these three ingredients worked out in detail and implemented using the package pennylane.

2.1 Quantum Computing the Deuteron

In this introduction we will briefly discuss the 2018 paper, Cloud Quantum Computing of an Atomic Nucleus. discuss the deuteron and the model maybe? It should be somewhere, maybe here instead of the notebook. Just need a light introduction before the real work.

From the paper we can read that the authors employed the projection of hamiltonian operators onto quantum qubits (X,Y,Z) to construct a quantum computatable hamiltonian by the Pionless EFT, a systematically improvable and model-independent approach to nuclear interactions in a regime where the momentum scale Q of the interesting physics. The important part of this paper is how the deuteron creation and annihilation operator is mapped onto qubits using Jordan-Wigner transformation. Qubits can be used by quantum computers for operations based on Pauli matrices (denoted as X_q, Y_q, and Z_q on qubit q).

2.1.1 Defining the Hamiltonian

The deuteron Hamiltonian is

$$H_N = \sum_{n,n'=0}^{N-1} \langle n'|(T+V)|n\rangle \, a^\dagger_{n'} a_n. \tag{2.1}$$

The operators $a^\dagger_n$ and a_n here are the creation and annihilation operators. A deuteron is created or annihilated in the harmonic-oscillator s-wave state $|n\rangle$. The matrix elements of the kinetic and potential energy are represented by:

$$\begin{aligned}\langle n'|T|n\rangle =& \frac{\hbar\omega}{2}\left[(2n+3/2)\delta^{n'}_n - \sqrt{n(n+1/2)}\delta^{n'+1}_n \right.\\ & \left. -\sqrt{(n+1)(n+3/2)}\delta^{n'-1}_n\right]\\ \langle n'|V|n\rangle =& V_0\delta^0_n\delta^{n'}_n\end{aligned} \tag{2.2}$$

Here, $V_0 = -5.68658111$MeV, and $n, n' = 0, 1, \ldots N-1$, for a basis of dimension N. We also set $\hbar\omega = 7$MeV.

2.1.2 Mapping the deuteron onto qubits

Quantum computers manipulate qubits by operations based on Pauli matrices (denoted as X_q, Y_q, and Z_q on qubit q). The deuteron creation and annihilation operators can be mapped onto Pauli matrices via the Jordan-Wigner transformation, defined as,

$$\begin{aligned}a^\dagger_n &\to \frac{1}{2}\left[\prod_{j=0}^{n-1} -Z_j\right](X_n - iY_n)\\ a_n &\to \frac{1}{2}\left[\prod_{j=0}^{n-1} -Z_j\right](X_n + iY_n)\,.\end{aligned} \tag{2.3}$$

Note that other mappings can be used, with the various mappings having benefits and drawbacks that may apply to your problem of interest.

A spin up $|\uparrow\rangle$ (down $|\downarrow\rangle$) on qubit n corresponds to zero (one) deuteron in the state $|n\rangle$. For $N = 2, 3$ we have the components of the Hamiltonian (all numbers are in units of MeV)

$$\begin{aligned}H_2 =& 5.906709I + 0.218291Z_0 - 6.125Z_1\\ & - 2.143304\,(X_0X_1 + Y_0Y_1)\end{aligned} \tag{2.4}$$

$$H_3 = H_2 + 9.625\,(I - Z_2) - 3.913119\,(X_1X_2 + Y_1Y_2) \tag{2.5}$$

2.1.3 First Steps: N=2

Author(s): Kyle Godbey

Maintainer: Kyle Godbey

With the background of the previous page, we now begin our quantum computing journey! This entry will walk you through how to set up our Hamiltonian for the deuteron defined previously as well as an appropriate ansatz for the problem.

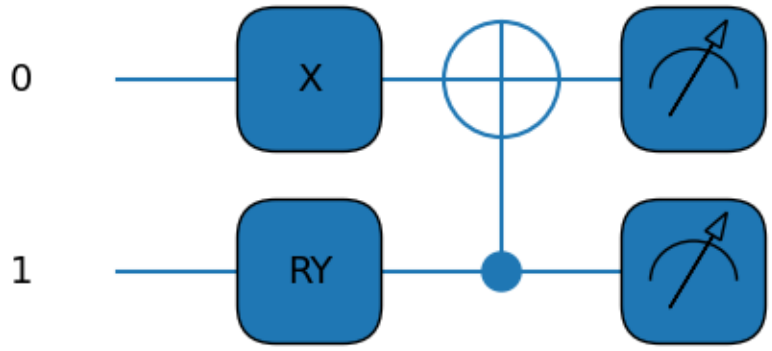
Now let's set up our imports! For this first example and the next we will just use pennylane and define the circuits and Hamiltonian directly in the form from Cloud Quantum Computing of an Atomic Nucleus.

```
%matplotlib inline

import matplotlib.pyplot as plt
from pennylane import numpy as np
import pennylane as qml
import warnings
warnings.filterwarnings('ignore')

plt.style.use(['science','notebook'])
```

Now we will define our 'device' (a simulator in this case) as well as our ansatz and Hamiltonian.

```
# In this case we just need 2 qubits

dev = qml.device("default.qubit", wires=2)

# Defining our ansatz circuit for the N=2 case

def circuit(params,wires):
    t0 = params[0]
    qml.PauliX(wires=0)
    qml.RY(t0, wires=1)
    qml.CNOT(wires=[1,0])

# And building our Hamiltonian for the N=2 case

coeffs = [5.906709, 0.218291, -6.125, -2.143304, -2.143304]
obs = [qml.Identity(0), qml.PauliZ(0), qml.PauliZ(1), qml.PauliX(0) @ qml.PauliX(1),␣
 ↪qml.PauliY(0) @ qml.PauliY(1)]

H = qml.Hamiltonian(coeffs, obs)
cost_fn = qml.ExpvalCost(circuit, H, dev)

# Let's print it out

print(H)

@qml.qnode(dev)
def draw_circuit(params):
    circuit(params,wires=dev.wires)
    return qml.expval(H)

qml.drawer.use_style('default')
print(qml.draw_mpl(draw_circuit)(init_params))
```

```
  (-6.125) [Z1]
+ (0.218291) [Z0]
+ (5.906709) [I0]
+ (-2.143304) [X0 X1]
+ (-2.143304) [Y0 Y1]
(<Figure size 500x300 with 1 Axes>, <Axes:>)
```



Great! That's a good looking Hamiltonian, even if it was a bit tedious to write it out. We'll find out a more programmatic way to do it later, but for now let's move on to the fun stuff.

Now we'll set up what we'll need for the VQE procedure; namely some initial parameters and convergence info. You can select the initial guess randomly, but in this case we'll set it manually for repeatability.

```
# Our parameter array, only one lonely element in this case :c

init_params = np.array([2.5,])

# Convergence information and step size

max_iterations = 500
conv_tol = 1e-06
step_size = 0.01
```

Finally, the VQE block. A lot of the stuff in this block is bookkeeping – the real magic happens in the `opt` object that contains an built-in optimizer. You can have a look at the documentation and change this if you like, but we'll also have a look at this later.

```
opt = qml.GradientDescentOptimizer(stepsize=step_size)

params = init_params

gd_param_history = [params]
gd_cost_history = []

for n in range(max_iterations):
```

(continues on next page)

(continued from previous page)

```
    # Take a step in parameter space and record your energy
    params, prev_energy = opt.step_and_cost(cost_fn, params)

    # This keeps track of our energy for plotting at comparisons
    gd_param_history.append(params)
    gd_cost_history.append(prev_energy)

    # Here we see what the energy of our system is with the new parameters
    energy = cost_fn(params)

    # Calculate difference between new and old energies
    conv = np.abs(energy - prev_energy)

    if n % 20 == 0:
        print(
            "Iteration = {:},  Energy = {:.8f} MeV,  Convergence parameter = {"
            ":.8f} MeV".format(n, energy, conv)
        )

    if conv <= conv_tol:
        break

print()
print("Final value of the energy = {:.8f} MeV".format(energy))
print("Number of iterations = ", n)
```

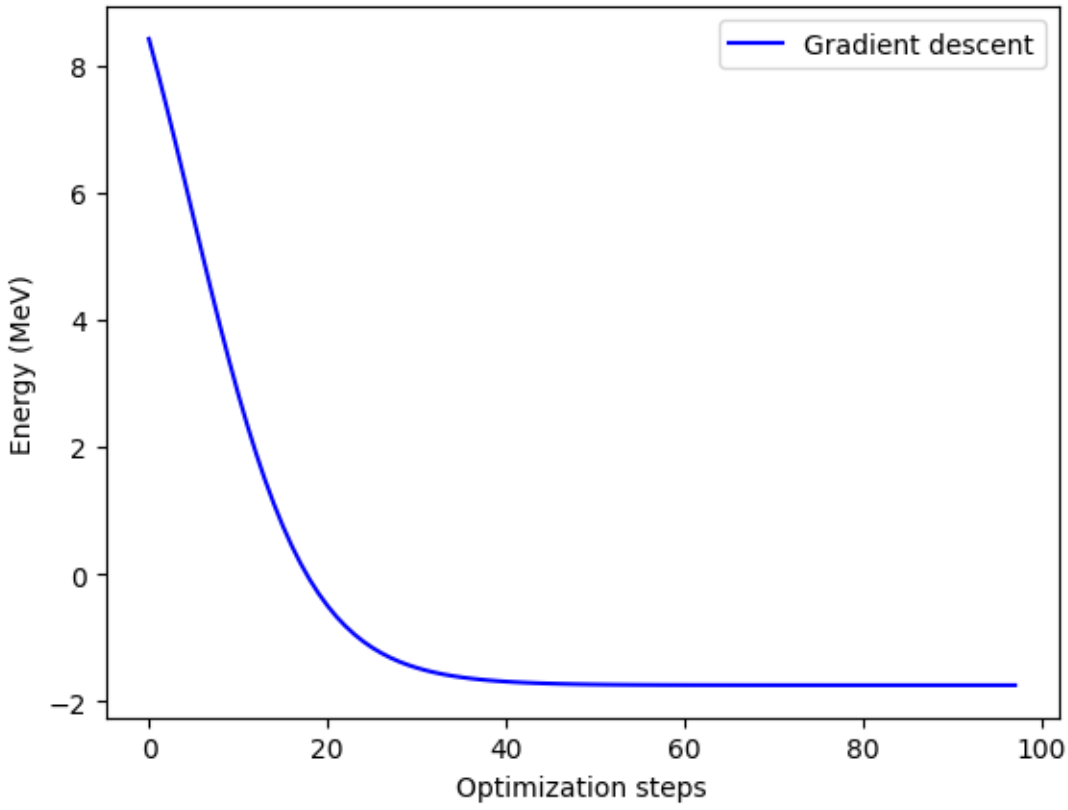
```
Iteration = 0,  Energy = 7.89426366 MeV,  Convergence parameter = 0.52891694 MeV
Iteration = 20,  Energy = -0.66950925 MeV,  Convergence parameter = 0.17005461 MeV
Iteration = 40,  Energy = -1.70101182 MeV,  Convergence parameter = 0.00828148 MeV
Iteration = 60,  Energy = -1.74716419 MeV,  Convergence parameter = 0.00034480 MeV
Iteration = 80,  Energy = -1.74907865 MeV,  Convergence parameter = 0.00001426 MeV

Final value of the energy = -1.74915572 MeV
Number of iterations =  97
```

Not too bad! Well, it's a far cry from the true, real-life binding energy of the deuteron, but it's a start! You can have a look at the paper linked above and find some tricks to extrapolate to the true value.

But just looking at the VQE result, if you compare this to the value reported in the paper you'll see that we match their value very well, which is encouraging.

Let's plot the convergence next.

```
plt.plot(gd_cost_history, "b", label="Gradient descent")

plt.ylabel("Energy (MeV)")
plt.xlabel("Optimization steps")
plt.legend()
plt.show()
```



Finally, we can plot how we trace the potential energy surface (PES) as we find a solution. For this 1-D case it's not as interesting, but it might hint at how you can improve performance.

If you're running this yourself, you can either generate the surface yourself or put this code in a cell to download it locally:

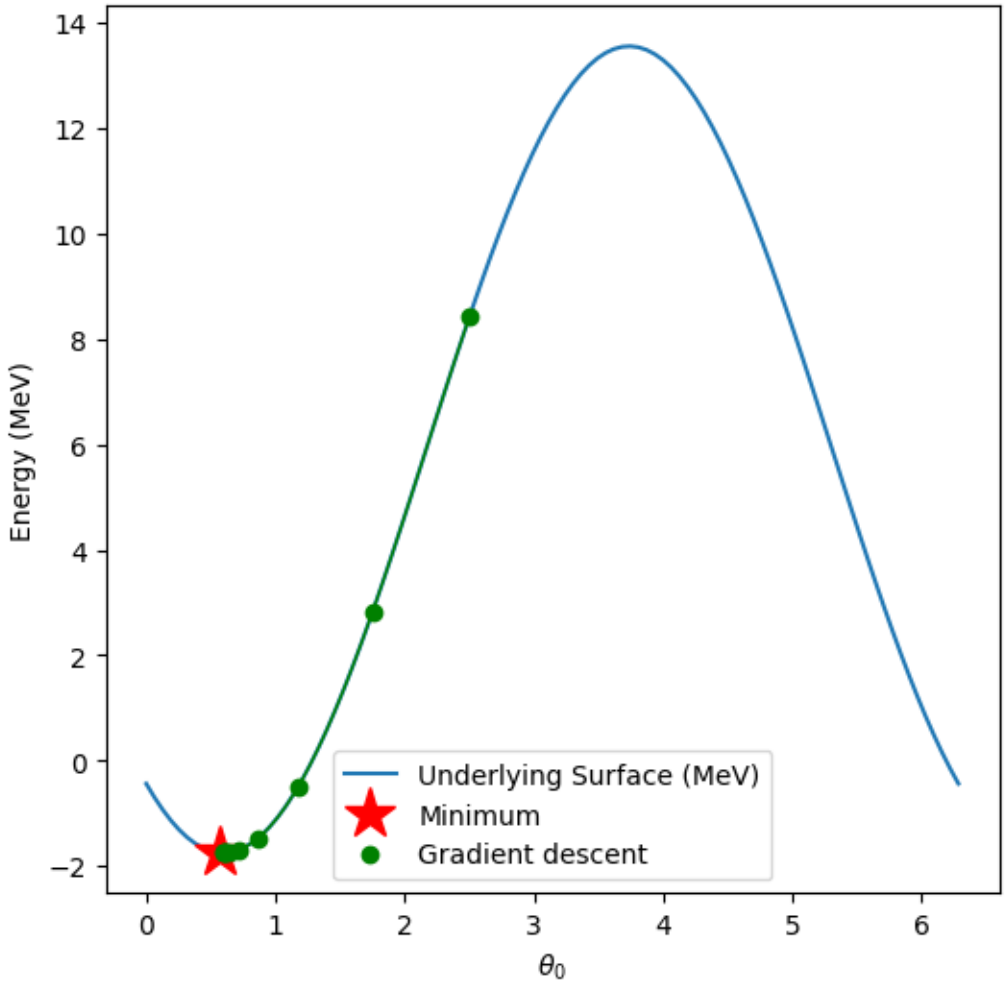
```
!wget https://github.com/kylegodbey/nuclear-quantum-computing/raw/main/
nuclear-qc/vqe/deut_pes_n2.npy
```

```
# Discretize the parameter space
theta0 = np.linspace(0.0, 2.0 * np.pi, 100)

# Load energy value at each point in parameter space
pes = np.load("deut_pes_n2.npy")

# Get the minimum of the PES
minloc=np.unravel_index(pes.argmin(),pes.shape)

# Plot energy landscape
fig, axes = plt.subplots(figsize=(6, 6))

plt.plot(theta0,pes[:,0],label="Underlying Surface (MeV)")

plt.xlabel(r"$\theta_0$")
plt.ylabel(r"Energy (MeV)")
plt.plot(theta0[minloc[0]],cost_fn([theta0[minloc[0]],]),"r*",markersize=20,label=
↪"Minimum")

# Plot optimization path for gradient descent. Plot every 10th point.
gd_color = "g"
```

(continues on next page)

(continued from previous page)

```
plt.plot(
    np.array(gd_param_history)[::10, 0],
    np.array(gd_cost_history)[::10],
    ".",markersize=12,
    color=gd_color,
    linewidth=2,
    label="Gradient descent",
)
plt.plot(
    np.array(gd_param_history)[:-1, 0],
    np.array(gd_cost_history)[:],
    "-",
    color=gd_color,
    linewidth=1,
)


plt.legend()
plt.show()
```



Perfect! Now that you've got this working, you can go back and play around with the parameters of the VQE, potential,

initial guesses, etc. and see how your convergence changes! When it comes to running on real hardware for serious problems, one key goal is to get the number of evaluations as low as possible. This is an even bigger deal for larger circuits, like the N=3 case we'll look at next!

2.1.4 Complexity++: N=3

Author(s): Kyle Godbey

Maintainer: Kyle Godbey

Our previous example was great for introducing us to the methods and computational framework, but it's time to take another step towards something a little trickier and expand our basis by one.

The key difference here is that now our ansatz will require another parameter, making it a 2d optimization problem. Our Hamiltonian will also change of course, meaning the bulk of the changes will be in those two cells since our VQE machinery can still handle it well. Even though most of this page will be the same as the last be sure to step through it all if only to see the nice two dimensional plot of the PES.

Let's set up imports again, it never gets old! We will again just use pennylane and define the circuits and Hamiltonian directly in the form from Cloud Quantum Computing of an Atomic Nucleus.

```
%matplotlib inline

import matplotlib.pyplot as plt
from pennylane import numpy as np
import pennylane as qml
import warnings
warnings.filterwarnings('ignore')

plt.style.use(['science','notebook'])
```

Now we will define our 'device' (a simulator in this case) as well as our ansatz and Hamiltonian.

Here is where the first big change comes in, since we now need 3 qubits. Our ansatz circuit is thus a little more complicated than before, being a little deeper and requiring another variational parameter.

Our Hamiltonian is also changed, of course, but here we're still writing it out by hand, with automatic generation being introduced next.

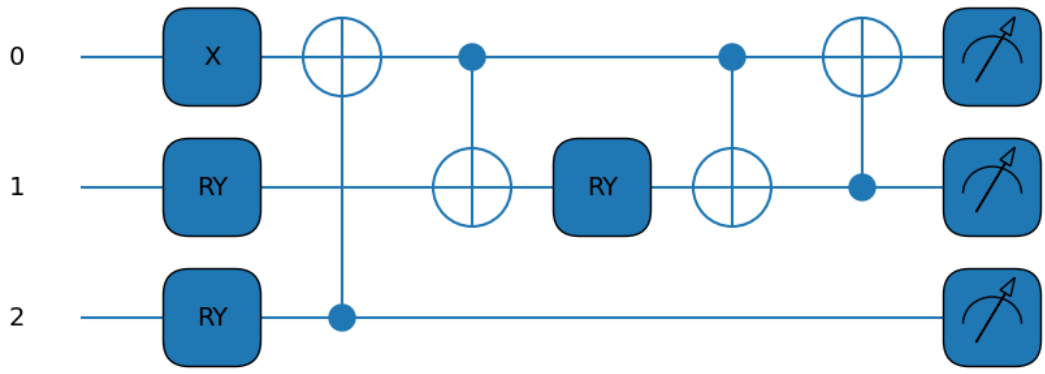
```
# In this case we now need 3 qubits

dev = qml.device("default.qubit", wires=3)

# Defining our ansatz circuit for the N=3 case

def circuit(params,wires):
    t0 = params[0]
    t1 = params[1]
    qml.PauliX(wires=0)
    qml.RY(t1, wires=1)
    qml.RY(t0, wires=2)
    qml.CNOT(wires=[2,0])
    qml.CNOT(wires=[0,1])
    qml.RY(-t1, wires=1)
    qml.CNOT(wires=[0,1])
    qml.CNOT(wires=[1,0])
```

(continues on next page)

(continued from previous page)

```
# And building our Hamiltonian for the N=3 case

coeffs = [15.531709 ,-2.1433, -2.1433, 0.21829, -6.125, -9.625, -3.91, -3.91]
obs = [qml.Identity(0),qml.PauliX(0) @ qml.PauliX(1), qml.PauliY(0) @ qml.PauliY(1),
 ↪qml.PauliZ(0),qml.PauliZ(1),qml.PauliZ(2),qml.PauliX(1) @ qml.PauliX(2),qml.
 ↪PauliY(1) @ qml.PauliY(2)]

H = qml.Hamiltonian(coeffs, obs)
cost_fn = qml.ExpvalCost(circuit, H, dev)

# Let's print it out

print(H)

@qml.qnode(dev)
def draw_circuit(params):
    circuit(params,wires=dev.wires)
    return qml.expval(H)

qml.drawer.use_style('default')
print(qml.draw_mpl(draw_circuit)(init_params))
```

```
  (-9.625) [Z2]
+ (-6.125) [Z1]
+ (0.21829) [Z0]
+ (15.531709) [I0]
+ (-3.91) [X1 X2]
+ (-3.91) [Y1 Y2]
+ (-2.1433) [X0 X1]
+ (-2.1433) [Y0 Y1]
(<Figure size 900x400 with 1 Axes>, <Axes:>)
```



Amazing! That's an even better looking Hamiltonian! The circuit is starting to look a little more involved as well.

Now we'll set up what we'll need for the VQE procedure; namely some initial parameters and convergence info. You can select the initial guess randomly, but in this case we'll set it manually for repeatability.

In this case, we also need an additional parameter as mentioned when we set up our circuit.

```
# Our parameter array, now our params have a friend :3

init_params = np.array([2.5,4.5])

# Convergence information and step size

max_iterations = 500
conv_tol = 1e-06
step_size = 0.01
```

Finally, the VQE block. We're still using the standard gradient descent optimizer since it worked so well before, but soon the time will come to shop around for better options.

```
opt = qml.GradientDescentOptimizer(stepsize=step_size)

params = init_params

gd_param_history = [params]
gd_cost_history = []

for n in range(max_iterations):

    # Take a step in parameter space and record your energy
    params, prev_energy = opt.step_and_cost(cost_fn, params)

    # This keeps track of our energy for plotting at comparisons
    gd_param_history.append(params)
    gd_cost_history.append(prev_energy)

    # Here we see what the energy of our system is with the new parameters
    energy = cost_fn(params)

    # Calculate difference between new and old energies
    conv = np.abs(energy - prev_energy)

    if n % 20 == 0:
        print(
            "Iteration = {:},  Energy = {:.8f} MeV,  Convergence parameter = {"
            ":.8f} MeV".format(n, energy, conv)
        )

    if conv <= conv_tol:
        break

print()
print("Final value of the energy = {:.8f} MeV".format(energy))
print("Number of iterations = ", n)
```

```
Iteration = 0,  Energy = 22.73185830 MeV,  Convergence parameter = 0.16533643 MeV
Iteration = 20,  Energy = 13.86733622 MeV,  Convergence parameter = 0.85950160 MeV
Iteration = 40,  Energy = 0.66226895 MeV,  Convergence parameter = 0.41277410 MeV
Iteration = 60,  Energy = -1.94958676 MeV,  Convergence parameter = 0.01827050 MeV
Iteration = 80,  Energy = -2.04226133 MeV,  Convergence parameter = 0.00054725 MeV
Iteration = 100,  Energy = -2.04502496 MeV,  Convergence parameter = 0.00001629 MeV

Final value of the energy = -2.04510468 MeV
```

(continues on next page)

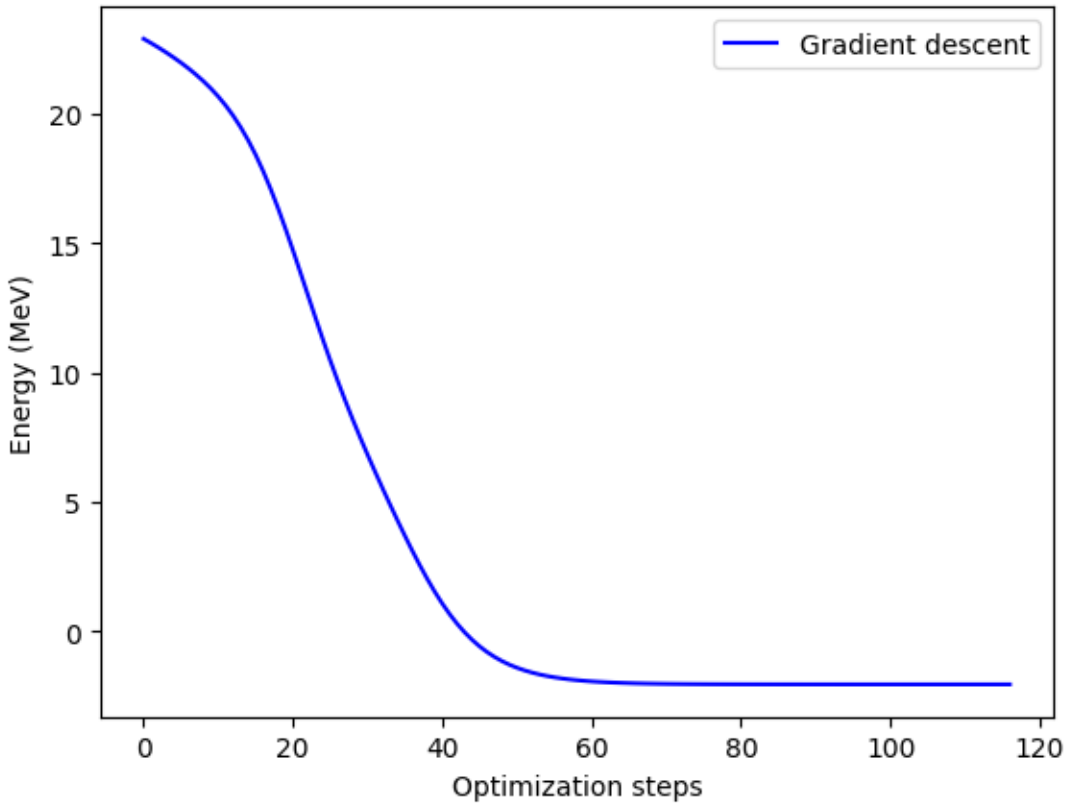
(continued from previous page)

```
Number of iterations =  116
```

As expected, we get closer to the true value with our expanded basis, though we're not all the way there. We still match our reference value very well, meaning we're on the right track for building out our robust nuclear VQE framework, so good job team!

Let's plot the convergence next and see if going to 2-D hurt us any.

```
plt.plot(gd_cost_history, "b", label="Gradient descent")

plt.ylabel("Energy (MeV)")
plt.xlabel("Optimization steps")
plt.legend()
plt.show()
```



Finally, we can plot how we trace the potential energy surface (PES) as we find a solution. Now that we've moved up a dimension, we'll be plotting an actual surface instead of a line, so be sure to take note of the features of the surface in parameter space.

If you're running this yourself, you can either generate the surface yourself or put this code in a cell to download it locally:

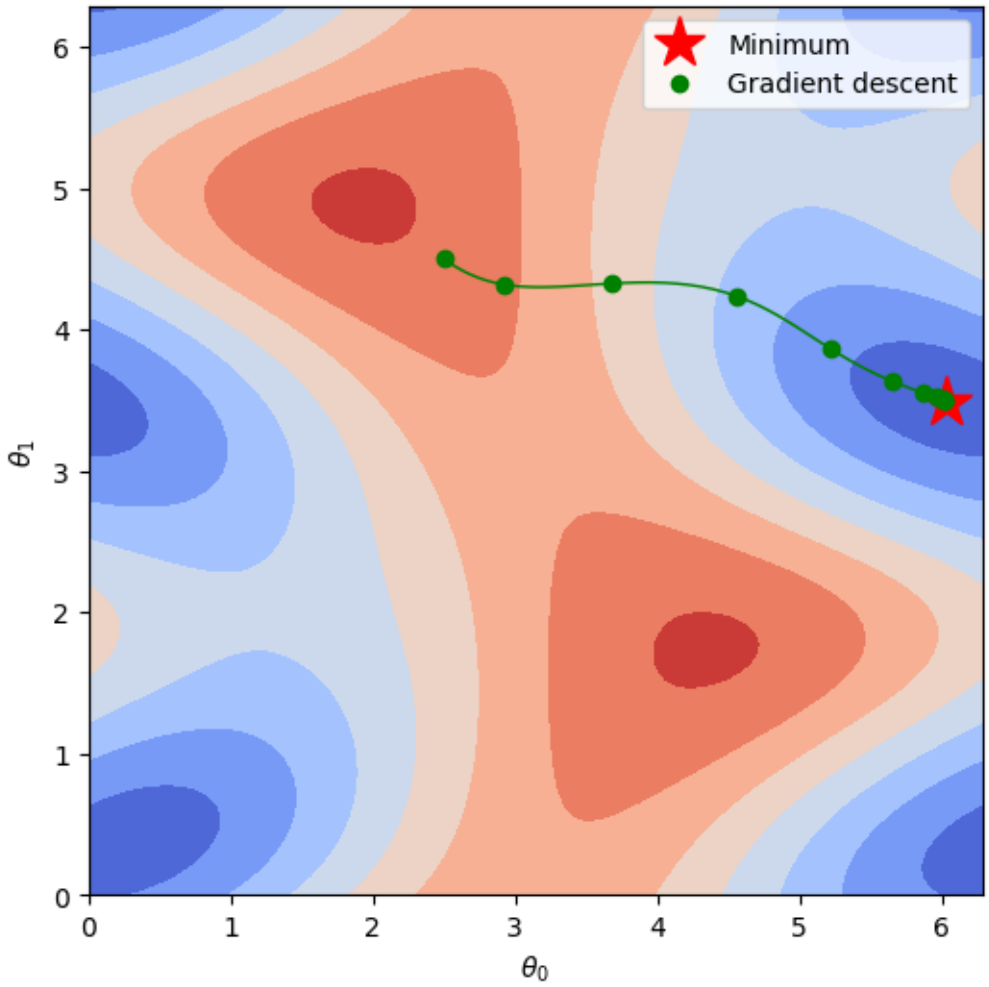
```
!wget        https://github.com/kylegodbey/nuclear-quantum-computing/raw/main/
nuclear-qc/vqe/deut_pes_n2.npy
```

```
# Discretize the parameter space
theta0 = np.linspace(0.0, 2.0 * np.pi, 100)
theta1 = np.linspace(0.0, 2.0 * np.pi, 100)
```

(continues on next page)

(continued from previous page)

```
# Load energy value at each point in parameter space
pes = np.load("deut_pes_n3.npy")

# Get the minimum of the PES
minloc=np.unravel_index(pes.argmin(),pes.shape)

# Plot energy landscape
fig, axes = plt.subplots(figsize=(6, 6))

cmap = plt.cm.get_cmap("coolwarm")
contour_plot = plt.contourf(theta0, theta1, pes.T, cmap=cmap)

plt.xlabel(r"$\theta_0$")
plt.ylabel(r"$\theta_1$")

plt.plot(theta0[minloc[0]],theta1[minloc[1]],"r*",markersize=20,label="Minimum")

# Plot optimization path for gradient descent. Plot every 10th point.
gd_color = "g"

plt.plot(
    np.array(gd_param_history)[::10, 0],
    np.array(gd_param_history)[::10, 1],
    ".",markersize=12,
    color=gd_color,
    linewidth=2,
    label="Gradient descent",
)
plt.plot(
    np.array(gd_param_history)[:, 0],
    np.array(gd_param_history)[:, 1],
    "-",
    color=gd_color,
    linewidth=1,
)


plt.legend()
plt.show()
```



This is all looking very nice, so now let's take it to the extreme! Next up we're going to programmatically generate our Hamiltonian and ansatz to an arbitrary, user-defined order. This means we'll probably take a big performance hit, but shooting for scalability is always a worthy endeavor.

2.1.5 A General Case

Author(s): Kyle Godbey

Maintainer: Kyle Godbey

To take this particular problem to its natural conclusion, we will be now devising a way to automatically generate our Hamiltonian and ansatz circuits based on what basis dimension we want to calculate.

To do this in the most straightforward manner, I will first implement the Hamiltonian in second quantization before mapping it to a Pauli representation. To help out, I'll be importing some tools from qiskit as well as pennylane, proving that frameworks can indeed work together!

```
%matplotlib inline

import matplotlib.pyplot as plt
from pennylane import numpy as np
```

(continues on next page)

(continued from previous page)

```
import pennylane as qml
from qiskit_nature.operators.second_quantization import FermionicOp
from qiskit_nature.mappers.second_quantization import JordanWignerMapper, ParityMapper
import time
from functools import partial
import warnings
warnings.filterwarnings('ignore')

plt.style.use(['science','notebook'])
```

We'll also need some helper functions to parse Pauli strings and construct our pennylane Hamiltonian, so I'll just dump them here.

```
def pauli_token_to_operator(token):
    qubit_terms = []

    for term in range(len(token)):
        # Special case of identity
        if token[term] == "I":
            pass
        else:
            #pauli, qubit_idx = term, term
            if token[term] == "X":
                qubit_terms.append(qml.PauliX(int(term)))
            elif token[term] == "Y":
                qubit_terms.append(qml.PauliY(int(term)))
            elif token[term] == "Z":
                qubit_terms.append(qml.PauliZ(int(term)))
            else:
                print("Invalid input.")
    if(qubit_terms==[]):
            qubit_terms.append(qml.Identity(0))
    full_term = qubit_terms[0]
    for term in qubit_terms[1:]:
        full_term = full_term @ term

    return full_term


def parse_hamiltonian_input(input_data):
    # Get the input
    coeffs = []
    pauli_terms = []
    chunks = input_data.split("\n")
    # Go through line by line and build up the Hamiltonian
    for line in chunks:
        #line = line.strip()
        tokens = line.split(" ")
        # Parse coefficients
        sign, value = tokens[0][0], tokens[1]

        coeff = float(value)
        if sign == "-":
            coeff *= -1
        coeffs.append(coeff)
```

(continues on next page)

(continued from previous page)

```
        # Parse Pauli component
        pauli = tokens[3][::-1]

        pauli_terms.append(pauli_token_to_operator(pauli))

    return qml.Hamiltonian(coeffs, pauli_terms)
```

Something else that will be very important is a function to compute the matrix elements $\langle n'|(\hat{T}+\hat{V})|n\rangle$ needed for our Hamiltonian.

```
def kron(i,j):
    if(i==j):
        return 1
    else:
        return 0

def matrix_element(i,j):

    ele = 0.0

    ele = ((2.0*i + 1.5)*kron(i,j) - np.sqrt(i*(i+0.5))*kron(i,j+1) \
    - np.sqrt((i+1)*(i+1.5))*kron(i,j-1)) * 3.5

    ele += -5.68658111 * kron(i,0) * kron(i,j)


    return ele
```

Lastly, we'll make a function that returns a pennylane compatible Hamiltonian given a requested basis dimension, N.

```
def deuteron_ham(N,mapper=JordanWignerMapper):
    # Start out by zeroing what will be our fermionic operator
    op = 0
    for i in range(N):
        for j in range(N):
            # Construct the terms of the Hamiltonian in terms of creation/
 ↪annihilation operators
            op += matrix_element(i,j) * \
            FermionicOp([(("+", i),("-", j)], 1.0)])

    hamstr = "+ "+str(mapper().map(second_q_op=op))

    hamiltonian = parse_hamiltonian_input(hamstr)

    return hamiltonian
```

Let's take a look at our new Hamiltonian generator in the next code block. We'll start by setting the basis dimension to 3 so we can compare our two Hamiltonians: handcrafted and computer generated.

```
dim = 3

# Building our Hamiltonian for the N=3 case, as before

coeffs = [15.531709 ,-2.1433, -2.1433, 0.21829, -6.125, -9.625, -3.91, -3.91]
obs = [qml.Identity(0),qml.PauliX(0) @ qml.PauliX(1), qml.PauliY(0) @ qml.PauliY(1),
 ↪qml.PauliZ(0),qml.PauliZ(1),qml.PauliZ(2),qml.PauliX(1) @ qml.PauliX(2),qml.
 ↪PauliY(1) @ qml.PauliY(2)]
```

(continues on next page)

(continued from previous page)

```
ham = qml.Hamiltonian(coeffs, obs)

# Let's print it out

print("Original Hamiltonian: \n",ham)

# Let's now use our new function!

H = deuteron_ham(dim)

print("Generated Hamiltonian: \n",H)
```

```
Original Hamiltonian:
   (-9.625) [Z2]
+ (-6.125) [Z1]
+ (0.21829) [Z0]
+ (15.531709) [I0]
+ (-3.91) [X1 X2]
+ (-3.91) [Y1 Y2]
+ (-2.1433) [X0 X1]
+ (-2.1433) [Y0 Y1]
Generated Hamiltonian:
   (-9.625) [Z2]
+ (-6.125) [Z1]
+ (0.21829055499999983) [Z0]
+ (15.531709445) [I0]
+ (-3.913118960624632) [Y1 Y2]
+ (-3.913118960624632) [X1 X2]
+ (-2.1433035249352805) [Y0 Y1]
+ (-2.1433035249352805) [X0 X1]
```

We still need to define our device and circuit, so we'll let them be general as well.

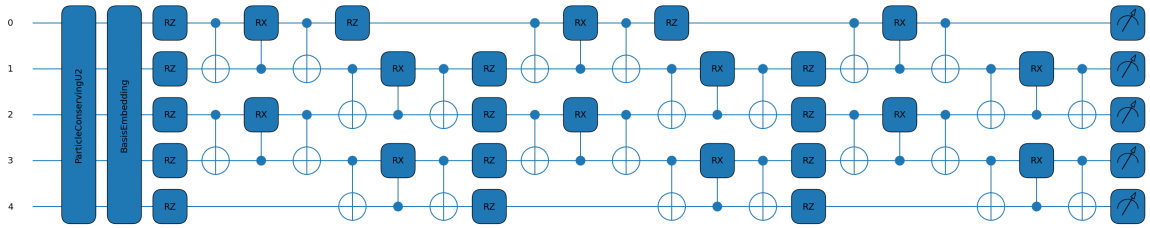
```
# Set the order you'd like to go to

dim = 5

dev = qml.device("default.qubit", wires=dim)

# Define a general ansatz for arbitrary numbers of dimensions

particles = 1

ref_state = qml.qchem.hf_state(particles, dim)

ansatz = partial(qml.ParticleConservingU2, init_state=ref_state)

layers = dim - 2

@qml.qnode(dev)
def draw_circuit(params):
    ansatz(params,wires=dev.wires).decomposition()
    return qml.expval(H)
```

(continues on next page)

(continued from previous page)

```
# Defining Hamiltonian

H = deuteron_ham(dim)

print(H)

cost_fn = qml.ExpvalCost(ansatz, H, dev)

qml.drawer.use_style('default')
print(qml.draw_mpl(draw_circuit)(init_params))
```

```
  (-16.625) [Z4]
+ (-13.125) [Z3]
+ (-9.625) [Z2]
+ (-6.125) [Z1]
+ (0.21829055499999983) [Z0]
+ (45.281709445000004) [I0]
+ (-7.424621202458749) [Y3 Y4]
+ (-7.424621202458749) [X3 X4]
+ (-5.670648111106878) [Y2 Y3]
+ (-5.670648111106878) [X2 X3]
+ (-3.913118960624632) [Y1 Y2]
+ (-3.913118960624632) [X1 X2]
+ (-2.1433035249352805) [Y0 Y1]
+ (-2.1433035249352805) [X0 X1]
(<Figure size 2600x600 with 1 Axes>, <Axes:>)
```



Quintuple Amazing! That's yet another good looking Hamiltonian, and the circuit isn't bad either…

Now we'll set up what we'll need for the VQE procedure, this time doing a loop over how many parameters we'll need.

```
# Our parameter array

init_params = np.random.uniform(low=-np.pi / 2, high=np.pi / 2, size=qml.
 ↪ParticleConservingU2.shape(n_layers=layers, n_wires=dim))

# Convergence information and step size

max_iterations = 500
conv_tol = 1e-05
step_size = 0.1
```

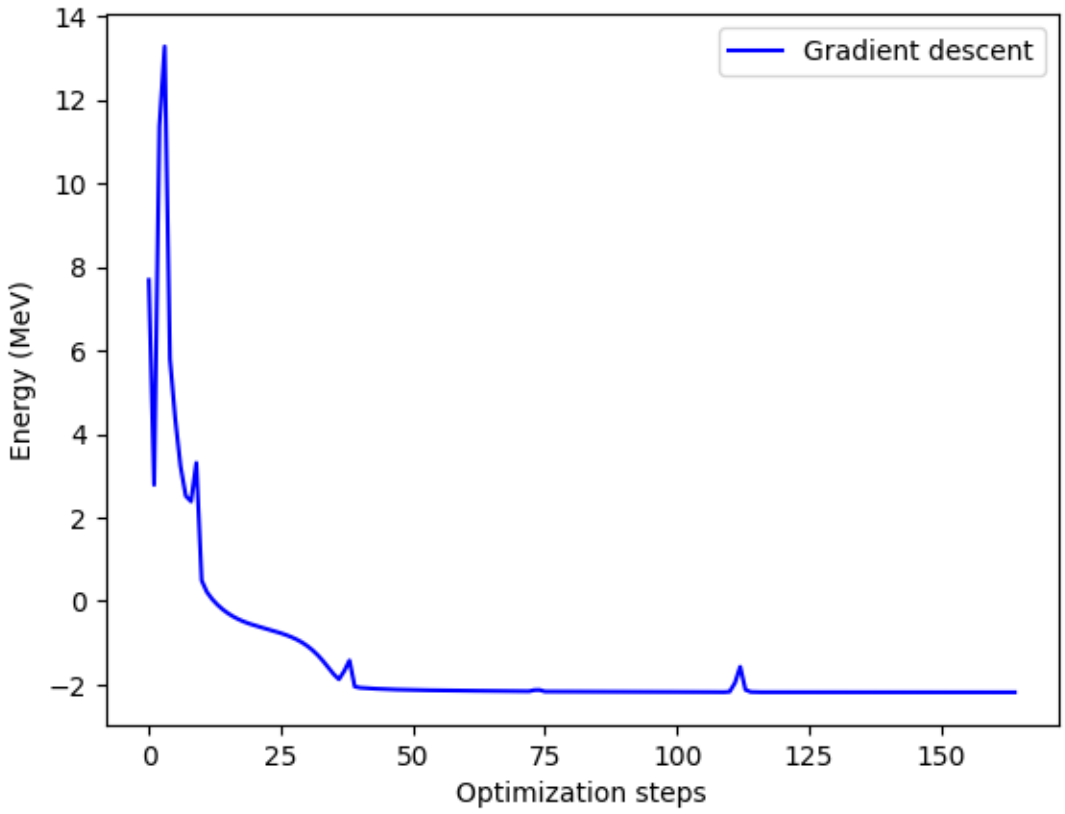
Finally, the VQE block. We're still using the standard gradient descent optimizer since it worked so well before, but soon the time will come to shop around for better options. I have also added a quick and dirty variable step size just to speed things up slightly.

```
opt = qml.GradientDescentOptimizer(stepsize=step_size)

params = init_params

gd_param_history = [params]
gd_cost_history = []

accel = 0
prev_conv = -1.0

start = time.time()

for n in range(max_iterations):
    fac = (1.0)
    opt = qml.GradientDescentOptimizer(stepsize=step_size)

    # Take a step in parameter space and record your energy
    params, prev_energy = opt.step_and_cost(cost_fn, params)

    # This keeps track of our energy for plotting at comparisons
    gd_param_history.append(params)
    gd_cost_history.append(prev_energy)

    # Here we see what the energy of our system is with the new parameters
    energy = cost_fn(params)

    # Calculate difference between new and old energies
    conv = np.abs(energy - prev_energy)

    if(energy - prev_energy > 0.0 and step_size > 0.001):
        #print("Lowering!")
        accel = 0
        step_size = 0.5*step_size

    if(conv < prev_conv): accel += 1
    prev_conv = conv

    if(accel > 10 and step_size < 1.0):
        #print("Accelerating!")
        step_size = 1.1*step_size
    end = time.time()

    if n % 10 == 0:
        print(
            "It = {:},  Energy = {:.8f} MeV,  Conv = {"
            ":.8f} MeV, Time Elapsed = {:.3f} s".format(n, energy, conv,end-start)
        )
        start = time.time()


    if conv <= conv_tol:
        break

print()
print("Final value of the energy = {:.8f} MeV".format(energy))
print("Number of iterations = ", n)
```

```
It = 0,  Energy = 25.43598895 MeV,  Conv = 3.74111148 MeV, Time Elapsed = 0.811 s
It = 10,  Energy = -1.84990698 MeV,  Conv = 0.01447123 MeV, Time Elapsed = 8.448 s
It = 20,  Energy = -1.98298605 MeV,  Conv = 0.01369684 MeV, Time Elapsed = 8.302 s
It = 30,  Energy = -2.14395469 MeV,  Conv = 0.01093344 MeV, Time Elapsed = 8.459 s
It = 40,  Energy = -2.16828413 MeV,  Conv = 0.00097018 MeV, Time Elapsed = 8.162 s
It = 50,  Energy = -2.17146313 MeV,  Conv = 0.00033026 MeV, Time Elapsed = 8.317 s
It = 60,  Energy = -2.17539857 MeV,  Conv = 0.00686297 MeV, Time Elapsed = 8.247 s
It = 70,  Energy = -2.17669224 MeV,  Conv = 0.00011480 MeV, Time Elapsed = 8.408 s
It = 80,  Energy = -2.17847872 MeV,  Conv = 0.00023824 MeV, Time Elapsed = 8.537 s
It = 90,  Energy = -2.18142522 MeV,  Conv = 0.00030970 MeV, Time Elapsed = 8.183 s
It = 100,  Energy = -2.18264553 MeV,  Conv = 0.00060788 MeV, Time Elapsed = 8.361 s
It = 110,  Energy = -2.18294148 MeV,  Conv = 0.00002063 MeV, Time Elapsed = 8.373 s
It = 120,  Energy = -2.18321824 MeV,  Conv = 0.00003227 MeV, Time Elapsed = 8.444 s

Final value of the energy = -2.18336450 MeV
Number of iterations =  129
```

If you choose a bigger basis, you do a better job! This is to be expected, but notice that we've taken quite the performance hit by going to a more general representation. We'll explore how to do better at this in the optimization section, but for now let's check the convergence.

```
plt.plot(gd_cost_history, "b", label="Gradient descent")

plt.ylabel("Energy (MeV)")
plt.xlabel("Optimization steps")
plt.legend()
plt.show()
```



Since we are now allowed to move in higher dimensions, making a general PES plotter gets a little tricky, but feel free to play around and see if you can think of clever ways to plot how your solution explores the space!

This pretty much concludes our exploration of quantum computing for the deuteron from the point of view of Hamiltonian and state preparation, but there's plenty of other things that can be tried to improve performance! One approach would involve optimizing the ansatz instead of using a generic form, which could be a fun challenge. Another approach is to speed up the VQE algorithm itself by being clever with how we iterate which is what we'll briefly explore next.

2.2 Quantum Natural Gradient

Author(s): Kyle Godbey

Maintainer: Kyle Godbey

For this last application, let's think of a way to improve our VQE algorithm by exploiting some features of our quantum computing architecture. What we'll do is, instead of computing the gradient of the variational parameters of our ansatz, we'll instead compute the so-called Quantum Natural Gradient. This is an extension of the standard natural gradient which moves in the steepest descent direction of the information geometry, using something like the Fisher metric. For these quantum systems, we'll instead use the Fubini-Study metric (real part of the quantum geometric tensor) to determine the most optimal step in the ansatz parameters. For more details you should have a look at the Quantum Natural Gradient paper published in 2020.

While it's not too difficult to calculate this measure yourself by constructing the Hessian using parameter shift rules, we'll be using a built in pennylane method to make our lives easier.

Most of this will be the same as the N=3 case, so we'll toss it all in a cell together – be sure to check that you know what's happening though!

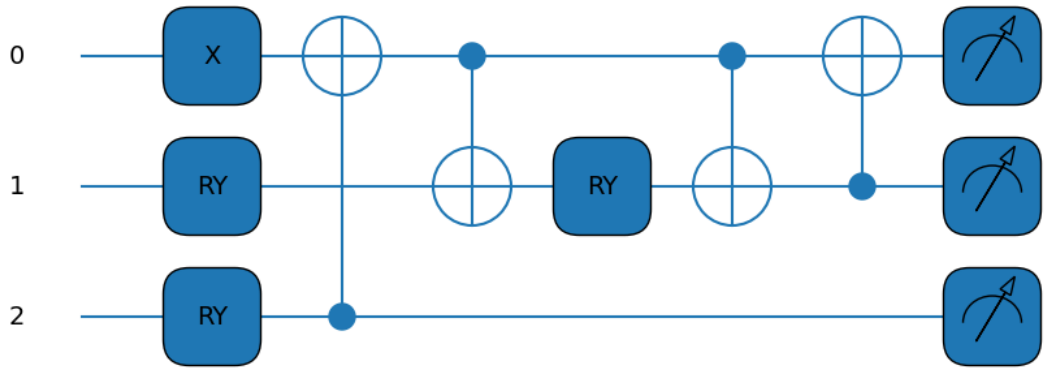
```
%matplotlib inline

import matplotlib.pyplot as plt
from pennylane import numpy as np
import pennylane as qml
import warnings
warnings.filterwarnings('ignore')

plt.style.use(['science','notebook'])
# In this case we now need 3 qubits

dev = qml.device("default.qubit", wires=3)

# Defining our ansatz circuit for the N=3 case

def circuit(params,wires):
    t0 = params[0]
    t1 = params[1]
    qml.PauliX(wires=0)
    qml.RY(t1, wires=1)
    qml.RY(t0, wires=2)
    qml.CNOT(wires=[2,0])
    qml.CNOT(wires=[0,1])
    qml.RY(-t1, wires=1)
    qml.CNOT(wires=[0,1])
    qml.CNOT(wires=[1,0])

# And building our Hamiltonian for the N=3 case
```

(continues on next page)

(continued from previous page)

```
coeffs = [15.531709 ,-2.1433, -2.1433, 0.21829, -6.125, -9.625, -3.91, -3.91]
obs = [qml.Identity(0),qml.PauliX(0) @ qml.PauliX(1), qml.PauliY(0) @ qml.PauliY(1),
 ↪qml.PauliZ(0),qml.PauliZ(1),qml.PauliZ(2),qml.PauliX(1) @ qml.PauliX(2),qml.
 ↪PauliY(1) @ qml.PauliY(2)]

H = qml.Hamiltonian(coeffs, obs)
cost_fn = qml.ExpvalCost(circuit, H, dev)

print(H)

@qml.qnode(dev)
def draw_circuit(params):
    circuit(params,wires=dev.wires)
    return qml.expval(H)

qml.drawer.use_style('default')
print(qml.draw_mpl(draw_circuit)(init_params))
```

```
  (-9.625) [Z2]
+ (-6.125) [Z1]
+ (0.21829) [Z0]
+ (15.531709) [I0]
+ (-3.91) [X1 X2]
+ (-3.91) [Y1 Y2]
+ (-2.1433) [X0 X1]
+ (-2.1433) [Y0 Y1]
(<Figure size 900x400 with 1 Axes>, <Axes:>)
```



Now we'll still set up what we need for our VQE procedure, but we'll be running it twice, so setting the initial parameters is a necessity for the sake of fairness.

```
# Hand-picked, small-batch initialization

init_params = np.array([2.5,4.5])
```

(continues on next page)

(continued from previous page)

```
# Convergence information and step size

max_iterations = 500
conv_tol = 1e-06
step_size = 0.01
```

For the VQE block, we'll run two one after another. First up is the same thing we had before, good ol' gradient descent.

```
opt = qml.GradientDescentOptimizer(stepsize=step_size)

params = init_params

gd_param_history = [params]
gd_cost_history = []

for n in range(max_iterations):

    # Take a step in parameter space and record your energy
    params, prev_energy = opt.step_and_cost(cost_fn, params)

    # This keeps track of our energy for plotting at comparisons
    gd_param_history.append(params)
    gd_cost_history.append(prev_energy)

    # Here we see what the energy of our system is with the new parameters
    energy = cost_fn(params)

    # Calculate difference between new and old energies
    conv = np.abs(energy - prev_energy)

    if n % 20 == 0:
        print(
            "Iteration = {:},  Energy = {:.8f} MeV,  Convergence parameter = {"
            ":.8f} MeV".format(n, energy, conv)
        )

    if conv <= conv_tol:
        break

print()
print("Final value of the energy = {:.8f} MeV".format(energy))
print("Number of iterations = ", n)
```

```
Iteration = 0,  Energy = 22.73185830 MeV,  Convergence parameter = 0.16533643 MeV
Iteration = 20,  Energy = 13.86733622 MeV,  Convergence parameter = 0.85950160 MeV
Iteration = 40,  Energy = 0.66226895 MeV,  Convergence parameter = 0.41277410 MeV
Iteration = 60,  Energy = -1.94958676 MeV,  Convergence parameter = 0.01827050 MeV
Iteration = 80,  Energy = -2.04226133 MeV,  Convergence parameter = 0.00054725 MeV
Iteration = 100,  Energy = -2.04502496 MeV,  Convergence parameter = 0.00001629 MeV

Final value of the energy = -2.04510468 MeV
Number of iterations =  116
```

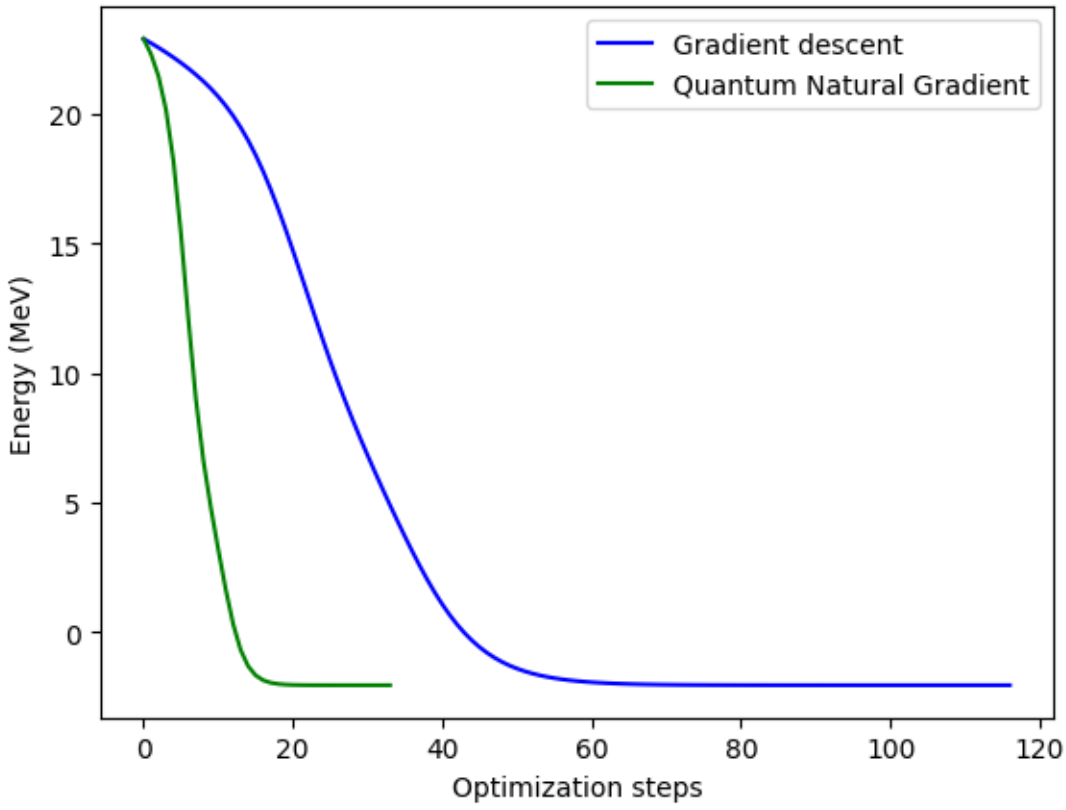
Looks the same as before, that's a good sign. Now let's try the QNG!

```
opt = qml.QNGOptimizer(stepsize=step_size)

params = init_params

qng_param_history = [params]
qng_cost_history = []

for n in range(max_iterations):

    # Take a step in parameter space and record your energy
    params, prev_energy = opt.step_and_cost(cost_fn, params)

    # This keeps track of our energy for plotting at comparisons
    qng_param_history.append(params)
    qng_cost_history.append(prev_energy)

    # Here we see what the energy of our system is with the new parameters
    energy = cost_fn(params)

    # Calculate difference between new and old energies
    conv = np.abs(energy - prev_energy)

    if n % 20 == 0:
        print(
            "Iteration = {:},  Energy = {:.8f} MeV,  Convergence parameter = {"
            ":.8f} MeV".format(n, energy, conv)
        )

    if conv <= conv_tol:
        break

print()
print("Final value of the energy = {:.8f} MeV".format(energy))
print("Number of iterations = ", n)
```

```
Iteration = 0,  Energy = 22.30149608 MeV,  Convergence parameter = 0.59569865 MeV
Iteration = 20,  Energy = -2.03927628 MeV,  Convergence parameter = 0.00595511 MeV

Final value of the energy = -2.04510917 MeV
Number of iterations =  33
```

Very nice! We sped things up a good bit, but let's check the convergence plot to see the behavior over the iterations.

```
plt.plot(gd_cost_history, "b", label="Gradient descent")
plt.plot(qng_cost_history, "g", label="Quantum Natural Gradient")

plt.ylabel("Energy (MeV)")
plt.xlabel("Optimization steps")
plt.legend()
plt.show()
```



Wow! It seems taking a more intelligent step in parameter space can work wonders towards finding an optimal solution. This works much better than simply adding a variable time step, like we did for the general case. Try implementing the QNG optimizer for higher dimensions and check the performance!

Finally, we can plot how we trace the potential energy surface (PES) as we find a solution. It's a little more exciting now that we have two paths – let's imagine they're racing through the mountains.

If you're running this yourself, you can either generate the surface yourself or put this code in a cell to download it locally:

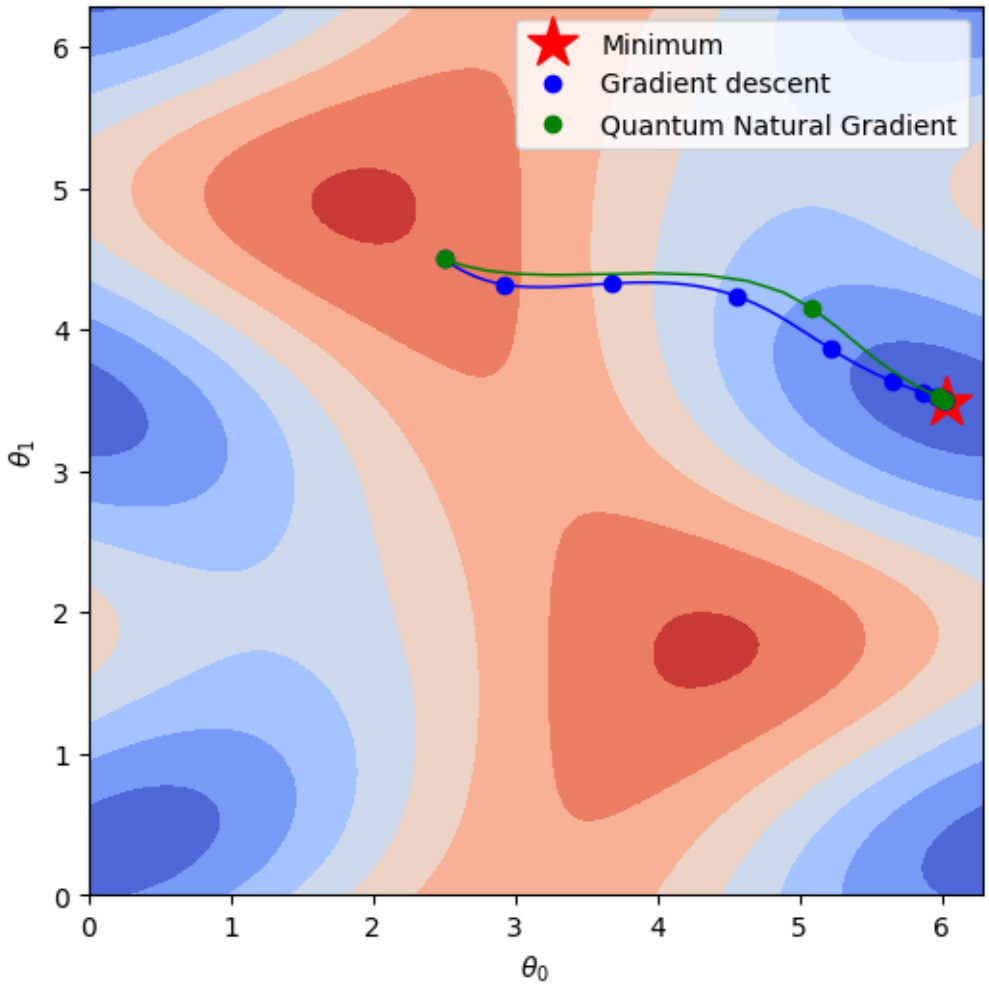
```
!wget https://github.com/kylegodbey/nuclear-quantum-computing/raw/main/
nuclear-qc/vqe/deut_pes_n3.npy
```

```
# Discretize the parameter space
theta0 = np.linspace(0.0, 2.0 * np.pi, 100)
theta1 = np.linspace(0.0, 2.0 * np.pi, 100)

# Load energy value at each point in parameter space
pes = np.load("deut_pes_n3.npy")

# Get the minimum of the PES
minloc=np.unravel_index(pes.argmin(),pes.shape)

# Plot energy landscape
fig, axes = plt.subplots(figsize=(6, 6))

cmap = plt.cm.get_cmap("coolwarm")
contour_plot = plt.contourf(theta0, theta1, pes.T, cmap=cmap)

plt.xlabel(r"$\theta_0$")
```

(continues on next page)

(continued from previous page)

```
plt.ylabel(r"$\theta_1$")

plt.plot(theta0[minloc[0]],theta1[minloc[1]],"r*",markersize=20,label="Minimum")

# Plot optimization path for gradient descent. Plot every 10th point.
gd_color = "b"

plt.plot(
    np.array(gd_param_history)[::10, 0],
    np.array(gd_param_history)[::10, 1],
    ".",markersize=12,
    color=gd_color,
    linewidth=2,
    label="Gradient descent",
)
plt.plot(
    np.array(gd_param_history)[:, 0],
    np.array(gd_param_history)[:, 1],
    "-",
    color=gd_color,
    linewidth=1,
)

# Plot optimization path for qng. Plot every 10th point.
qng_color = "g"

plt.plot(
    np.array(qng_param_history)[::10, 0],
    np.array(qng_param_history)[::10, 1],
    ".",markersize=12,
    color=qng_color,
    linewidth=2,
    label="Quantum Natural Gradient",
)
plt.plot(
    np.array(qng_param_history)[:, 0],
    np.array(qng_param_history)[:, 1],
    "-",
    color=qng_color,
    linewidth=1,
)

plt.legend()
plt.show()
```



You can see that the paths are not massively different, but the QNG results in a much sharper gradient in the parameter space. Exemplified by the spacing between the points along the line, showing the value every ten time steps. This steeper landscape leads to bigger steps towards the minimum without risking overshooting, making it an excellent fit in situations where you can make use of it!

This concludes our exploration of playing with VQE ad nauseam, next up we'll explore what happens when things get a bit more noisy.

CHAPTER

THREE

RUNNING ON "HARDWARE"

Author: Alexandra Semposki

In the last chapter, we extensively discussed the VQE method and showed its results on a simulator. However, we did not explain much about that simulator—what is a quantum simulator in comparison to a real quantum computer? What differences arise between the two, and what issues can one run into using the two devices?

A **quantum simulator**, such as the `qiskit.Aer` simulator, is a device that mimics a real quantum computer, but in the sense that it computes everything the way a quantum computer would do, *except* it acts as if the device has no noise. This is not realistic because it is known that real quantum devices will have noise in their circuits (a full explanation of sources and effects of noise in a real quantum device is given on the next page).

A **real quantum computer**, such as the IBM quantum machines available for use through Qiskit, have a limited number of qubits to use, and these qubits all have errors come up when running quantum circuits with them. Quantum simulators generally have many more qubits available for use, but do tend to only be able to accurately calculate up to a few dozen qubits.

The quantum simulator is very useful for testing out a quantum circuit or full code before sending it to a real quantum computer and waiting for your turn in the queue. Since the IBM quantum computers available are a limited number, there can be long job queues until your circuit is run, so double-checking your work on a simulator first is always a good idea. But what about when you get your results back from the quantum computer and you see you have errors in your results from the real devices? What then? On the next page, and in much of this chapter, we will be exploring different kinds of error correction and mitigation, and trying out an error mitigation technique on the $N = 2$ case of the deuteron from the last chapter to see how the world of errors works in quantum computing.

3.1 Noisy Circuits

Author: Alexandra Semposki

Maintainer: Alexandra Semposki

Now that VQE has been extensively covered for a noiseless system, we can begin to consider what happens when we add noise into the circuit. Of course, some questions one would have are how the noise occurs in the circuit, how does it affect the circuit results, and how would one correct such errors? In this notebook, we'll cover all of these questions and end with an error mitigation technique that will become very useful for the calculations in the rest of this Jupyter Book.

3.1.1 On the origin of noise

If you've only ever run a quantum circuit on a simulator, you've probably never encountered errors on your qubits (unless you were using a noisy backend, of course). However, if you run your circuit(s) on a real quantum device, like IBM's quantum computers, you'll notice (once your job stops running) that you don't have just the results you thought you would—instead, you'll have some probabilities in bins you didn't expect at all. This is an example of this noise we're talking about. There are two clear places errors can occur that you'll probably be able to guess quite quickly:

- Qubits
- Quantum gates

Both of these hardware components are highly susceptible to error. Some of the general types of error are described below.

- **Decoherence**: a qubit will eventually interact with its environment in a real quantum device, which leads to the qubit entangling with this environment, which in turn leads to the qubit altering from its original state into this entangled state.
- **Dephasing**: this term describes the process of a qubit's phase changing over time; this can be visualised by thinking of a qubit as a vector in the Bloch sphere, and then rotating this vector slowly away from its original position.
- **Relaxation**: this occurs when a qubit is in a more energetic state, or "excited" state, which we can call $|1\rangle$, and over time it slowly returns back to its "ground" state, which we term $|0\rangle$.

We also can describe some of the specific errors that can occur in a qubit, and which lead to the terms described above.

- **Bit flips**: a qubit flips from a 0 to a 1 or vice versa; for example, a state may go from $|000\rangle$ to $|010\rangle$ without the action of the quantum gate that would have done this normally.
- **Phase flips**: a qubit in a superposition state will experience a sign flip between the two components; for example, for a multi-qubit state, the state may go from $(|001\rangle + |110\rangle)$ to $(|001\rangle - |110\rangle)$. Again, this is not a command from the user, it is one that appears to spontaneously occur with no command to do so.

The two error sources above can be corrected for using more qubits to track which qubit's bit or phase flipped—this is called **error correction**. The two types of error control are **error correction** and **error mitigation**. Error mitigation is not exactly the same thing as error correction; it deals with current practical methods in reducing error in quantum circuits, but not completely removing these errors [1]. Error correction tries to do the latter, but often requires hundreds of qubits to perform this task well, which is just not practical at this time for large-scale use.

3.1.2 Noiseless vs. noisy circuits: a simple example

Now let's get our hands dirty and build a very simple circuit to demonstrate this noise. We'll use a noiseless simulator first, and then move to a noisy backend to show the difference between simulated quantum circuits and what would happen on a real quantum device. Let's build a couple of the Bell states to start.

```
#import qiskit and other needed packages
import qiskit
import qiskit.test.mock
import numpy as np
import matplotlib.pyplot as plt
```

The first Bell state we want to build is given as

$$\frac{|00\rangle + |11\rangle}{\sqrt{2}}$$

```
#build the first Bell state
bell1 = qiskit.QuantumCircuit(2)
bell1.h(0)
bell1.cx(0,1)
bell1.measure_all()
print(bell1)
```

```
     ┌───┐      ┌─┐
q_0: ┤ H ├──■───┤M├──────
     └───┘┌─┴─┐ └╥┘┌─┐
q_1: ─────┤ X ├──╫─┤M├──
          └───┘  ║ └╥┘
meas: 2/═════════╩══╩═══
                 0  1
```

The second Bell state we will build is

$$\frac{|10\rangle + |01\rangle}{\sqrt{2}}$$

```
#second Bell state
bell2 = qiskit.QuantumCircuit(2)
bell2.x(1)
bell2.h(0)
bell2.cx(0,1)
bell2.measure_all()
print(bell2)
```

```
     ┌───┐      ┌─┐
q_0: ┤ H ├──■───┤M├──────
     ├───┤┌─┴─┐ └╥┘┌─┐
q_1: ┤ X ├┤ X ├──╫─┤M├──
     └───┘└───┘  ║ └╥┘
meas: 2/═════════╩══╩═══
                 0  1
```

3.1.3 Noiseless measurement

Now let's do a simple measurement of these circuits. The measurement tool is already included in both circuits, so all we need to do is send these circuits to a noiseless simulator to begin.

```
#call the Aer noiseless simulator
shots = 2**13
sim = qiskit.Aer.get_backend('aer_simulator')
qc_trans = qiskit.transpile(bell1, sim)
counts = sim.run(qc_trans, shots=shots).result().get_counts()
print('Counts:', counts)
```

```
Counts: {'11': 4104, '00': 4088}
```

As this is a **very** simple measurement, and we do not even wish to get any probabilities here, only the number of counts for each state of the Bell state, we only see these counts above. However, it is obvious that there are only two states, $|00\rangle$ and $|11\rangle$, the ones we expected. Now let's check the same thing for the second Bell state.

```
#call the Aer noiseless simulator
qc_trans2 = qiskit.transpile(bell2, sim)
counts2 = sim.run(qc_trans2, shots=shots).result().get_counts()
print('Counts:', counts2)
```

```
Counts: {'10': 4176, '01': 4016}
```

As expected, only the $|10\rangle$ and $|01\rangle$ state have shown up. Now let's move on to the noisy measurement with a noisy backend and see what changes!

3.1.4 Noisy measurement

We employ the `qiskit.test.mock` module, which uses data from a real IBM quantum device to construct a noisy simulator that will emulate this real quantum computer [2].

```
#load qiskit noisy backend (with more than 1 qubit)
noisy_backend = qiskit.test.mock.FakeLondon()
print("Basis gates:", noisy_backend.configuration().basis_gates)
print("Number of qubits:", noisy_backend.configuration().num_qubits)
```

```
Basis gates: ['id', 'u1', 'u2', 'u3', 'cx']
Number of qubits: 5
```

Now let's run our circuits again, on this noisy simulator, and see what we get.

```
#first Bell state
qc_transn1 = qiskit.transpile(bell1, noisy_backend)
countsn1 = noisy_backend.run(qc_transn1, shots=shots).result().get_counts()
print('Noisy counts for (|00> + |11>):', countsn1)

#second Bell state
qc_transn2 = qiskit.transpile(bell2, noisy_backend)
countsn2 = noisy_backend.run(qc_transn2, shots=shots).result().get_counts()
print('Noisy counts for (|10> + |01>):', countsn2)
```

```
Noisy counts for (|00> + |11>): {'11': 3517, '00': 3960, '01': 396, '10': 319}
Noisy counts for (|10> + |01>): {'10': 3670, '11': 145, '01': 3794, '00': 583}
```

Aha! The counts from the noisy backend clearly show counts in states that are not supposed to be included in the Bell states they correspond to, demonstrating the effect of the noise in a real quantum circuit. This will definitely become an issue with longer and larger computations, and will be something we would like to mitigate. On the next page, we discuss zero-noise extrapolation, a method of error mitigation that we'll be using in future chapters to dispose of as much error as we can while running on real quantum devices.

3.1.5 References

- [1] mitiq documentation, "About Error Mitigation," 2020. (Page found here).
- [2] FRIB-TA Summer School: Quantum Computing and Nuclear Few- and Many-Body Problems, Lecture 11 (link here).

3.2 Error Mitigation: Zero-Noise Extrapolation

Author: Alexandra Semposki

We know that, when running quantum circuits on a real quantum device, there will be some noise in our system (this is an example of fault-tolerance, a theory that claims there will always be such noise in quantum hardware). However, can we **quantify** how much effect this noise produces in our circuit? Yes, we can! One method to do this is to use zero-noise extrapolation, or ZNE.

3.2.1 How it works

ZNE makes corrections to the expectation values of some observable calculated using quantum circuits. (In the VQE chapter, this will be the expectation value of the Hamiltonian to find the ground state energy of the atomic system in question.) To make these corrections properly, we can use knowledge of some underlying noise structure, or we can apply this technique without this knowledge. ZNE is able to handle either case well.

Let's first take a look at the basic theory. We start with the expectation value E of some observable that was measured in a noisy system. We can then expand this in a power series with respect to a noise parameter λ around the zero-noise value we will call E_0:

$$E(\lambda) \simeq E_0 + \sum_{k=1}^{n} a_k \lambda^k + O(\lambda^{n+1}),$$

where a_k are noise model-dependent coefficients, and λ is the noise parameter [1]. If we can, instead of decreasing the noise down to zero, increase it by n increments $\hat{E}(c_i\lambda)$, we can write a better estimate for the noiseless expectation value as

$$\hat{E}^n(\lambda) = \sum_{i=0}^{n} \gamma_i \hat{E}(c_i\lambda),$$

(where γ_i are solutions to conditions $\sum_i \gamma_i = 0$ and $\sum_i \gamma_i c_i^k = 0$ for each $k = 1, ...n$) and fit these n data points with a linear or exponential curve (depending on the shape of the points we get) and extrapolate to the zero-noise limit [1, 2]. This is a very similar approach to extrapolating from a finite temperature backwards to a zero-temperature limit, or extrapolating to $q^2 = 0$ in nuclear physics applications.

3.2.2 Looking ahead

In the next chapter on the Variational Quantum Eigensolver (VQE), we'll see ZNE applied to the $N = 2, 3$ cases of the ground state energy of the deuteron using the module for ZNE in the mitiq package, both for a simulator backend with a noise model, and a real IBM quantum device.

3.2.3 References

- [1] Kandala, A., Temme, K., Córcoles, A.D. et al. Error mitigation extends the computational reach of a noisy quantum processor. Nature 567, 491–495 (2019) (click here for a direct link).
- [2] mitiq documentation, "About Error Mitigation," 2020. (Page found here).

3.2.4 Zero-Noise Extrapolation: the N=2 case

Authors: Kyle Godbey, Alexandra Semposki

Maintainer: Alexandra Semposki

Here we'll try developing some error mitigation for the $N = 2$ deuteron case from the VQE chapter. Let's begin by importing pennylane, mitiq, and any other packages we may need, and copy over the code from the $N = 2$ notebook for calculating the ground state energy.

```
%matplotlib inline

import matplotlib.pyplot as plt
from pennylane import numpy as np
import pennylane as qml
import warnings
warnings.filterwarnings('ignore')

import mitiq as mq
from mitiq.zne.scaling import fold_global
from mitiq.zne.inference import RichardsonFactory
from pennylane.transforms import mitigate_with_zne
from qiskit.providers.aer import AerSimulator
import qiskit.providers.aer.noise as noise

plt.style.use(['science','notebook'])
```

VQE for N=2 (with noisy simulator)

Here we'll pull from some of the VQE notebook to define the Hamiltonian and circuit again. If you would like to revisit the VQE process, go to this page.

Next, we define a simulator that has a noise model imbedded into it, so that we are simulating running on a real quantum device. Let's see how the convergence of the VQE routine goes.

```
#define the Hamiltonian we're using again, for N=2
coeffs = [5.906709, 0.218291, -6.125, -2.143304, -2.143304]
obs = [qml.Identity(0), qml.PauliZ(0), qml.PauliZ(1), qml.PauliX(0) @ qml.PauliX(1),␣
 ↪qml.PauliY(0) @ qml.PauliY(1)]

H = qml.Hamiltonian(coeffs, obs)

print(H)
```

```
  (-6.125) [Z1]
+ (0.218291) [Z0]
+ (5.906709) [I0]
```

(continues on next page)

(continued from previous page)

```
+ (-2.143304) [X0 X1]
+ (-2.143304) [Y0 Y1]
```

```
#define some noise model to use in pennylane from qiskit.test.mock
from qiskit.test.mock import FakeLima
from qiskit.providers.aer.noise import NoiseModel

backend = FakeLima()
noise_model = NoiseModel.from_backend(backend)

#set up noisy device
dev_sim = qml.device("qiskit.aer", wires=2, noise_model=noise_model, optimization_
 ↪level=0, shots=10000)

#now we'll define the circuit we want to use with this Hamiltonian
def circuit(params, wires):
    t0 = params[0]
    qml.PauliX(wires=0)
    qml.RY(t0, wires=1)
    qml.CNOT(wires=[1,0])
   # return qml.expval(H)

#define cost function
cost_fn = qml.ExpvalCost(circuit, H, dev_sim)

#set up the Qnode to run the above circuit on the simulator (is this necessary?)
sim_qnode = qml.QNode(circuit, dev_sim)
```

Now we define some initial parameter and convergence info, as before.

```
#parameter array
init_params = np.array([2.5,])

#convergence information and step size
max_iterations = 80
conv_tol = 1e-06
step_size = 0.01
```

Let's run VQE as in the VQE chapter. We should get a result that takes longer to converge due to the noise we have added into the device.

```
#VQE step
opt = qml.GradientDescentOptimizer(stepsize=step_size)

params = init_params

gd_param_history = [params]
gd_cost_history = []

for n in range(max_iterations):

    # Take a step in parameter space and record your energy
    params, prev_energy = opt.step_and_cost(cost_fn, params)

    # This keeps track of our energy for plotting at comparisons
```

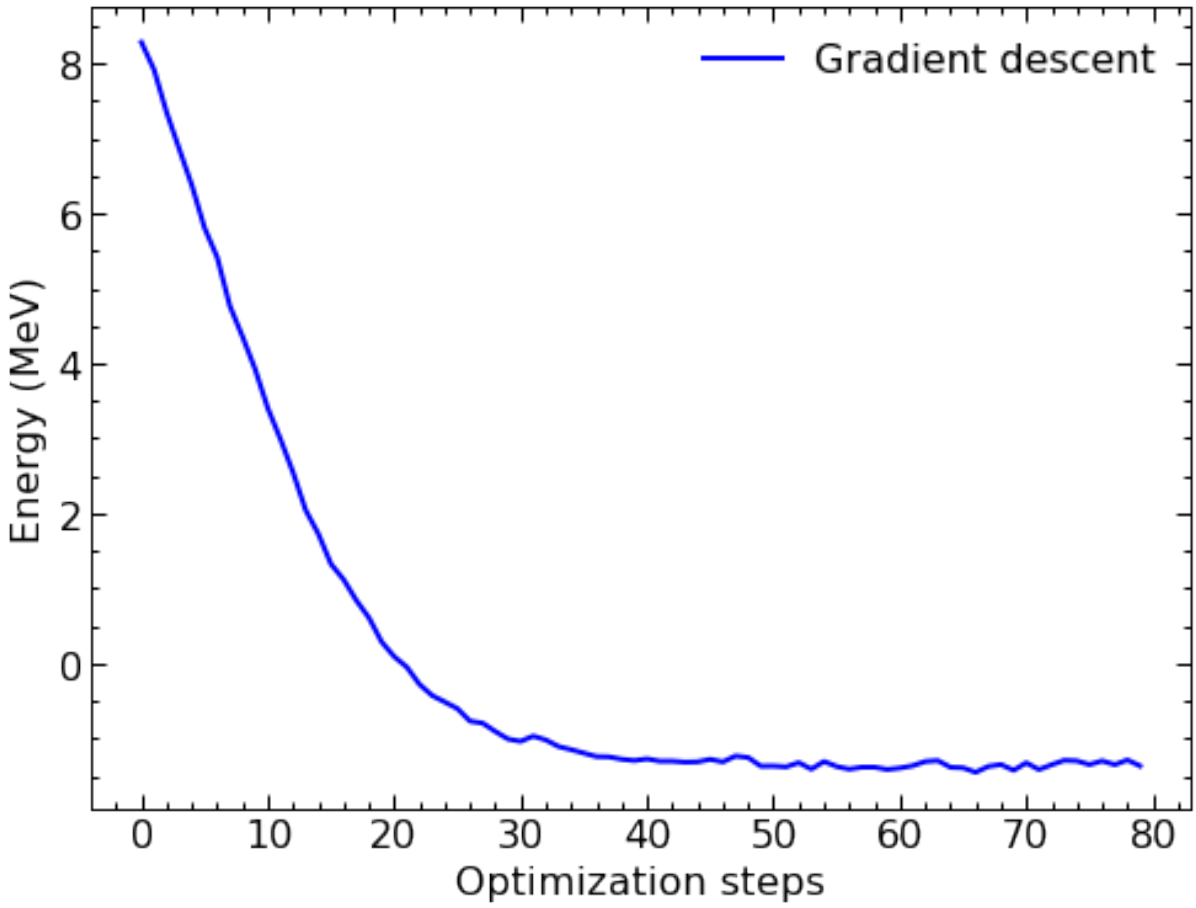
(continues on next page)

(continued from previous page)

```
    gd_param_history.append(params)
    gd_cost_history.append(prev_energy)

    # Here we see what the energy of our system is with the new parameters
    energy = cost_fn(params)

    # Calculate difference between new and old energies
    conv = np.abs(energy - prev_energy)

    if n % 20 == 0:
        print(
            "Iteration = {:},  Energy = {:.8f} MeV,  Convergence parameter = {"
            ":.8f} MeV".format(n, energy, conv)
        )

    if conv <= conv_tol:
        break

print()
print("Final value of the energy = {:.8f} MeV".format(energy))
print("Number of iterations = ", n)
```

```
Iteration = 0,  Energy = 7.84977776 MeV,  Convergence parameter = 0.43411675 MeV
Iteration = 20,  Energy = -0.04717536 MeV,  Convergence parameter = 0.13790896 MeV
Iteration = 40,  Energy = -1.28578824 MeV,  Convergence parameter = 0.01283113 MeV
Iteration = 60,  Energy = -1.35220863 MeV,  Convergence parameter = 0.04286391 MeV

Final value of the energy = -1.37330145 MeV
Number of iterations =  79
```

```
plt.plot(gd_cost_history, "b", label="Gradient descent")

plt.ylabel("Energy (MeV)")
plt.xlabel("Optimization steps")
plt.legend()
plt.show()
```



The noise here is quite evident in the plot above from the VQE routine results. Let's see if we can implement ZNE to take care of this as we iterate! We'll copy the cells above into the ones below so we can keep the results from above with the noisy simulator for comparison to the results after ZNE is used.

ZNE for N=2

```
#set up the extrapolation step and scale factors to use
extrapolate = RichardsonFactory.extrapolate
scale_factors = [1, 2, 3, 4]

def circuit(params, wires):
    t0 = params[0]
    qml.PauliX(wires=0)
    qml.RY(t0, wires=1)
    qml.CNOT(wires=[1,0])
    return qml.expval(H)

sim_qnode = qml.QNode(circuit, dev_sim)

#use ZNE to mitigate error
mitigated_qnode = mitigate_with_zne(scale_factors, fold_global, extrapolate)(sim_
 ↪qnode)
zne_result = mitigated_qnode(params,wires=2, shots=2**14)
```

(continues on next page)

(continued from previous page)

```
#print the result of the ground state energy
print('N=2 ground state energy result with ZNE: {} MeV'.format(zne_result))
```

```
N=2 ground state energy result with ZNE: -1.7711602082519509 MeV
```

This result is pretty close to the expected -1.75 MeV value for the N=2 ground state of the deuteron! However, it is clear from the above cell that the ZNE method was only applied on the circuit expectation value result and not the result from the VQE run. The latter is much more complicated to implement, and will be left as an exercise for the reader. See if you can nest ZNE into the VQE code above to apply ZNE at each step of the VQE routine! This should greatly improve our results from the VQE iterations.

We'll mention it in the hardware section as well, but some VQE implementations in various quantum computing libraries will include options for error mitigation baked-in, which makes getting up and running a lot easier than building our own.

3.3 Real Hardware: N=2 Case

Authors: Kyle Godbey, Alexandra Semposki

Maintainer: Kyle Godbey

Here we'll try running $N = 2$ deuteron case from the VQE chapter on real quantum hardware. This will, of course, be noisy just as the last chapter, but now our noise model is a little more true to life!

```
%matplotlib inline

import matplotlib.pyplot as plt
from pennylane import numpy as np
import pennylane as qml
import warnings
warnings.filterwarnings('ignore')

import mitiq as mq
from mitiq.zne.scaling import fold_global
from mitiq.zne.inference import RichardsonFactory
from pennylane.transforms import mitigate_with_zne
from qiskit.providers.aer import AerSimulator
import qiskit.providers.aer.noise as noise
from qiskit import IBMQ
from qiskit.providers.ibmq import RunnerResult
from pennylane_qiskit import upload_vqe_runner, vqe_runner
```

3.3.1 VQE for N=2 (on IBM Nairobi)

Here we'll pull from some of the VQE notebook to define the Hamiltonian and circuit again. If you would like to revisit the VQE process, go to *this page*.

```
#define the Hamiltonian we're using again, for N=2
coeffs = [5.906709, 0.218291, -6.125, -2.143304, -2.143304]
obs = [qml.Identity(0), qml.PauliZ(0), qml.PauliZ(1), qml.PauliX(0) @ qml.PauliX(1),␣
 ↪qml.PauliY(0) @ qml.PauliY(1)]
```

(continues on next page)

(continued from previous page)

```
H = qml.Hamiltonian(coeffs, obs)

print(H)
```

```
  (-6.125) [Z1]
+ (0.218291) [Z0]
+ (5.906709) [I0]
+ (-2.143304) [X0 X1]
+ (-2.143304) [Y0 Y1]
```

Now we'll set up our qiskit runtime job. For this, be sure to add your own IBMQ API token in `token` to authenticate with the IBMQ system.

```
token = "XXX"

IBMQ.enable_account(token)

program_id = upload_vqe_runner(hub="ibm-q", group="open", project="main")
```

Now we define some initial parameter and convergence info, as before, but we also need to define our job that wil be submitted to the qiskit runtime.

```
#now we'll define the circuit we want to use with this Hamiltonian
def circuit(params):
    t0 = params
    qml.PauliX(wires=0)
    qml.RY(t0, wires=1)
    qml.CNOT(wires=[1,0])
   # return qml.expval(H)

#parameter array
init_params = np.array([2.5,])

#convergence information and step size
max_iterations = 120
conv_tol = 1e-06
step_size = 0.01
shots = 10000

job = vqe_runner(
    program_id=program_id,
    backend="ibm_nairobi",
    hamiltonian=H,
    ansatz=circuit,
    x0=init_params,
    shots=shots,
    optimizer="SPSA",
    optimizer_config={"maxiter": 50},
    kwargs={"hub": "ibm-q", "group": "open", "project": "main"},
)
```

```
# Get runtime job result.
result = job.result()
```

(continues on next page)

(continued from previous page)

```
print(result)
```

```
     fun: -0.7380420742000001
 message: 'Optimization terminated successfully.'
    nfev: 150
     nit: 50
 success: True
       x: array([2.70905085])
```

Phew, that took a long time! But that's the curse of running on real hardware with other real humans interested in running their circuits too. The value returned is also not great.. let's see if we can improve that!

3.3.2 ZNE for N=2

```
#set up the extrapolation step and scale factors to use
extrapolate = RichardsonFactory.extrapolate
scale_factors = [1, 2, 3, 4]

params = result['x']
unmitig_energy = result['fun']

dev_ibm = qml.device('qiskit.ibmq', wires=2, backend='ibm_nairobi', ibmqx_token=token)

def circuit(params, wires):
    t0 = params[0]
    qml.PauliX(wires=0)
    qml.RY(t0, wires=1)
    qml.CNOT(wires=[1,0])
    return qml.expval(H)

ibm_qnode = qml.QNode(circuit, dev_ibm)

#use ZNE to mitigate error
mitigated_qnode = mitigate_with_zne(scale_factors, fold_global, extrapolate)(ibm_
 ↪qnode)
zne_result = mitigated_qnode(params,wires=2, shots=2**14)

#print the result of the ground state energy
print('N=2 ground state energy result with ZNE: {} MeV'.format(zne_result))
```

```
N=2 ground state energy result with ZNE: -1.4298696529541 MeV
```

The result is still pretty bad… but again, that's one side effect of the current generation of hardware and our implementation on it. To do this better you could enable error mitigation during the VQE iteration, at the cost of additional time. That can be left as an exercise for the reader ;)

CHAPTER

FOUR

DIMENSIONALITY REDUCTION

As physicists, we are usually trying to find the important degrees of freedom of the system we want to study. If we consider too few, our theories become too simple and can't describe what we see. If we include too many, then our calculations become impossible to perform (perhaps even on lightning fast quantum computers!). Finding the right degrees of freedom to consider has usually been more of an art than a science, and it is a feature that many successful theories have shared over the centuries. Dimensionality reduction techniques - a series of novel approaches mostly developed in the last decades thanks to the increase in classical computational power - have transformed many areas of science by allowing us to select the degrees of freedom that are 'just right'.

In this chapter we will explore one such dimensionality reduction technique called the Reduced Basis Method, which comes under the umbrella of Reduced Order Models. This reduction will enable us to tackle a simple and a not-so-simple problem on a quantum computer using just a small number of qubits. We start first with the simple one: the Quantum Harmonic Oscillator.

4.1 Quantum Harmonic Oscillator as a two level system

Author(s): Pablo Giuliani, Kyle Godbey

Maintainer: Pablo Giuliani

Adapted from https://kylegodbey.github.io/nuclear-rbm/ho/1dHO.html with permision of Eric Flynn

4.1.1 Background

Let us consider the Schrodinger equation for the 1-d Harmonic Oscillator:

$$\Big(-\frac{d^2}{dx^2} + V_\alpha(x) - \lambda \Big)\phi = 0, \tag{4.1}$$

$$\tag{4.2}$$

where $V_\alpha(x) = \alpha x^2$. This equation can be written a

$$F_\alpha\Big[\phi_\alpha(x)\Big] = H_\alpha\phi - \lambda\phi = 0 \tag{4.3}$$

where $\lambda = 2mE$ is the eigenvalue with energy E and $\alpha = m^2\omega^2$ is the harmonic oscilation parameter with oscilation frequency ω.

The shooting method and finite elements methods are among the many approaches to solve such types of equations using classical computers. During this notebook we will follow the finite element method, in which every object is projected onto a discretized space, the functions $\phi(x)$ becomes vectors, and the operators H_α matrices. These vectors and matrices could be composed of houndreds or thousands of elements, making it impossible to store such amount of information in a quantum computer with few qbits. For example, 7 qbits are available in the case of the free version of the IBM services.

Therefore, if we want to implement an algorithm in a quantum circuit to find solutions to this equation for many values of the parameter α we need another strategy. We approximate this equation using dimensionality reduction techniques, such as the Reduced Basis Method, as was done in this book. These techniques aim at reducing the redundant, or unnecessary, degrees of freedom of the system from many thousand to just a handful. This is done by first restricting the solution to the equation to lay in the span of a basis of n states:

$$\hat{\phi}_\alpha(x) = \sum_{i=1}^{n} a_i \phi_i(x) \tag{4.4}$$

where $\phi_i(x)$ are informed on previous solutions built on a classical computer to the Schrodinger equation for chosen values of α.

4.1.2 Finite element method approach

```
import numpy as np
import scipy as sci
from scipy import optimize
from scipy import special
import matplotlib.pyplot as plt
plt.style.use(['science','notebook'])
```

We first define some helper functions to return things like the potentials and Hamiltonian.

```
### NOTE: hbar = 1 in this demo
### First define exact solutions to compare numerical solutions to.
def V(x,alpha):
    '''
    1-d harmonic Oscillator potential

    Parameters
    ----------
    x : float or nd array
        position.
    alpha : float
        oscillator length parameter.

    Returns
    -------
    float or ndarray
        value of potential evaluated at x.

    '''
    return alpha*x**2
def construct_H(V,grid,mass,alpha):
    '''
    Uses 2nd order finite difference scheme to construct a discretized differential H␣
 ↪operator
    Note: mass is fixed to 1 for this demo.

    Parameters
    ----------
    V : TYPE
        DESCRIPTION.

    alpha : TYPE
```

(continues on next page)

(continued from previous page)

```
        oscillator parameter alpha used in V(x) = alpha*x**2.

    Returns
    -------
    H : ndarray
        2d numpy array
    '''
    dim = len(grid)
    off_diag = np.zeros(dim)
    off_diag[1] = 1
    H = -1*(-2*np.identity(dim) + sci.linalg.toeplitz(off_diag))/(mass*h**2) + np.
↪diag(V(grid,alpha))
    return H
def solve(H,grid,h):
    '''
    Parameters
    ----------
    H : 2d ndarray
        Hamiltonian Matrix.
    grid : ndarray
        Discretized 1d domain.
    h : float
        grid spacing.

    Returns
    -------
    evals : ndarray
        returns nd array of eigenvalues of H.
    evects : ndarray
        returns ndarray of eigenvectors of H.
    Eigenvalues and eigenvectors are ordered in ascending order.
    '''
    evals,evects = np.linalg.eigh(H)
    evects = evects.T
    for i,evect in enumerate(evects):
        #norm = np.sqrt(1/sci.integrate.simpson(evect*evect,grid))
        norm = 1/(np.linalg.norm(np.dot(evect,evect)))
        evects[i] = evects[i]*norm/np.sqrt(h)
    return evals,evects

def getSystem(H,psi_array,phi_array):
    '''
    Sets up system of equations for Galerkin projection <\phi_{j},F_{k}(phi_{k})>

    Parameters
    ----------
    H : ndarray
        Hamiltonian matrix.
    psi_array : ndarray
        array of projector functions. Assumes rows are corresponding to discretized␣
↪functions
    phi_array : TYPE
        DESCRIPTION.

    Returns
    -------
```

(continues on next page)

(continued from previous page)

```
    Function
        Function that takes in a vector of parameters of the form (a_{1},a_{2},....,\
↪lambda_{\alpha})
        and outputs of the system of equations. \lambda_{\alpha} is the eigenvalue of␣
↪HO equation.
    '''
    ##
    def system(a_vec):
        '''
        Parameters
        ----------
        a_vec : ndarray
            vector of parameters (a_{k},\lambda).

        Returns
        -------
        results : ndarray
            outputs of the Galerkin projection equations and normalization functions
↪<\hat{\phi}|\hat{\phi}> = 1.
            This systems can be solved using sci.optimize.fsolve or using your choice␣
↪of method.
        '''
        results = np.zeros(len(a_vec))
        results[-1] = -1 # from normalization
        for i in np.arange(0,len(a_vec)-1,1):
            for k in np.arange(0,len(a_vec)-1,1):
                results[i] += a_vec[k]*np.dot(psi_array[i],np.matmul(H,phi_array[k]))␣
↪- \
                a_vec[-1]*a_vec[k]*np.dot(psi_array[i],phi_array[k])
        for k in np.arange(0,len(a_vec)-1,1):
            for kp in np.arange(0,len(a_vec)-1,1):
                results[-1] += a_vec[kp]*a_vec[k]*np.dot(psi_array[kp],phi_array[k])
        return results
    return system
```

Now we can define some parameters of our system, including our mesh.

```
#First define global variables
h = 10**(-1) ### grid spacing for domain (Warning around 10**(-3) it starts to get␣
 ↪slow).
### HO global parameters
n = 0 # principle quantum number to solve in HO
mass = 1.0 # mass for the HO system
# define the domain boundaries
x_a = -10 # left boundary
x_b = 10 # right boundary
x_array = np.arange(x_a,x_b+h,h)
m = len(x_array)
print('Number of grid points: ',m)
```

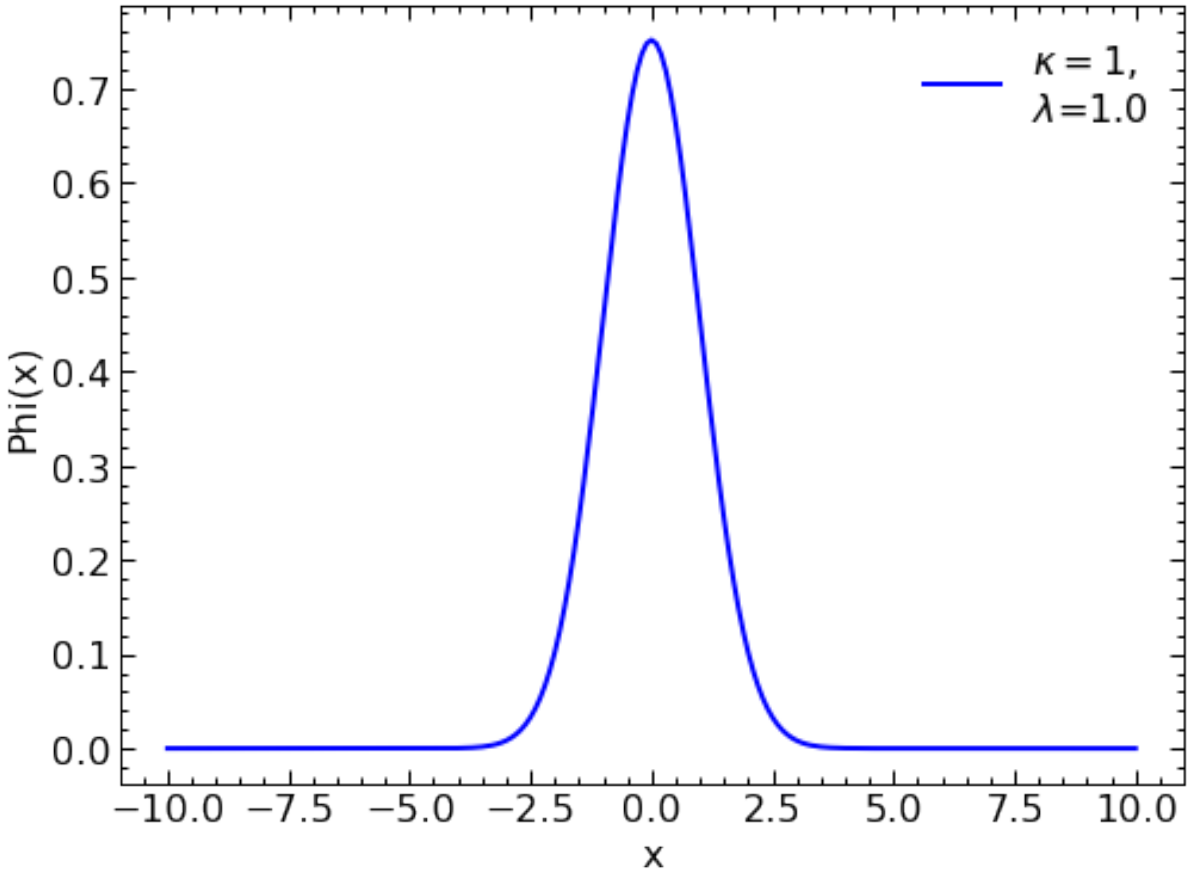
```
Number of grid points:  201
```

```
H = construct_H(V,x_array,1,1)
evals, evects = solve(H,x_array,h)
```

(continues on next page)

(continued from previous page)

```
plt.plot(x_array, evects[0], color="blue",label = r'$\kappa=1,$' "\n" '$\lambda$=
↪'+str(round(evals[0],2)))

plt.legend(loc='upper right')

plt.ylabel("Phi(x)")
plt.xlabel("x")
plt.show()
```



4.1.3 The Reduced Basis Method approach

Construction of the Reduced Basis

Now we select a set of values of α to solve with the finite element method. These solutions will be used as the basis in the reduced basis model.

```
# Select alpha values to use to solve SE exactly.
alpha_vals = [.5,1,2,5,15]  #Here, we choose 3 values of alpha to solve exactly. This␣
↪results in 3 basis functions
# initialize solution arrays. T is the matrix that will hold wavefunction solutions.
# T has the form T[i][j], i = alpha, j = solution components
T = np.zeros((len(alpha_vals),m))
```

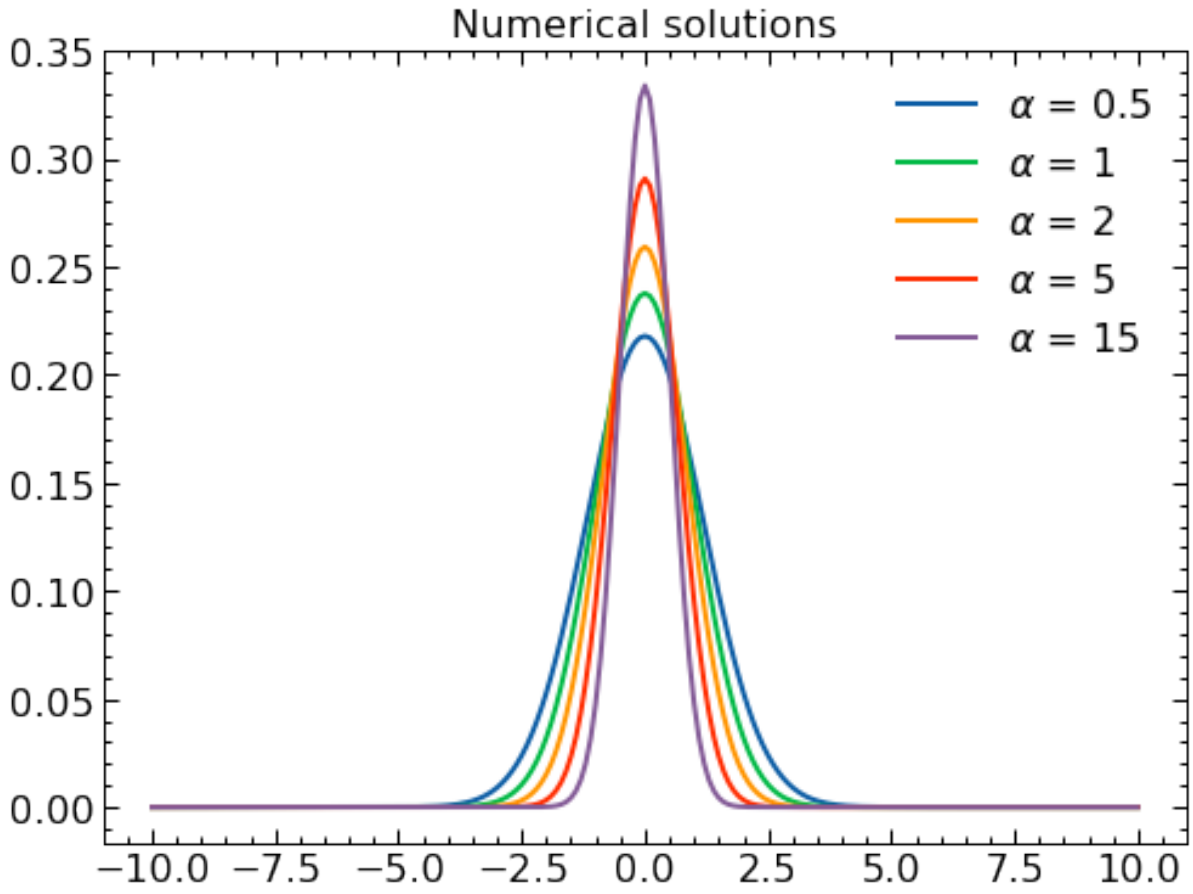
(continues on next page)

(continued from previous page)

```
# T_evals holds the eigenvalues for each evect in T.
T_evals = np.zeros(len(alpha_vals))
```

```
for i,alpha_sample in enumerate(alpha_vals):
    H = construct_H(V,x_array,mass,alpha_sample) # construct the Hamiltonian matrix␣
 ↪for given alpha_sample.
    evals, evects = solve(H,x_array,h) # solve the system for evals and evects.
    T[i] = evects[n]/np.linalg.norm(evects[n]) # assign the nth evect to solution␣
 ↪array T
    T_evals[i] = evals[n] # assign the nth eigenvalue to the eigenvalue array T_eval.
    print(f'alpha = {alpha_sample}, lambda = {evals[n]}')



# Make plots of the numerical wavefunction
fig, ax = plt.subplots(1,1)
for i in range(len(alpha_vals)):

    ax.plot(x_array,(T[i]),label=r'$\alpha$ = '+str(alpha_vals[i]))

ax.set_title('Numerical solutions')

ax.legend()

plt.show()
```

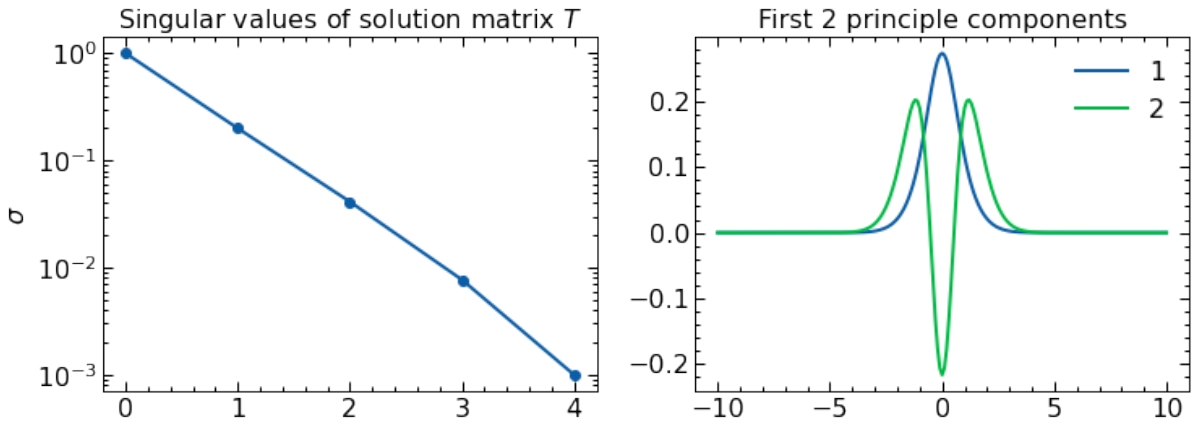
```
alpha = 0.5, lambda = 0.7067941428963106
alpha = 1, lambda = 0.9993746086404902
alpha = 2, lambda = 1.4129624545775548
alpha = 5, lambda = 2.23293859175503
alpha = 15, lambda = 3.8635854863645682
```



Proper Orthogonal Decomposition

After obtained a sample of what the solutions look like for some values of the parameters we proceed now to construct our basis with two elements. These elements will be selected as the first two principal components of a Principal Component Analysis, also related to Singular Value Decomposition. This procedure aims at capturing the two components that can better explain (in a linear way) the variability among our samples. In the Reduced Basis Methods community it is known as Proper Orthogonal Decompositon.

```
U, s, Vh = np.linalg.svd(T)
phi1=-1*Vh[0] #multiplying by -1 so they are positive at x=0, not necessary but the␣
 ↪plots look better
phi2=-1*Vh[1]
n_comps = 2 # number of principle components to plot (i.e number of column vectors of␣
 ↪SVD matrix V to plot)
fig, ax = plt.subplots(1,2,figsize=(12,4))
ax[0].semilogy(s/s[0],'o-')
ax[0].set_title(r'Singular values of solution matrix $T$')
ax[0].set_ylabel(r'$\sigma$')

ax[1].plot(x_array,phi1,label=str(1))
ax[1].plot(x_array,phi2,label=str(2))
ax[1].set_title(f'First {n_comps} principle components')
ax[1].legend()
plt.show()
```



Effective two level system "Hamiltonian"

We will now model an effective hamiltonian in which two "particles" interact, each one represents one of these two components.

Here we explicitly construct $\hat{\phi}_\alpha = a_1\phi_1 + a_2\phi_2$ and use the Galerkin method, that is, projecting $F_\alpha(\hat{\phi}_\alpha(x))$ over 2 linearly independent functions projecting functions $\{\psi_i\}_{i=1}^2$.

$$\langle \psi 1 | F_\alpha(\hat{\phi}_\alpha(x))\rangle = 0 \tag{4.5}$$

$$\langle \psi 2 | F_\alpha(\hat{\phi}_\alpha(x))\rangle = 0 \tag{4.6}$$

We can interpret this as enforcing the orthogonality of $F_\alpha(\hat{\phi}_\alpha(x))$ to the subspace spanned by $\{\psi_i\}$ that is, by finding $\hat{\phi}_\alpha$ such that $F_\alpha(\hat{\phi}_\alpha)$ is approximately zero up to the ability of the set $\{\psi_i\}$. The choice of projecting functions $\{\psi_i\}$ is arbitrary, but here we choose the solution set $\{\phi_i\}$ to be our projecting functions to make our lives easier. Since λ is also unknown, we need an additional equation. This comes from the normalization conditions:

$$\langle \hat{\phi}_{\alpha_k} | \hat{\phi}_{\alpha_k} \rangle = 1 \tag{4.7}$$

We can re-write the projecting equations taking advantage of the linear form of F_α to obtain an effective 2-level system Hamiltonian. We note that:

$$\langle \phi_i | F_\alpha(\hat{\phi}_\alpha(x))\rangle = \tag{4.8}$$

$$\langle \phi_i | a_1 H_\alpha \phi_1 + a_2 H_\alpha \phi_2 - a_1\hat{\lambda}\phi_1 - a_2\hat{\lambda}\phi_2\rangle \tag{4.9}$$

Since $\langle \phi_i | \phi_j \rangle = \delta_{i,j}$, we arrive at the following matrix equation for the projecting equations:

$$\tilde{H}_\alpha |a\rangle = \hat{\lambda}|a\rangle \tag{4.10}$$

where $|a\rangle = \{a_1, a_2\}$ and

$$\tilde{H}_\alpha = \begin{bmatrix} \langle\phi_1|H_\alpha|\phi_1\rangle & \langle\phi_1|H_\alpha|\phi_2\rangle \\ \langle\phi_2|H_\alpha|\phi_1\rangle & \langle\phi_2|H_\alpha|\phi_2\rangle \end{bmatrix} \tag{4.11}$$

while the normalization condition translates into:

$$\langle a|a\rangle = a_1^2 + a_2^2 = 1 \tag{4.12}$$

Now we proceed to construct this Hamiltonian matrix for our problem at hand. We note that although $\tilde{H}_\alpha$ depends on α, the dependance is only in the potential part and it is affine (linear): $H_\alpha = H_0 + \alpha H_1$, where $H_0 = -\frac{d^2}{dx^2}$ and $H_1 = x^2$. We then decompose $\tilde{H}_\alpha = \tilde{H}_0 + \alpha\tilde{H}_1$. We now construct these matrices:

```
dim0 = len(x_array)
off_diag = np.zeros(dim0)
off_diag[1] = 1

H0=-1*(-2*np.identity(dim0) + sci.linalg.toeplitz(off_diag))/(mass*h**2)
H1=np.diag(V(x_array,1))

tildeH0=np.array([[0.0,0.0],[0.0,0.0]])
tildeH1=np.array([[0.0,0.0],[0.0,0.0]])

tildeH0[0][0]=np.dot(phi1,np.dot(H0,phi1))
tildeH0[0][1]=np.dot(phi1,np.dot(H0,phi2))
tildeH0[1][0]=np.dot(phi2,np.dot(H0,phi1))
tildeH0[1][1]=np.dot(phi2,np.dot(H0,phi2))

tildeH1[0][0]=np.dot(phi1,np.dot(H1,phi1))
tildeH1[0][1]=np.dot(phi1,np.dot(H1,phi2))
tildeH1[1][0]=np.dot(phi2,np.dot(H1,phi1))
tildeH1[1][1]=np.dot(phi2,np.dot(H1,phi2))

print(tildeH0)
print(tildeH1)
```

```
[[ 0.7770743  -1.240322  ]
 [-1.240322    4.30501452]]
[[0.33343621 0.51501739]
 [0.51501739 1.49714625]]
```

With these two Hamiltonians, we can then combine them according to the equation above via these helper functions.

```
def tildeH(alpha):
    return tildeH0+alpha*tildeH1

def systemSolver(alpha):
    resultssystem= np.linalg.eig(tildeH(alpha))
    if resultssystem[1][0][0]<0:
        resultssystem[1][0]=resultssystem[1][0]*(-1)
    if resultssystem[1][1][0]<0:
        resultssystem[1][1]=resultssystem[1][1]*(-1)
    return resultssystem

def phibuilder(alpha):
    coefficients=systemSolver(alpha)[1][0]
    return coefficients[0]*phi1+coefficients[1]*phi2
```

Let's print out our total Hamiltonian given $\alpha = 3$

```
tildeH(3)
```

```
array([[1.77738292, 0.30473019],
       [0.30473019, 8.79645327]])
```

Looks great! Now we can compute the coefficients of our basis functions via direct diagonalization, again with $\alpha = 3$.

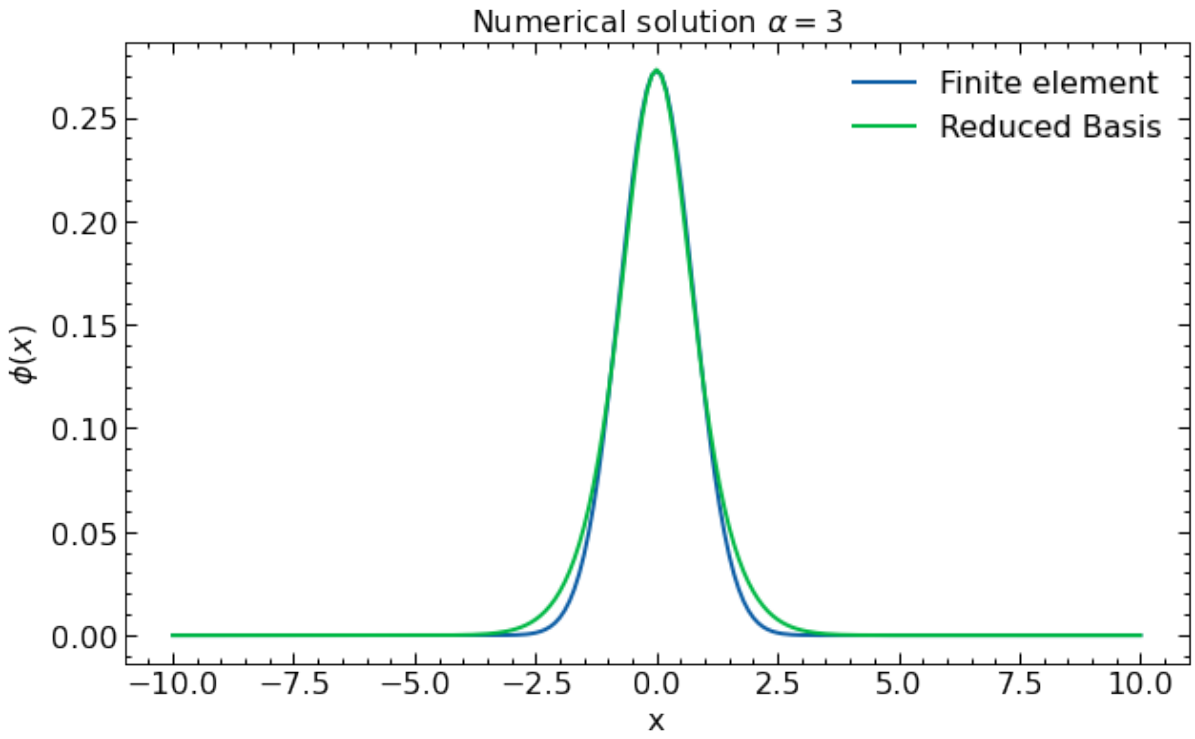
```
solution = systemSolver(3)
print("Eigenvalues: ", solution[0])
print("a1 Coefficient: ",solution[1][0][0])
print("a2 Coefficient: ",solution[1][0][1])
```

```
Eigenvalues:  [1.76417802 8.80965817]
a1 Coefficient:  0.9990624419940102
a2 Coefficient:  0.04329245889257087
```

Now we try our solution solver against the finite element computation to see how well our RBM solution is doing.

```
# Now we can construct our RBM for an alpha of our choosing.
alpha_k = 3
solGaler=phibuilder(alpha_k)
lamGaler=systemSolver(alpha_k)[0][0]

alpha_vals = [alpha_k]

T = np.zeros((len(alpha_vals),m))

T_evals = np.zeros(len(alpha_vals))

for i,alpha_sample in enumerate(alpha_vals):
    H = construct_H(V,x_array,mass,alpha_sample) # construct the Hamiltonian matrix␣
↪for given alpha_sample.
    evals, evects = solve(H,x_array,h) # solve the system for evals and evects.
    T[i] = evects[n]/np.linalg.norm(evects[n]) # assign the nth evect to solution␣
↪array T
    T_evals[i] = evals[n] # assign the nth eigenvalue to the eigenvalue array T_eval.
    print(f'alpha = {alpha_sample}, lambda = {evals[n]}')



# Make plots of the numerical wavefunction
fig, ax = plt.subplots(1,1,figsize=(10,6))
for i in range(len(alpha_vals)):

    ax.plot(x_array,(T[i]),label= r'Finite element')

ax.plot(x_array,(phi1),label=r'Reduced Basis')



ax.set_title('Numerical solution ' r'$\alpha=$'+str(alpha_vals[i]))

ax.legend()
plt.ylabel(r'$\phi(x)$')
plt.xlabel("x")
ax.legend()
plt.show()
```

```
alpha = 3, lambda = 1.7301737711983323
```



Excellent! Of course, if we take more basis functions we will better reproduce the finite element solution, but the procedure remains largely the same. Now that we've laid the groundwork for representing our complicated coordinate space problems in a natural reduced basis, we can explore how this translates to a representation on a quantum computer.

Before that, we make a final comment in regards to approximations such as the reduced basis method. We are using a basis of two elements ϕ_1 and ϕ_2 in which we believe ϕ_1 captures most of the dynamics of the system as α changes, while ϕ_2 accounts for small corrections. This is due to how we built the basis throught the Principal Component Analysis. A great advantage of such procedure is that, if the coefficient associated with ϕ_2 starts becoming large, that will signal that our approximation is deteriorating and that more basis states would be needed for a correct representation of the underlying system. Give it a try, can you choose a parameter α way beyond what we have been using and see what happens with the coefficients a_1 and a_2?

4.2 Reduced Basis Method on a Quantum Computer

Author(s): Kyle Godbey, Pablo Giuliani

Maintainer: Kyle Godbey

Now it's time to take the two level representation we just solved classically and implement it for use on quantum hardware. The bulk of this notebook is based on the VQE example for the deuteron, so check that out for more descriptions *here*.

```
%matplotlib inline

import matplotlib.pyplot as plt
from pennylane import numpy as np
import scipy as sci
import pennylane as qml
from qiskit_nature.operators.second_quantization import FermionicOp
from qiskit_nature.mappers.second_quantization import JordanWignerMapper, ParityMapper
import time
```

(continues on next page)

(continued from previous page)

```
from functools import partial
import warnings
warnings.filterwarnings('ignore')
plt.style.use(['science','notebook'])
```

We'll also need some helper functions to parse Pauli strings and construct our pennylane Hamiltonian, so I'll just dump them here along with the helpers from the previous example.

```
def pauli_token_to_operator(token):
    qubit_terms = []

    for term in range(len(token)):
        # Special case of identity
        if token[term] == "I":
            pass
        else:
            #pauli, qubit_idx = term, term
            if token[term] == "X":
                qubit_terms.append(qml.PauliX(int(term)))
            elif token[term] == "Y":
                qubit_terms.append(qml.PauliY(int(term)))
            elif token[term] == "Z":
                qubit_terms.append(qml.PauliZ(int(term)))
            else:
                print("Invalid input.")
    if(qubit_terms==[]):
            qubit_terms.append(qml.Identity(0))
    full_term = qubit_terms[0]
    for term in qubit_terms[1:]:
        full_term = full_term @ term

    return full_term


def parse_hamiltonian_input(input_data):
    # Get the input
    coeffs = []
    pauli_terms = []
    chunks = input_data.split("\n")
    # Go through line by line and build up the Hamiltonian
    for line in chunks:
        #line = line.strip()
        tokens = line.split(" ")
        # Parse coefficients
        sign, value = tokens[0][0], tokens[1]

        coeff = float(value)
        if sign == "-":
            coeff *= -1
        coeffs.append(coeff)

        # Parse Pauli component
        pauli = tokens[3]

        pauli_terms.append(pauli_token_to_operator(pauli))
```

(continues on next page)

(continued from previous page)

```
    return qml.Hamiltonian(coeffs, pauli_terms)

### NOTE: hbar = 1 in this demo
### First define exact solutions to compare numerical solutions to.
def V(x,alpha):
    '''
    1-d harmonic Oscillator potential

    Parameters
    ----------
    x : float or nd array
        position.
    alpha : float
        oscillator length parameter.

    Returns
    -------
    float or ndarray
        value of potential evaluated at x.

    '''
    return alpha*x**2
def construct_H(V,grid,mass,alpha):
    '''
    Uses 2nd order finite difference scheme to construct a discretized differential H␣
 ↪operator
    Note: mass is fixed to 1 for this demo.

    Parameters
    ----------
    V : TYPE
        DESCRIPTION.

    alpha : TYPE
        oscillator parameter alpha used in V(x) = alpha*x**2.

    Returns
    -------
    H : ndarray
        2d numpy array
    '''
    dim = len(grid)
    off_diag = np.zeros(dim)
    off_diag[1] = 1
    H = -1*(-2*np.identity(dim) + sci.linalg.toeplitz(off_diag))/(mass*h**2) + np.
 ↪diag(V(grid,alpha))
    return H
def solve(H,grid,h):
    '''
    Parameters
    ----------
    H : 2d ndarray
        Hamiltonian Matrix.
    grid : ndarray
        Discretized 1d domain.
    h : float
```

(continues on next page)

(continued from previous page)

```
        grid spacing.

    Returns
    -------
    evals : ndarray
        returns nd array of eigenvalues of H.
    evects : ndarray
        returns ndarray of eigenvectors of H.
    Eigenvalues and eigenvectors are ordered in ascending order.
    '''
    evals,evects = np.linalg.eigh(H)
    evects = evects.T
    for i,evect in enumerate(evects):
        #norm = np.sqrt(1/sci.integrate.simpson(evect*evect,grid))
        norm = 1/(np.linalg.norm(np.dot(evect,evect)))
        evects[i] = evects[i]*norm/np.sqrt(h)
    return evals,evects
```

Just like in the other examples, we'll need a pennylane compatible Hamiltonian given a requested basis dimension, N. For this problem however we'll pass a matrix with the matrix elements in directly.

```
def ham(N,mat_ele,mapper=JordanWignerMapper):
    # Start out by zeroing what will be our fermionic operator
    op = 0
    for i in range(N):
        for j in range(N):
            # Construct the terms of the Hamiltonian in terms of creation/
 ↪annihilation operators
            op +=  float(mat_ele[i][j]) * \
            FermionicOp([(["+", i),("-", j)], 1.0)])

    hamstr = "+ "+str(mapper().map(second_q_op=op))

    hamiltonian = parse_hamiltonian_input(hamstr)

    return hamiltonian
```

For the last bit of classical setup, we will define our parameters, along with the alpha values we want to inform our basis. We won't plot them again here, since they will be the same as before.

This is where we'll choose our basis size too, but we'll stick with two for now to be consistent.

```
nbasis = 2

h = 10**(-2) ### grid spacing for domain (Warning around 10**(-3) it starts to get␣
 ↪slow).

### HO global parameters
n = 0 # principle quantum number to solve in HO
mass = 1.0 # mass for the HO system

# define the domain boundaries
x_a = -10 # left boundary
x_b = 10 # right boundary
x_array = np.arange(x_a,x_b+h,h)
m = len(x_array)
```

(continues on next page)

(continued from previous page)

```
# Select alpha values to use to solve SE exactly.
alpha_vals = [.5,1,2,5,15]  #Here, we choose 5 values of alpha to solve exactly. This␣
 ↪results in 3 basis functions
# initialize solution arrays. T is the matrix that will hold wavefunction solutions.
# T has the form T[i][j], i = alpha, j = solution components
T = np.zeros((len(alpha_vals),m))
# T_evals holds the eigenvalues for each evect in T.
T_evals = np.zeros(len(alpha_vals))

for i,alpha_sample in enumerate(alpha_vals):
    H = construct_H(V,x_array,mass,alpha_sample) # construct the Hamiltonian matrix␣
 ↪for given alpha_sample.
    evals, evects = solve(H,x_array,h) # solve the system for evals and evects.
    T[i] = evects[n]/np.linalg.norm(evects[n]) # assign the nth evect to solution␣
 ↪array T
    T_evals[i] = evals[n] # assign the nth eigenvalue to the eigenvalue array T_eval.
    print(f'alpha = {alpha_sample}, lambda = {evals[n]}')

U, s, Vh = np.linalg.svd(T)
phi = []
for i in range(nbasis):
    phi.append(-1*Vh[i])

print("Basis constructed!")
```

```
alpha = 0.5, lambda = 0.7071036561740655
alpha = 1, lambda = 0.9999937499625953
alpha = 2, lambda = 1.4142010622616896
alpha = 5, lambda = 2.236036727063667
alpha = 15, lambda = 3.872889593935947
Basis constructed!
```

The last bit of housekeeping will involved us constructing our effective Hamiltonian for this two particle system again, just as the previous example.

```
dim0 = len(x_array)
off_diag = np.zeros(dim0)
off_diag[1] = 1

H0=-1*(-2*np.identity(dim0) + sci.linalg.toeplitz(off_diag))/(mass*h**2)
H1=np.diag(V(x_array,1))

tildeH0=np.zeros((nbasis,nbasis))
tildeH1=np.zeros((nbasis,nbasis))

for i in range(nbasis):
    for j in range(nbasis):
        tildeH0[i][j] = np.dot(phi[i],np.dot(H0,phi[j]))
        tildeH1[i][j] = np.dot(phi[i],np.dot(H1,phi[j]))

def tildeH(alpha):
    return tildeH0+alpha*tildeH1

def systemSolver(alpha):
```

(continues on next page)

(continued from previous page)

```
    resultssystem= np.linalg.eig(tildeH(alpha))
    if resultssystem[1][0][0]<0:
        resultssystem[1][0]=resultssystem[1][0]*(-1)
    if resultssystem[1][1][0]<0:
        resultssystem[1][1]=resultssystem[1][1]*(-1)
    return resultssystem

def phibuilder(coefficients):
    return coefficients[0]*phi[0]+coefficients[1]*phi[1]
```

Now we can get into the quantum side of things! We'll be defining our ansatz in a general way, but we've included a simpler ansatz as well.

For the Hamiltonian we'll use the helper function we just made to get the matrix elements and build our quantum-ready problem.

```
# Set alpha

alph = 3

# Set the order you'd like to go to

dim = nbasis

dev = qml.device("default.qubit", wires=dim)

# Define a general ansatz for arbitrary numbers of dimensions

particles = 1

ref_state = qml.qchem.hf_state(particles, dim)

gen_ansatz = partial(qml.ParticleConservingU2, init_state=ref_state)

layers = dim

def ansatz(params,wires):
    t0 = params[0]
    qml.PauliX(wires=0)
    qml.RY(t0, wires=1)
    qml.CNOT(wires=[1,0])

# generate the cost function
@qml.qnode(dev)
def prob_circuit(params,ansatz):
    gen_ansatz(params, wires=dev.wires)
    return qml.probs(wires=dev.wires)

# Defining Hamiltonian

mat_ele = tildeH(3)

H = ham(dim, mat_ele)

print(H)
```

(continues on next page)

(continued from previous page)

```
cost_fn = qml.ExpvalCost(gen_ansatz, H, dev)
```

```
  (-4.410191127127895) [Z0]
+ (-0.8896021203242128) [Z1]
+ (5.299793247452108) [I0]
+ (0.1526076698211456) [Y0 Y1]
+ (0.1526076698211456) [X0 X1]
```

We're definitely on the right track, now to do the VQE.

Let's set up our initial parameters.

```
# Our parameter array

init_params = np.random.uniform(low=-np.pi / 2, high=np.pi / 2, size=qml.
 ↪ParticleConservingU2.shape(n_layers=layers, n_wires=dim))
#init_params = np.array([2.5,])
print(init_params)
# Convergence information and step size

max_iterations = 500
conv_tol = 1e-05
step_size = 0.1
```

```
[[-1.28431806 -0.44740746 -0.75320554]
 [-1.15793302 -1.50851015 -0.48926459]]
```

And now the VQE loop just as the other examples.

```
opt = qml.GradientDescentOptimizer(stepsize=step_size)

params = init_params

gd_param_history = [params]
gd_cost_history = []

accel = 0
prev_conv = -1.0

start = time.time()

for n in range(max_iterations):
    fac = (1.0)
    opt = qml.GradientDescentOptimizer(stepsize=step_size)

    # Take a step in parameter space and record your energy
    params, prev_energy = opt.step_and_cost(cost_fn, params)

    # This keeps track of our energy for plotting at comparisons
    gd_param_history.append(params)
    gd_cost_history.append(prev_energy)

    # Here we see what the energy of our system is with the new parameters
    energy = cost_fn(params)
```

(continues on next page)

(continued from previous page)

```
    # Calculate difference between new and old energies
    conv = np.abs(energy - prev_energy)

    if(energy - prev_energy > 0.0 and step_size > 0.001):
        #print("Lowering!")
        accel = 0
        step_size = 0.5*step_size

    if(conv < prev_conv): accel += 1
    prev_conv = conv

    if(accel > 10 and step_size < 1.0):
        #print("Accelerating!")
        step_size = 1.1*step_size
    end = time.time()

    if n % 1 == 0:
        print(
            "It = {:},  Energy = {:.8f},  Conv = {"
            ":.8f}, Time Elapsed = {:.3f} s".format(n, energy, conv,end-start)
        )
        start = time.time()


    if conv <= conv_tol:
        break

print()
print("Final value of the energy = {:.8f}".format(energy))
print("Number of iterations = ", n)
```

```
It = 0,  Energy = 3.45108482,  Conv = 0.65783415, Time Elapsed = 0.057 s
It = 1,  Energy = 1.85248904,  Conv = 1.59859577, Time Elapsed = 0.040 s
It = 2,  Energy = 1.78916111,  Conv = 0.06332794, Time Elapsed = 0.032 s
It = 3,  Energy = 1.77401069,  Conv = 0.01515042, Time Elapsed = 0.033 s
It = 4,  Energy = 1.76922936,  Conv = 0.00478133, Time Elapsed = 0.033 s
It = 5,  Energy = 1.76740033,  Conv = 0.00182903, Time Elapsed = 0.031 s
It = 6,  Energy = 1.76662534,  Conv = 0.00077498, Time Elapsed = 0.031 s
It = 7,  Energy = 1.76628210,  Conv = 0.00034324, Time Elapsed = 0.088 s
It = 8,  Energy = 1.76612743,  Conv = 0.00015467, Time Elapsed = 0.031 s
It = 9,  Energy = 1.76605728,  Conv = 0.00007015, Time Elapsed = 0.033 s
It = 10,  Energy = 1.76602540,  Conv = 0.00003189, Time Elapsed = 0.029 s
It = 11,  Energy = 1.76601089,  Conv = 0.00001451, Time Elapsed = 0.032 s
It = 12,  Energy = 1.76600429,  Conv = 0.00000660, Time Elapsed = 0.029 s

Final value of the energy = 1.76600429
Number of iterations =  12
```

We converged! That's great, but how do we interpret these results? The value we converged to here is exactly the same as the eigenvalue we recovered when we did direct diagonalization, as you may expect.

In order to uncover the coefficients of our basis elements, we need to do a little more and measure the probabilities of our system.

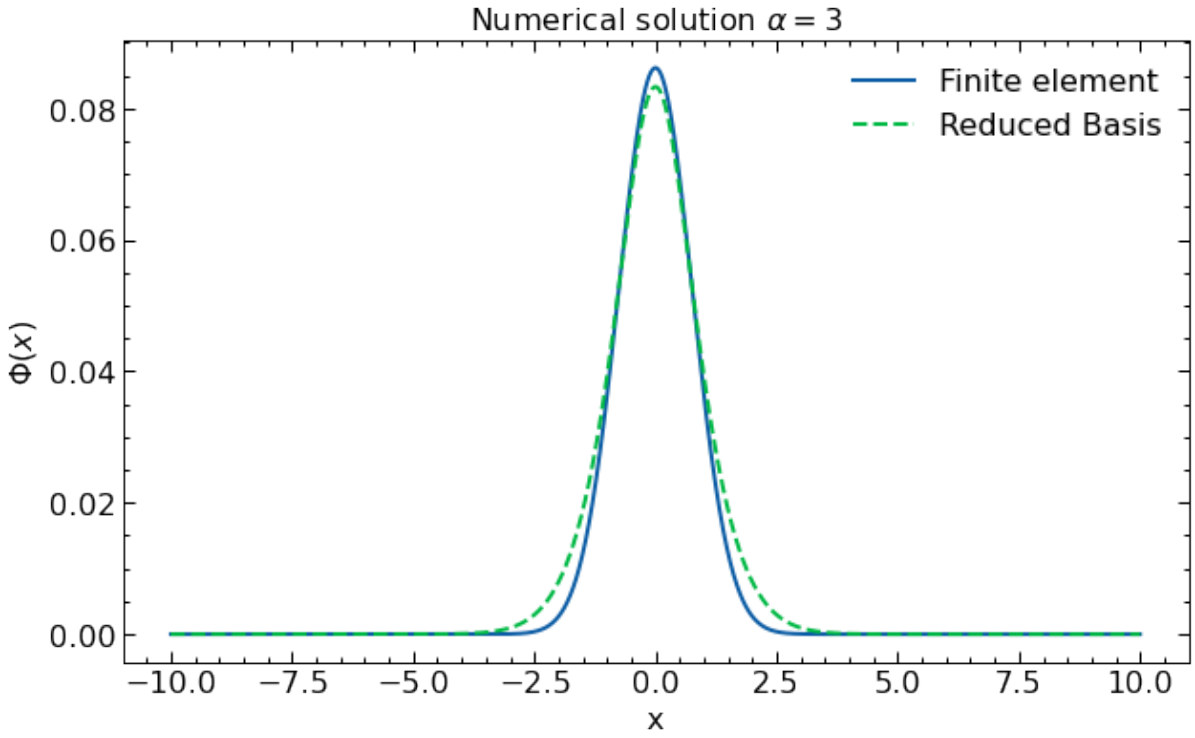
```
probs = prob_circuit(params,gen_ansatz)

coeffs = np.sqrt(probs[1:3])

print("a1 Coefficient: ",coeffs[0])
print("a2 Coefficient: ",coeffs[1])
```

```
a1 Coefficient:  0.9991031231376774
a2 Coefficient:  0.04234323247626702
```

Brilliant! It's exactly what we expected to see, and we can finish out by plotting our solution against the true value just for completion's sake.

```
# Now we can construct our RBM for an alpha of our choosing.
alpha_k = alph
solGaler=phibuilder(coeffs)
lamGaler=energy

alpha_vals = [alpha_k]

T = np.zeros((len(alpha_vals),m))

T_evals = np.zeros(len(alpha_vals))

for i,alpha_sample in enumerate(alpha_vals):
    H = construct_H(V,x_array,mass,alpha_sample) # construct the Hamiltonian matrix␣
↪for given alpha_sample.
    evals, evects = solve(H,x_array,h) # solve the system for evals and evects.
    T[i] = evects[i]/np.linalg.norm(evects[i]) # assign the nth evect to solution␣
↪array T
    T_evals[i] = evals[i] # assign the nth eigenvalue to the eigenvalue array T_eval.
    print(f'Finite Ele alpha = {alpha_sample}, lambda = {evals[i]}')

print(f'QC Galerkin alpha = {alph}, lambda = {lamGaler}')



# Make plots of the numerical wavefunction
fig, ax = plt.subplots(1,1,figsize=(10,6))
for i in range(len(alpha_vals)):

    ax.plot(x_array,(T[i]),label= r'Finite element')

ax.plot(x_array,(solGaler),label=r'Reduced Basis',linestyle='dashed')


ax.set_title('Numerical solution ' r'$\alpha=$'+str(alpha_vals[i]))

ax.legend()
ax.legend(loc='upper right')
plt.ylabel("$\Phi(x)$")
plt.xlabel("x")

plt.show()
```

```
Finite Ele alpha = 3, lambda = 1.7320320573659755
QC Galerkin alpha = 3, lambda = 1.7660042901259638
```



Phew, that was an adventure! But we were able to successfully reproduce everything we could do on the classical computing side just fine. By utilizing the reduced basis method we were able to vastly reduce the dimensionality of solving our problem in this simple case. For our last adventure we'll try to pull together everything we've built in order to solve a tricky non-linear quantum many-body problem.

CHAPTER

FIVE

FINAL CHALLENGE: SOLVING THE GROSS-PITAEVSKII EQUATION

Author(s): Kyle Godbey, Pablo Giuliani

Maintainer: Kyle Godbey

Parts adapted from this paper with permission from the authors.

5.1 Background

The Gross-Pitaevskii (GP) equation (also see here for a RBM application) is a nonlinear Schrödinger equation that approximately describes the low-energy properties of dilute Bose-Einstein condensates. Using a self-consistent mean field approximation, the many-body wavefunction is reduced to a description in terms of a single complex-valued wavefunction $\phi(\vec{r})$. We work with the one-dimensional Gross-Pitaevskii equation with a harmonic trapping potential by letting F_α be:

$$F_{q,\kappa}(\phi) = -\phi'' + \kappa x^2 \phi + q|\phi|^2\phi - \lambda_{q,\kappa}\phi = 0, \tag{5.1}$$

where the parameters are now $\alpha = \{\kappa, q\}$, which are proportional to the strength of the harmonic trapping and the self-coupling of the wavefunction,respectively, while $\lambda_{q,\kappa}$ is proportional to the ground state energ. $\phi(x)$ is a single variable function that depends on x and it is normalized to unity.

For a classical algorithm, the finite element method will work for this non-linear equation in the same way as it did for the harmonic oscillator. For the quantum computing counterpart, we will need to follow the same procedure as with the Harmonic Oscillator problem and cast the equations in the reduced basis. We once again expand our solution in a basis of n elements:

$$\hat{\phi}_\alpha(x) = \sum_{i=1}^{n} a_i \phi_i(x) \tag{5.2}$$

where $\phi_i(x)$ are informed on previous solutions built on a classical computer to the non-linear Schrodinger equation for chosen values of $\alpha = \{\kappa, q\}$.

The main challenge now resides in the fact that we can't write F_α in terms of linear operators since the "Hamiltonian" is dependent on the actual solution $\phi(x)$: $H_\alpha \equiv H_\alpha[\phi] = (-\frac{d^2}{dx^2} + \kappa x^2 + q|\phi|^2)$. The Galerkin projection equations are general enough that this problem can be tackled directly, which leads to a series of non-linear equations (up to fourth powers) in the unknown coefficients a_i and the approximated eigenvalue $\hat{\lambda}$.

In order to tackle this problem in a quantum circuit with the machinery we have developed up to this point we propose an approach that allows one to avoid having to solve a non-linear problem by instead solving a linear one many times in an iteratively scheme. This is the approach frequently used in many Density Functional Theory implementations. We make a guess, which we call ϕ_0, on the solution of the original non-linear problem. We now iteratively find the solution ϕ_n to a linear Schrodinger equation with Hamiltonian $H_\alpha[\phi_{n-1}]$. If the original guess ϕ_0 was close enough to the true solution, then with enough iterations we should reach self consistency and obtain a wave function such that $F_\alpha[\phi] = H_\alpha[\phi]\phi - \lambda\phi = 0$.

In the following we implement this iterative approach to solve the GP equation first in a classical computer using the finite element method, and later using a reduced basis with two states. We then proceed to design the solution for a quantum computer.

5.2 Finite element method approach

```
from pennylane import numpy as np
import scipy as sci
from scipy import optimize
from scipy import special
import matplotlib.pyplot as plt
import pennylane as qml
from qiskit_nature.operators.second_quantization import FermionicOp
from qiskit_nature.mappers.second_quantization import JordanWignerMapper, ParityMapper
import time
from functools import partial
import warnings
plt.style.use(['science','notebook'])

warnings.filterwarnings('ignore')
```

```
### NOTE: hbar = 1 in this demo

def V(x,kappa):
    '''
    1-d harmonic Oscillator potential

    Parameters
    ----------
    x : float or nd array
        position.
    alpha : float
        oscillator length parameter.

    Returns
    -------
    float or ndarray
        value of potential evaluated at x.

    '''
    return kappa*x**2

def construct_H(grid,mass,kappa,q,PHIgrid):
   #PHIgrid is the value of the guess for the nonlinear part at each location
    dim = len(grid)
    off_diag = np.zeros(dim)
    off_diag[1] = 1
    H = -1*(-2*np.identity(dim) + sci.linalg.toeplitz(off_diag))/(mass*h**2) + np.
 ↪diag(V(grid,kappa))+ q*np.diag(PHIgrid**2)

    return H
```

```
#First define global variables needed for the Finite element method
h =  1*10**(-1) ### grid spacing for domain (Warning around 10**(-3) it starts to get␣
 ↪slow).
```

(continues on next page)

(continued from previous page)

```
n=0 #The number of the state we are interest in (the ground state in this case)
mass = 1.0 # mass for the system
# define the domain boundaries
x_a = -10 # left boundary
x_b = 10 # right boundary
x_array = np.arange(x_a,x_b+h,h)
m = len(x_array)
print('Number of grid points: ',m)


def solve(H,grid,h):
    '''
    Parameters
    ----------
    H : 2d ndarray
        Hamiltonian Matrix.
    grid : ndarray
        Discretized 1d domain.
    h : float
        grid spacing.

    Returns
    -------
    evals : ndarray
        returns nd array of eigenvalues of H.
    evects : ndarray
        returns ndarray of eigenvectors of H.
    Eigenvalues and eigenvectors are ordered in ascending order.
    '''
    evals,evects = np.linalg.eigh(H)
    evects = evects.T
    for i,evect in enumerate(evects):
        norm = 1/(np.linalg.norm(evect))
        evects[i] = evects[i]*norm/np.sqrt(h)
    return evals,evects


def IterativeSolver(grid,q,kappa,maxIterations):
    #This solver will find a self consistent solution by iteratively solving the␣
 ↪linear problem as stated above. Since we might find oscillatory behavior
    # (the solution does not converge but bounces between two possibilities), we don
 ↪'t update the potential part directly but rather make an average with the
    # previous 6 solutions, making the convergence slower but increasing the chances␣
 ↪of getting the correct self consistent answer.

    Hn = construct_H(grid,1,kappa,0,np.zeros(len(x_array))) # First Hamiltonian with␣
 ↪the q interaction put to zero
    PHIn=solve(Hn,grid,h)[1][0]
    ListOfPhiOlds=[np.copy(PHIn),np.copy(PHIn),np.copy(PHIn),np.copy(PHIn),np.
 ↪copy(PHIn),np.copy(PHIn)]

    for IJ in range(maxIterations):
        if IJ<maxIterations/2:
            qeff=q*(IJ/maxIterations)*2
        else:
            qeff=q
```

(continues on next page)

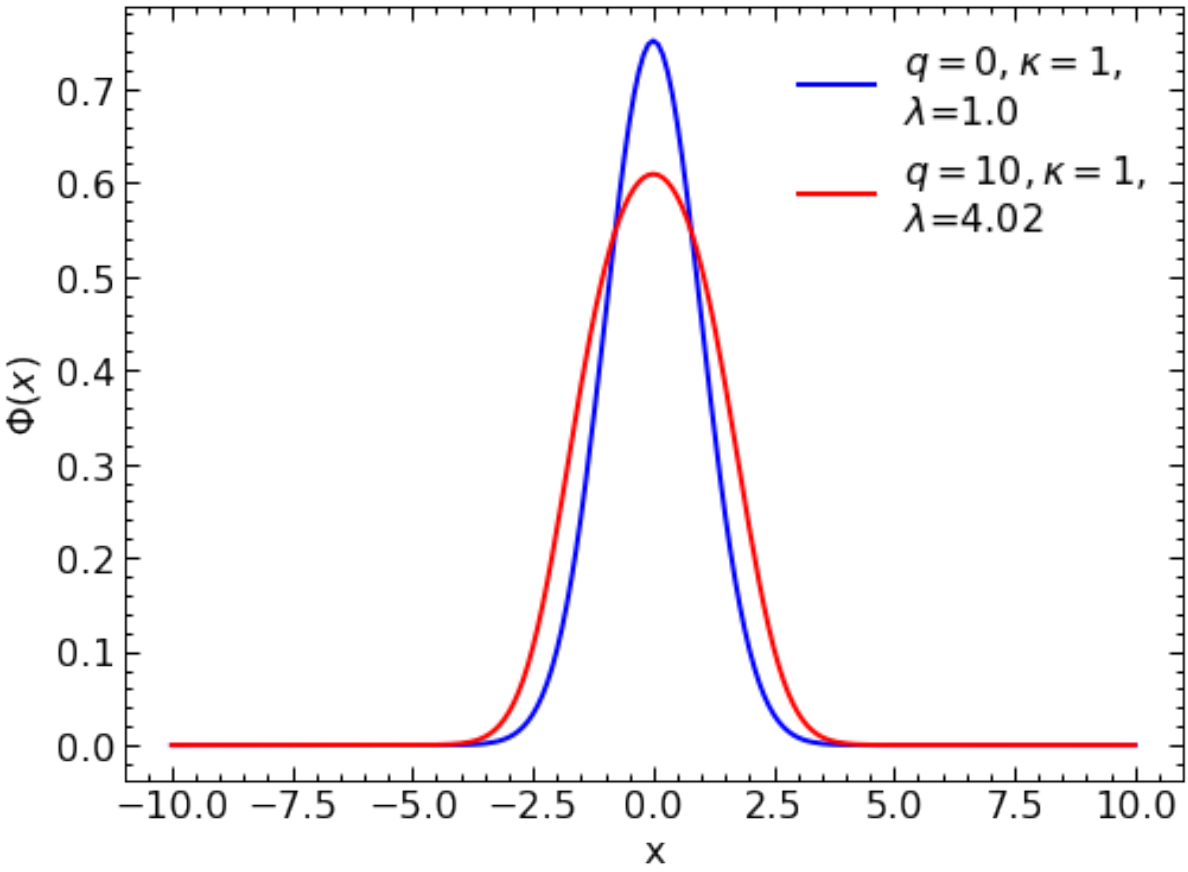
(continued from previous page)

```
        Hn=construct_H(grid,1,kappa,qeff,(ListOfPhiOlds[-1]+ListOfPhiOlds[-
↪2]+ListOfPhiOlds[-3]+ListOfPhiOlds[-4]+ListOfPhiOlds[-5]+ListOfPhiOlds[-6])/6.0)
        SOL=solve(Hn,grid,h)

        PHIn=SOL[1][0]
        ListOfPhiOlds.append(np.copy(PHIn))
    #This returns the set of eigenvalues, followed by the set of eigenvecctors.
    return SOL
```

```
Number of grid points:  201
```

```
#In here we compare the solution without self interaction (q=0) with one with self␣
↪interaction after 1000 iterations of the algorithm (q=10)

solution0=IterativeSolver(x_array,0,1,1)
solution=IterativeSolver(x_array,10,1,1000)
```

```
plt.plot(x_array, solution0[1][0], color="blue",label = r'$q=0, \kappa=1,$' "\n" '$\
↪lambda$='+str(round(solution0[0][0],2)))
plt.plot(x_array, abs(solution[1][0]), color="red",label = r'$q=10, \kappa=1,$' "\n" '
↪$\lambda$='+str(round(solution[0][0],2)))
plt.legend(loc='upper right')

plt.ylabel("$\Phi(x)$")
plt.xlabel("x")
plt.show()
```



5.3 The Reduced Basis Method approach

5.3.1 Construction of the Reduced Basis

Now we select a set of values of α to solve with the finite element method. These solutions will be used as the basis in the reduced basis model.

```
# Select alpha values to use to solve SE exactly.
alpha_vals = [[0,1],[1,1],[0,2],[5,1],[10,1]]  #[q,kappa] Here, we choose 5 values of␣
↪alpha to solve exactly. This results in 3 basis functions
# initialize solution arrays. T is the matrix that will hold wavefunction solutions.
# T has the form T[i][j], i = alpha, j = solution components
T = np.zeros((len(alpha_vals),m))
# T_evals holds the eigenvalues for each evect in T.
T_evals = np.zeros(len(alpha_vals))
```

```
for i,alpha_sample in enumerate(alpha_vals):

    evals, evects = IterativeSolver(x_array,alpha_sample[0],alpha_sample[1],700) #␣
↪solve the system for evals and evects.
    T[i] = evects[n]/np.linalg.norm(evects[n]) # assign the nth evect to solution␣
↪array T
    T_evals[i] = evals[n] # assign the nth eigenvalue to the eigenvalue array T_eval.
```

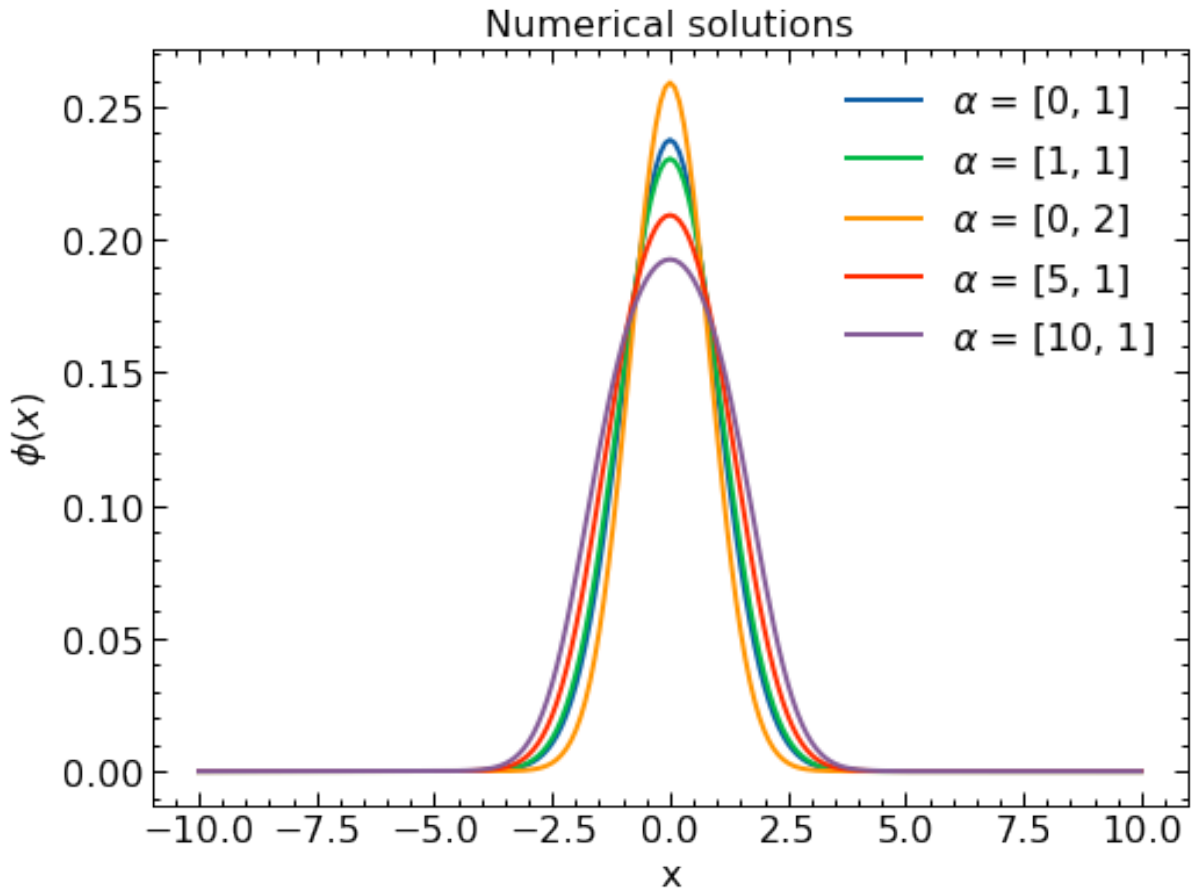
(continues on next page)

(continued from previous page)

```
    print(f'alpha = {alpha_sample}, lambda = {round(evals[n],3)}')
```

```
alpha = [0, 1], lambda = 0.999
alpha = [1, 1], lambda = 1.383
alpha = [0, 2], lambda = 1.413
alpha = [5, 1], lambda = 2.69
alpha = [10, 1], lambda = 4.023
```

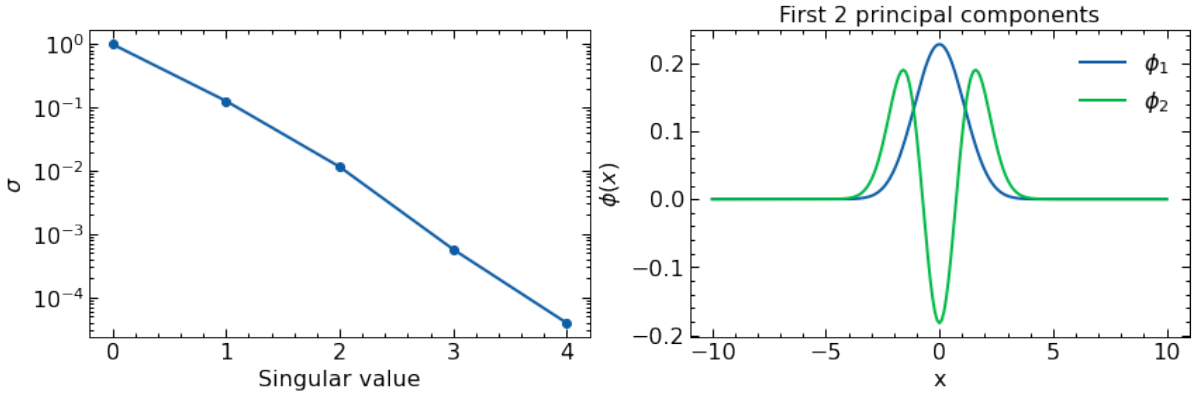
```
# Make plots of the numerical wavefunctions
# fig, ax = plt.subplots(1,1,figsize=(5,3))
for i in range(len(alpha_vals)):

    plt.plot(x_array,(T[i]),label=r'$\alpha$ = '+str(alpha_vals[i]))

plt.title('Numerical solutions')

plt.legend()

plt.ylabel(r'$\phi(x)$')
plt.xlabel("x")

plt.show()
```



5.3.2 Proper Orthogonal Decomposition

Followling the same procedure as with the Harmonic Oscillator, after obtained a sample of what the solutions look like for some values of the parameters we proceed now to construct our basis with two elements. These elements will be selected as the first two principal components of a Principal Component Analysis or Proper Orthogonal Decomposition aiming at capturing the two components that can better explain (in a linear way) the variability among our samples.

```
U, s, Vh = np.linalg.svd(T)
phi1=-1*Vh[0] #multiplying by -1 so they are positive at x=0, not necessary but the␣
 ↪plots look better
phi2=1*Vh[1]
n_comps = 2 # number of principle components to plot (i.e number of column vectors of␣
 ↪SVD matrix V to plot)
fig, ax = plt.subplots(1,2,figsize=(14,4))
ax[0].semilogy(s/s[0],'o-')
ax[0].set_xlabel(r'Singular value')
ax[0].set_ylabel(r'$\sigma$')

ax[1].plot(x_array,phi1,label=r'$\phi_1$')
ax[1].plot(x_array,phi2,label=r'$\phi_2$')
ax[1].set_xlabel(r'x')
ax[1].set_ylabel(r'$\phi(x)$')
ax[1].set_title(f'First {n_comps} principal components')
ax[1].legend()
plt.show()
```



5.3.3 Effective two level system "Hamiltonian"

This discussion retraces everything we did before for the Harmonic Oscillator. The main difference is that now the Hamiltonian effectively depends on the solution $H_\alpha \equiv H_\alpha[\phi]$, but since we are aiming at solving the problem iteratively, at each stage we have a fixed linear Hamiltonian.

Here we explicitly construct $\hat{\phi}_\alpha = a_1\phi_1 + a_2\phi_2$ and use the Galerkin method, that is, projecting $F_\alpha(\hat{\phi}_\alpha(x))$ over 2 linearly independent functions projecting functions $\{\psi_i\}_{i=1}^2$.

$$\langle\psi_1|F_\alpha(\hat{\phi}_\alpha(x))\rangle = 0 \tag{5.3}$$

$$\langle\psi_2|F_\alpha(\hat{\phi}_\alpha(x))\rangle = 0 \tag{5.4}$$

We can interpret this as enforcing the orthogonality of $F_\alpha(\hat{\phi}_\alpha(x))$ to the subspace spanned by $\{\psi_i\}$ that is, by finding $\hat{\phi}_\alpha$ such that $F_\alpha(\hat{\phi}_\alpha)$ is approximately zero up to the ability of the set $\{\psi_i\}$. The choice of projecting functions $\{\psi_i\}$ is

arbitrary, but here we choose the solution set $\{\phi_i\}$ to be our projecting functions to make our lives easier. Since λ is also unknown, we need an additional equation. This comes from the normalization conditions:

$$\langle\hat{\phi}_{\alpha_k}|\hat{\phi}_{\alpha_k}\rangle = 1 \tag{5.5}$$

We can re-write the projecting equations taking advantage of the linear form of F_α to obtain an effective 2-level system Hamiltonian. We note that:

$$\langle\phi_i|F_\alpha(\hat{\phi}_\alpha(x))\rangle = \tag{5.6}$$

$$\langle\phi_i|a_1 H_\alpha\phi_1 + a_2 H_\alpha\phi_2 - a_1\hat{\lambda}\phi_1 - a_2\hat{\lambda}\phi_2\rangle \tag{5.7}$$

Since $\langle\phi_i|\phi_j\rangle = \delta_{i,j}$, we arrive at the following matrix equation for the projecting equations:

$$\tilde{H}_\alpha|a\rangle = \hat{\lambda}|a\rangle \tag{5.8}$$

where $|a\rangle = \{a_1, a_2\}$ and

$$\tilde{H}_\alpha = \begin{bmatrix}\langle\phi_1|H_\alpha|\phi_1\rangle & \langle\phi_1|H_\alpha|\phi_2\rangle \\ \langle\phi_2|H_\alpha|\phi_1\rangle & \langle\phi_2|H_\alpha|\phi_2\rangle\end{bmatrix} \tag{5.9}$$

while the normalization condition translates into:

$$\langle a|a\rangle = a_1^2 + a_2^2 = 1 \tag{5.10}$$

Now we proceed to construct this Hamiltonian matrix for our problem at hand. We note that although $\tilde{H}_\alpha$ depennds on α, the dependance is only in the two potential parts and it is affine (linear): $H_\alpha = H_0 + \kappa H_1 + qH_2$, where $H_0 = -\frac{d^2}{dx^2}$, $H_1 = x^2$, and $H_2 = |\phi(x)|^2$. We then decompose $\tilde{H}_\alpha = \tilde{H}_0 + \kappa\tilde{H}_1 + q\tilde{H}_2$. We now construct these matrices, H_2 being a function that must be constructed every time we wish to iterate the approach.l

```
dim0 = len(x_array)
off_diag = np.zeros(dim0)
off_diag[1] = 1

H0=-1*(-2*np.identity(dim0) + sci.linalg.toeplitz(off_diag))/(mass*h**2)
H1=np.diag(V(x_array,1))

tildeH0=np.array([[0.0,0.0],[0.0,0.0]])
tildeH1=np.array([[0.0,0.0],[0.0,0.0]])

tildeH0[0][0]=np.dot(phi1,np.dot(H0,phi1))
tildeH0[0][1]=np.dot(phi1,np.dot(H0,phi2))
tildeH0[1][0]=np.dot(phi2,np.dot(H0,phi1))
tildeH0[1][1]=np.dot(phi2,np.dot(H0,phi2))

tildeH1[0][0]=np.dot(phi1,np.dot(H1,phi1))
tildeH1[0][1]=np.dot(phi1,np.dot(H1,phi2))
tildeH1[1][0]=np.dot(phi2,np.dot(H1,phi1))
tildeH1[1][1]=np.dot(phi2,np.dot(H1,phi2))



#In the following we define the pieces needed for H2 tilde

H2_11=np.diag(phi1**2)
H2_12=np.diag(phi1*phi2)
H2_22=np.diag(phi2**2)

tildeH2_11=np.array([[0.0,0.0],[0.0,0.0]])
```

(continues on next page)

(continued from previous page)

```
tildeH2_11[0][0]=np.dot(phi1,np.dot(H2_11,phi1))
tildeH2_11[0][1]=np.dot(phi1,np.dot(H2_11,phi2))
tildeH2_11[1][0]=np.dot(phi2,np.dot(H2_11,phi1))
tildeH2_11[1][1]=np.dot(phi2,np.dot(H2_11,phi2))




tildeH2_12=np.array([[0.0,0.0],[0.0,0.0]])
tildeH2_12[0][0]=np.dot(phi1,np.dot(H2_12,phi1))
tildeH2_12[0][1]=np.dot(phi1,np.dot(H2_12,phi2))
tildeH2_12[1][0]=np.dot(phi2,np.dot(H2_12,phi1))
tildeH2_12[1][1]=np.dot(phi2,np.dot(H2_12,phi2))


tildeH2_22=np.array([[0.0,0.0],[0.0,0.0]])
tildeH2_22[0][0]=np.dot(phi1,np.dot(H2_22,phi1))
tildeH2_22[0][1]=np.dot(phi1,np.dot(H2_22,phi2))
tildeH2_22[1][0]=np.dot(phi2,np.dot(H2_22,phi1))
tildeH2_22[1][1]=np.dot(phi2,np.dot(H2_22,phi2))

def tildeH2(coeffs):

    return (coeffs[0]**2*tildeH2_11+2*coeffs[0]*coeffs[1]*tildeH2_
↪12+coeffs[1]**2*tildeH2_22)/h
```

We print the effective Hamiltonian parts for some sets of parameters just as a sanity check

```
print('H0=', tildeH0)
print('H1=',tildeH1)

print('H2=',tildeH2([1,0]))
```

```
H0= [[ 0.42023097 -0.58365092]
 [-0.58365092  2.61560759]]
H1= [[0.59436083 0.82835933]
 [0.82835933 2.3627665 ]]
H2= [[ 0.36584365 -0.13715205]
 [-0.13715205  0.16958109]]
```

Now we are ready to create the iterative solver (in the Reduced Basis space!)

```
def tildeH(alpha,coeffs):
    return tildeH0+alpha[1]*tildeH1+alpha[0]*tildeH2(coeffs)

def systemSolver(alpha,coeffs):
    resultssystem= np.linalg.eig(tildeH(alpha,coeffs))
    # if resultssystem[1][0][0]<0:
    #     resultssystem[1][0]=resultssystem[1][0]*(-1)
    # if resultssystem[1][1][0]<0:
    #     resultssystem[1][1]=resultssystem[1][1]*(-1)
    return [resultssystem[0],np.transpose(resultssystem[1])]

def phibuilderFromCoeffs(coefficients):
```

(continues on next page)

(continued from previous page)

```
    coefficients0=[coefficients[0]/(np.linalg.norm(coefficients))/np.sqrt(h),
↪coefficients[1]/(np.linalg.norm(coefficients))/np.sqrt(h)]
    return coefficients0[0]*phi1+coefficients0[1]*phi2

def IterativeRBMsolver(alpha,maxIterations):
    sol0=systemSolver([0,alpha[1]],[1,0])

    ListOfPhiOlds=[np.copy(sol0[1][0]),np.copy(sol0[1][0]),np.copy(sol0[1][0]),np.
↪copy(sol0[1][0]),np.copy(sol0[1][0]),np.copy(sol0[1][0])]
    for IJ in range(maxIterations):
        #We start building the interaction term adiabatically, so as not to scare the␣
↪system too much
        if IJ<maxIterations/2:
            qeff=alpha[0]*(IJ/maxIterations)*2
        else:
            qeff=alpha[0]
        #As before, we use an average of the terms for the potential to avoid␣
↪oscillations
        sol0=systemSolver([qeff,alpha[1]],
        (ListOfPhiOlds[-1]+ListOfPhiOlds[-2]+ListOfPhiOlds[-3]+ListOfPhiOlds[-
↪4]+ListOfPhiOlds[-5]+ListOfPhiOlds[-6])/6.0
        )


        ListOfPhiOlds.append(np.copy(sol0[1][0]))
    return sol0
```

Now we try our solution solver against the finite element computation:

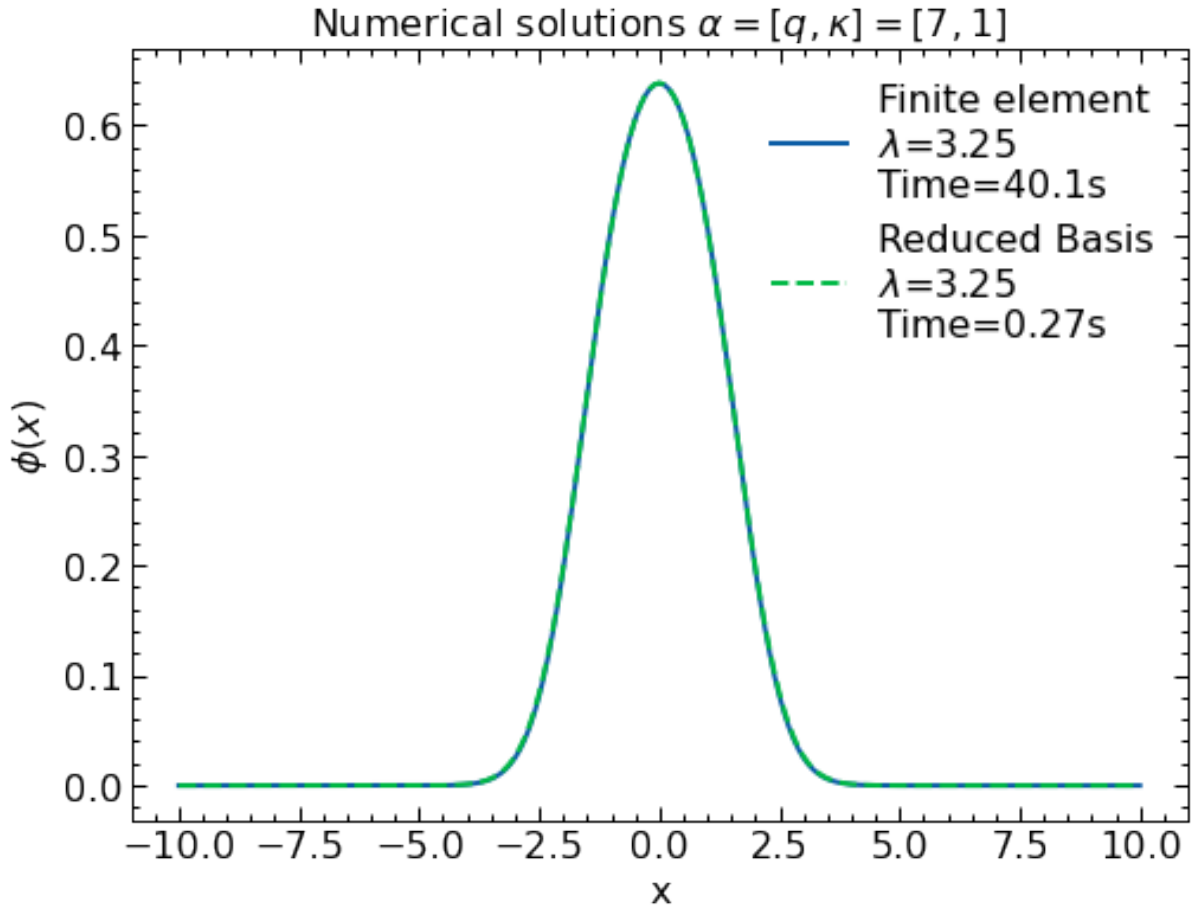
```
alpha_k = [7,1]



st = time.time()
solFiniteFull=IterativeSolver(x_array,alpha_k[0],alpha_k[1],700)
et = time.time()
elapsed_timeFinite = et - st

solFinite=solFiniteFull[1][0]
lamFinite=solFiniteFull[0][0]



st = time.time()
solFull=IterativeRBMsolver(alpha_k,700)
et = time.time()
elapsed_timeRB = et - st

solGaler=phibuilderFromCoeffs(solFull[1][0])
lamGaler=solFull[0][0]
```

```
# Make plots of the numerical wavefunctions
fig, ax = plt.subplots(1,1)
```

(continues on next page)

(continued from previous page)

```
ax.plot(x_array,solFinite,label=r'Finite element'"\n" '$\lambda$=
↪'+str(round(lamFinite,2)) + "\n" 'Time='+str(round(elapsed_timeFinite,1)) + 's'  )


ax.plot(x_array,-solGaler,label= r'Reduced Basis'"\n" '$\lambda$='+str(round(lamGaler,
↪2))+ "\n" 'Time='+str(round(elapsed_timeRB,2)) + 's' ,linestyle='dashed')



ax.set_title('Numerical solutions ' r'$\alpha=[q,\kappa]=[7,1]$')
plt.ylabel(r'$\phi(x)$')
plt.xlabel("x")
ax.legend()

plt.show()
print("Computational speed gain= ",round(elapsed_timeFinite/elapsed_timeRB,1))
```




```
Computational speed gain=  148.9
```

Interestingly enough, not only the Reduced Basis reproduces the wave function and energy perfectly, it does so almost 200 times faster than the original finite element computation (hurray for reduced basis methods!).

5.4 Quantum Computing on the Reduced Basis

Now that we've laid the groundwork for getting RBM solutions for this nonlinear problem, we can explore how to get it running in the quantum paradigm. The next cell will contain a lot of setup without much explanation, so it's recommended that you check back in the VQE, noise, and RBM sections for individual explanations.

```
def pauli_token_to_operator(token):
    qubit_terms = []

    for term in range(len(token)):
        # Special case of identity
        if token[term] == "I":
            pass
        else:
            #pauli, qubit_idx = term, term
            if token[term] == "X":
                qubit_terms.append(qml.PauliX(int(term)))
            elif token[term] == "Y":
                qubit_terms.append(qml.PauliY(int(term)))
            elif token[term] == "Z":
                qubit_terms.append(qml.PauliZ(int(term)))
            else:
                print("Invalid input.")
    if(qubit_terms==[]):
            qubit_terms.append(qml.Identity(0))
    full_term = qubit_terms[0]
    for term in qubit_terms[1:]:
        full_term = full_term @ term

    return full_term


def parse_hamiltonian_input(input_data):
    # Get the input
    coeffs = []
    pauli_terms = []
    chunks = input_data.split("\n")
    # Go through line by line and build up the Hamiltonian
    for line in chunks:
        #line = line.strip()
        tokens = line.split(" ")
        # Parse coefficients
        sign, value = tokens[0][0], tokens[1]

        coeff = float(value)
        if sign == "-":
            coeff *= -1
        coeffs.append(coeff)

        # Parse Pauli component
        pauli = tokens[3]

        pauli_terms.append(pauli_token_to_operator(pauli))

    return qml.Hamiltonian(coeffs, pauli_terms)

def ham(N,mat_ele,mapper=JordanWignerMapper):
```

(continues on next page)

(continued from previous page)

```
    # Start out by zeroing what will be our fermionic operator
    op = 0
    for i in range(N):
        for j in range(N):
            # Construct the terms of the Hamiltonian in terms of creation/
 ↪annihilation operators
            op +=  float(mat_ele[i][j]) * \
            FermionicOp([(("+", i),("-", j)], 1.0)])

    hamstr = "+ "+str(mapper().map(second_q_op=op))

    hamiltonian = parse_hamiltonian_input(hamstr)

    return hamiltonian
```

We'll define functions to help with running VQE since we will likely need to do it a few times.

`update_ham` will return an updated hamiltonian given basis coefficients and the alpha vector

`run_vqe_fixed` will run the VQE algorithm for a fixed hamiltonian defined by input coefficients

`run_vqe` performs VQE with a dynamically updating Hamiltonian during the gradient descent. This algorithm decreases the total number of iterations required for convergence at the cost of additional measurements per step – but it tends towards being much more efficient.

```
def update_ham(alpha,coeffs):
    # Define dimension
    dim = len(coeffs)

    # Define Hamiltonian
    mat_ele = tildeH(alpha,coeffs)

    H = ham(dim, mat_ele)
    return H

def run_vqe_fixed(alpha, coeffs, init_params, dev, max_iter=500, conv_tol=1e-6, step_
 ↪size=0.1, debug=False):

    H = update_ham(alpha,coeffs)

    cost_fn = qml.ExpvalCost(ansatz, H, dev)

    opt = qml.GradientDescentOptimizer(stepsize=step_size)

    params = init_params

    gd_param_history = [params]
    gd_cost_history = []

    start = time.time()

    for n in range(max_iter):
        # Take a step in parameter space and record your energy
        params, prev_energy = opt.step_and_cost(cost_fn, params)

        # This keeps track of our energy for plotting at comparisons
        gd_param_history.append(params)
```

(continues on next page)

(continued from previous page)

```
        gd_cost_history.append(prev_energy)

        # Here we see what the energy of our system is with the new parameters
        energy = cost_fn(params)

        # Calculate difference between new and old energies
        conv = np.abs(energy - prev_energy)

        end = time.time()

        if (n % 1 == 0) and debug:
            print(
                "It = {:},  Energy = {:.8f},  Conv = {"
                ":.8f}, Time Elapsed = {:.3f} s".format(n, energy, conv,end-start)
            )
            start = time.time()


        if conv <= conv_tol:
            break

    if(debug): print("Final value of the energy = {:.8f}".format(energy))
    if(debug): print("Number of iterations = ", n)

    return params, energy, n, gd_cost_history

def run_vqe(alpha, coeffs, init_params, dev, mixing=0.5, max_iter=500, \
    noisy=False, conv_tol=1e-6, step_size=0.1, debug=False):
    if(noisy):
        prob_circ = noisy_prob_circuit
    else:
        prob_circ = prob_circuit

    H = update_ham(alpha,coeffs)

    cost_fn = qml.ExpvalCost(ansatz, H, dev)

    opt = qml.GradientDescentOptimizer(stepsize=step_size)

    params = init_params

    probs = prob_circ(params)
    new_coeffs = np.sqrt(probs[1:3])

    gd_param_history = [params]
    gd_cost_history = []

    start = time.time()
    energy = 10.0
    for n in range(max_iter):
        # Take a step in parameter space and record your energy
        params, prev_energy = opt.step_and_cost(cost_fn, params)
        prev_energy = energy
        # This keeps track of our energy for plotting at comparisons
        gd_param_history.append(params)
        gd_cost_history.append(prev_energy)
```

(continues on next page)

(continued from previous page)

```
        # Here we see what the energy of our system is with the new parameters
        energy = cost_fn(params)

        # Calculate difference between new and old energies
        conv = np.abs(energy - prev_energy)

        # Update hamiltonian with new coeffs
        probs = prob_circ(params)
        new_coeffs = np.sqrt(probs[1:3])
        coeffs = (1.0-mixing)*coeffs + mixing*new_coeffs

        H = update_ham(alpha,coeffs)

        cost_fn = qml.ExpvalCost(ansatz, H, dev)

        end = time.time()

        if (n % 1 == 0) and debug:
            print(
                "It = {:},  Energy = {:.8f},  Conv = {"
                ":.8f}, Time Elapsed = {:.3f} s".format(n, energy, conv,end-start)
            )
            start = time.time()


        if conv <= conv_tol:
            break

    if(debug): print("Final value of the energy = {:.8f}".format(energy))
    if(debug): print("Number of iterations = ", n)

    return params, energy, n, gd_cost_history
```

Up next is setting up our circuits! This is exactly from the *Quantum Computing RBM* example since we're not concerning ourselves with noise just yet.

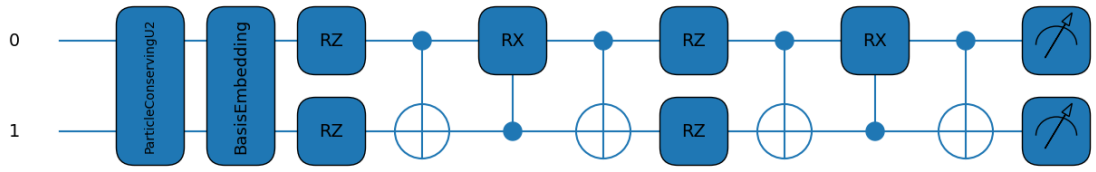
```
# Set the order you'd like to go to

dim = 2

# Setup the device. Here we'll use the default qubit just to get it working
dev = qml.device("default.qubit", wires=dim)

# Define a general ansatz for arbitrary numbers of dimensions

particles = 1

ref_state = qml.qchem.hf_state(particles, dim)

ansatz = partial(qml.ParticleConservingU2, init_state=ref_state)

layers = dim

# generate the prob function
@qml.qnode(dev)
```

(continues on next page)

(continued from previous page)

```
def prob_circuit(params):
    ansatz(params, wires=dev.wires)
    return qml.probs(wires=dev.wires)
```

```
(<Figure size 1300x300 with 1 Axes>, <Axes:>)
```



Now let's see what happens when running VQE for only the first element in the basis.

```
init_params = np.random.uniform(low=-np.pi / 2, high=np.pi / 2, size=qml.
↪ParticleConservingU2.shape(n_layers=layers, n_wires=dim))

update_ham(alpha_k,[1,0])

classical=np.linalg.eig(tildeH(alpha_k,[1,0]))
print("Result expected from classical computation: ", classical[0][0])
params, energy, n, _ = run_vqe_fixed(alpha_k,np.array([1,0]),init_params,dev)
print("Result from VQE: ",energy)
```

```
Result expected from classical computation:  3.391048364894118
Result from VQE:  3.391049270960056
```

Awesome! With our basis coefficients set to `[1,0]`, we get the same result as we expect from before. Now comes the fun part of iteratively solving the problem! There are two approaches we could take:

- "Standard" Iterative VQE | This method is the simplest conceptually and closely tracks the methods used to iteratively solve the problem on classical systems - namely you use the VQE to solve the optimal parameters for a fixed initial Hamiltonian, then you compute the coefficients of the basis and do it again. Between VQE runs you can mix the coefficients to ensure convergence, but ultimately you will obtain the optimal parameters and coefficients for the problem.
- "Dynamic" VQE | This method performs a measurement to determine the coefficients every step within the VQE algorithm and dynamically updates the Hamiltonian. This leads to faster convergence for this nonlinear problem and fewer steps overall.

We have implemented both methods in the notebook, and show both in the following cells. Feel free to play around with the parameters that tune the VQE algorithm and see what the effects are!

```
debug=False
classical=IterativeRBMsolver(alpha_k,1000)

sol_params, energy, n, _ = run_vqe_fixed(alpha_k,[1,0],init_params,dev)
probs = prob_circuit(sol_params)
new_coeffs = np.sqrt(probs[1:3])
coeffs = new_coeffs
```

(continues on next page)

(continued from previous page)

```
if(debug): print("Coefficients: ",coeffs)

mixing = 0.5

total_it = n
prev_energy = 10.0
for i in range(200):
    coeffs = (1.0-mixing)*coeffs + mixing*new_coeffs
    sol_params, energy, n, _ = run_vqe_fixed(alpha_k,coeffs,sol_params,dev)
    total_it += n
    if(abs(energy-prev_energy) < 1e-6):
        print("Convergence tolerance met!")
        break
    prev_energy = energy
    probs = prob_circuit(sol_params)
    new_coeffs = np.sqrt(probs[1:3])
    if(debug): print("Coefficients: ",coeffs)

print("Result from classical computation: ", classical[0][0])
print("Coefficients from classical computation: ",-classical[1][0][0],-
 ↪classical[1][0][1])
print("Result from standard VQE: ",energy)
print("Coefficients from standard VQE: ",new_coeffs[0],new_coeffs[1])
print("Total Iterations Required: ", total_it)
```

```
Convergence tolerance met!
Result from classical computation:  3.250220285551576
Coefficients from classical computation:  0.9913354984896011 0.13135421361484464
Result from standard VQE:  3.2502193216050816
Coefficients from standard VQE:  0.9913354828795378 0.13135433142456146
Total Iterations Required:  68
```

```
sol_params, energy, n, _ = run_vqe(alpha_k,np.array([1,0]),init_params,dev)

probs = prob_circuit(sol_params)
coeffs = np.sqrt(probs[1:3])
print("Result from classical computation: ", classical[0][0])
print("Coefficients from classical computation: ",-classical[1][0][0],-
 ↪classical[1][0][1])
print("Result from dynamic VQE: ",energy)
print("Coefficients from dynamic VQE: ",coeffs[0],coeffs[1])
print("Total Iterations Required: ", n)
```

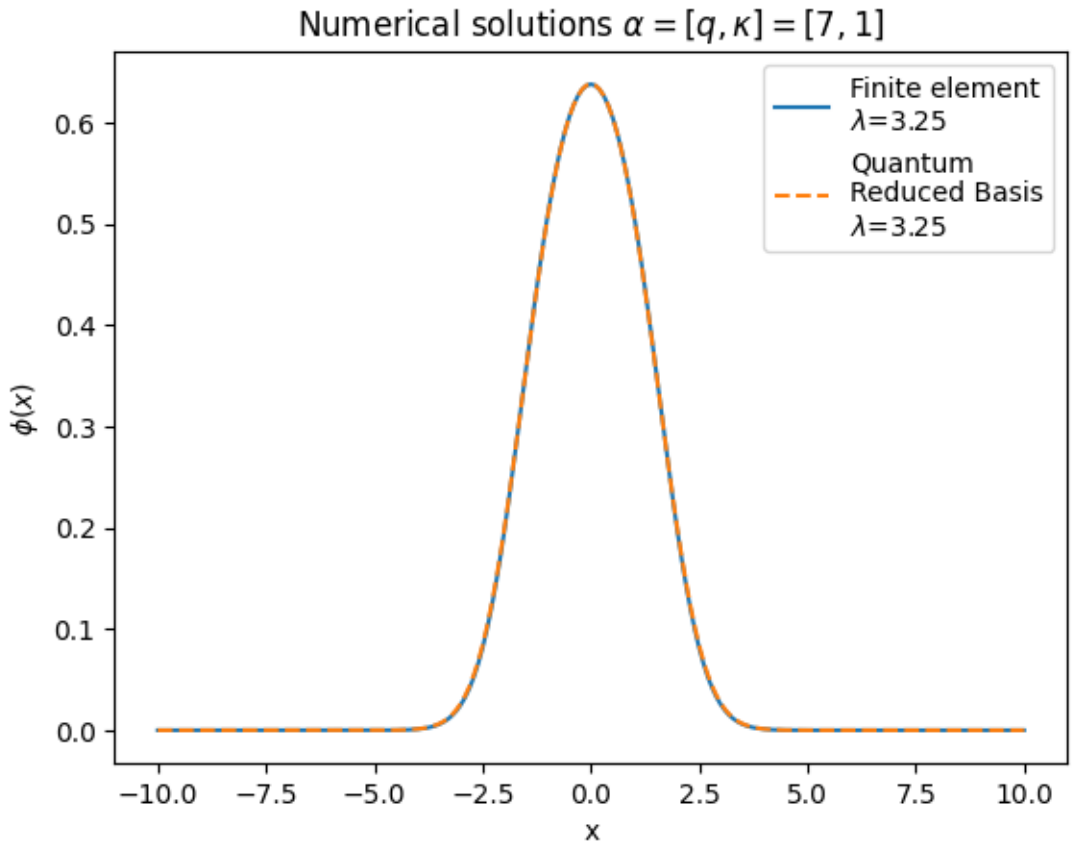
```
Result from classical computation:  3.250220285551576
Coefficients from classical computation:  0.9913354984896011 0.13135421361484464
Result from dynamic VQE:  3.2502198387574284
Coefficients from dynamic VQE:  0.9913355015975414 0.13135419015909147
Total Iterations Required:  23
```

```
solGaler=phibuilderFromCoeffs(coeffs)
lamGaler=solFull[0][0]

# Make plots of the numerical wavefunction
```

(continues on next page)

(continued from previous page)

```
fig, ax = plt.subplots(1,1)

ax.plot(x_array,solFinite,label=r'Finite element'"\n" '$\lambda$=
 ↪'+str(round(lamFinite,2)))
ax.plot(x_array,solGaler,label= r'Quantum' "\n" 'Reduced Basis'"\n" '$\lambda$=
 ↪'+str(round(lamGaler,2)),linestyle='dashed')

ax.set_title('Numerical solutions ' r'$\alpha=[q,\kappa]=[7,1]$')
plt.ylabel(r'$\phi(x)$')
plt.xlabel("x")
ax.legend()

plt.show()
```



They both perform really well, but the dynamic VQE requires less messy code to get it running, so that's what we'll use next! What we'll do now is add some noise to our circuit and see how that affects our performance.

```
#define some noise model to use in pennylane from qiskit.test.mock
from qiskit.test.mock import FakeLima
from qiskit.providers.aer.noise import NoiseModel
import mitiq as mq
from mitiq.zne.scaling import fold_global
from mitiq.zne.inference import RichardsonFactory
from pennylane.transforms import mitigate_with_zne
```

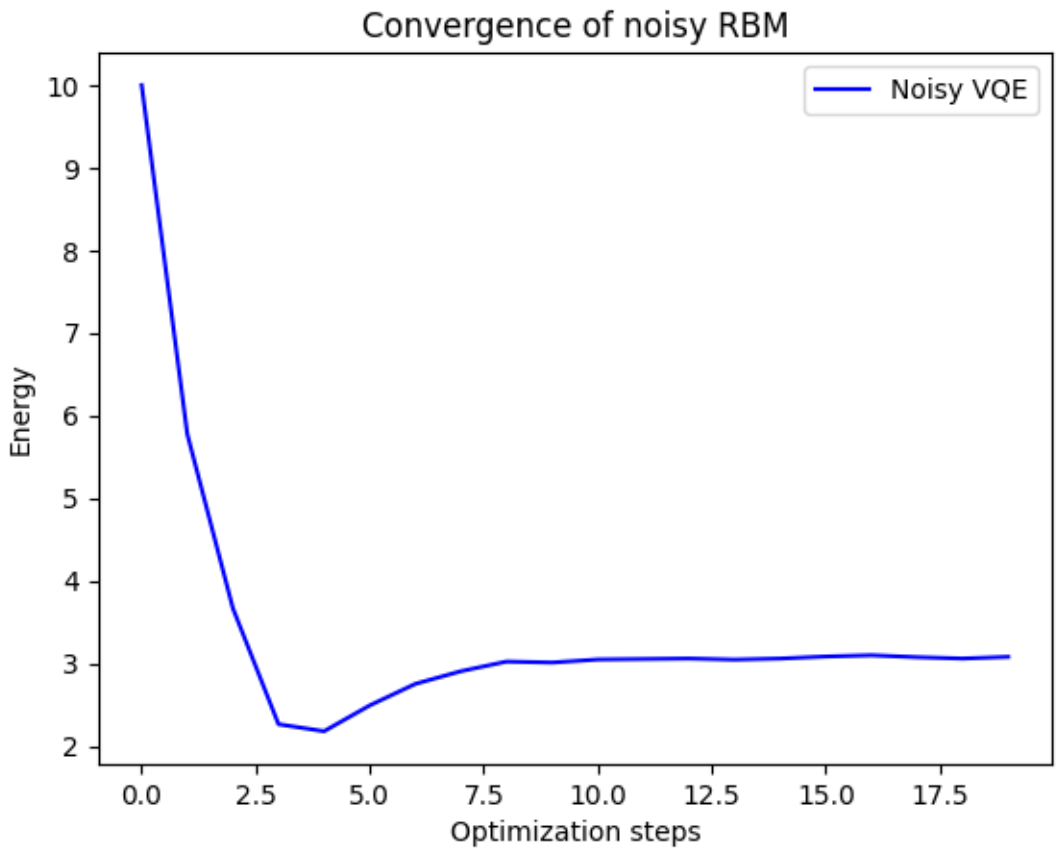
(continues on next page)

(continued from previous page)

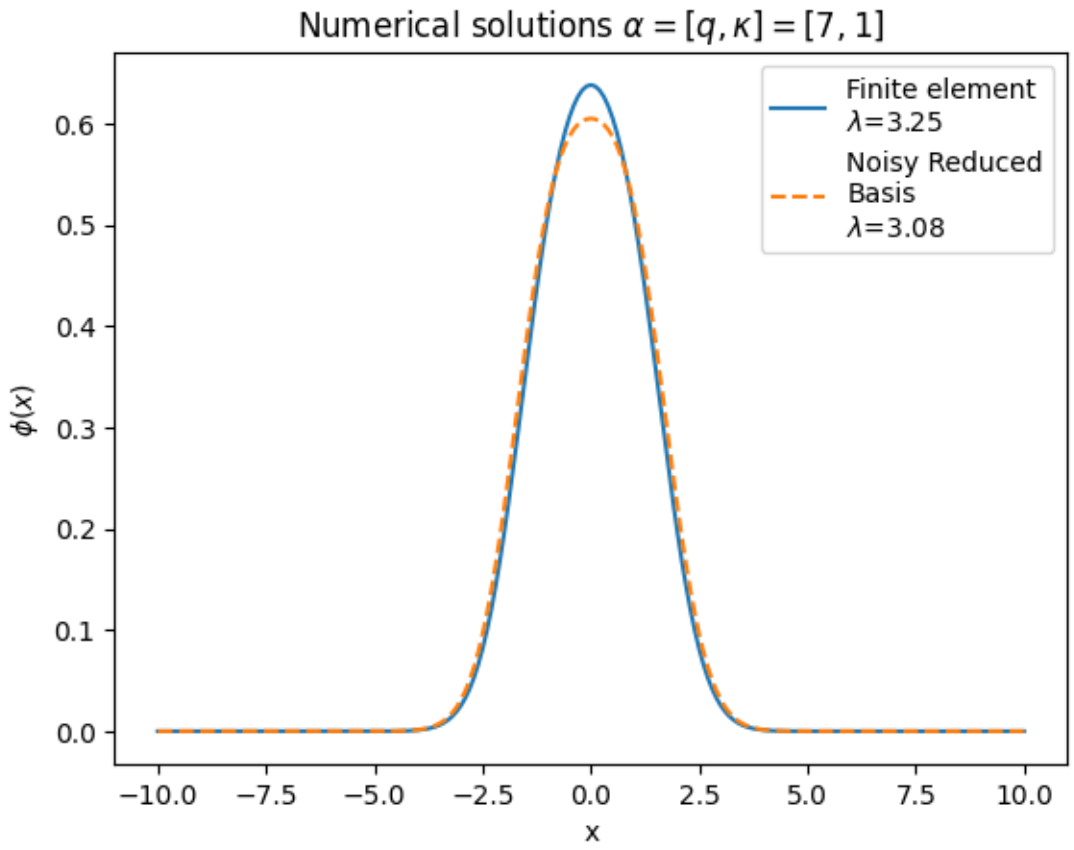
```
backend = FakeLima()
noise_model = NoiseModel.from_backend(backend)

#set up noisy device
dev_sim = qml.device("qiskit.aer", wires=2, noise_model=noise_model, optimization_
 ↪level=0, shots=10000)

# generate the prob function
@qml.qnode(dev_sim)
def noisy_prob_circuit(params):
    ansatz(params, wires=dev_sim.wires)
    return qml.probs(wires=dev_sim.wires)

sol_params, energy, n, energy_history = run_vqe(alpha_k,np.array([1,0]),init_params,
 ↪dev_sim,\
    max_iter=20,debug=False,noisy=True)

probs = noisy_prob_circuit(sol_params)
coeffs = np.sqrt(probs[1:3])

print("Result expected from classical computation: ", classical[0][0])
print("Coefficients expected from classical computation: ",-classical[1][0][0],-
 ↪classical[1][0][1])
print("Result from noisy VQE: ",energy)
print("Coefficients from noisy VQE: ",coeffs[0],coeffs[1])
print("Total Iterations Required: ", n)
```

```
Result expected from classical computation:  3.250220285551576
Coefficients expected from classical computation:  0.9913354984896011 0.
 ↪13135421361484464
Result from noisy VQE:  3.0800371851346857
Coefficients from noisy VQE:  0.9394679345246436 0.17175564037317667
Total Iterations Required:  19
```

```
plt.plot(energy_history, "b", label="Noisy VQE")

plt.ylabel("Energy")
plt.xlabel("Optimization steps")
plt.title("Convergence of noisy RBM")
plt.legend()
plt.show()
```




```
solGaler=phibuilderFromCoeffs(coeffs)
lamGaler=energy

# Make plots of the numerical wavefunction
fig, ax = plt.subplots(1,1)

ax.plot(x_array,solFinite,label=r'Finite element'"\n" '$\lambda$=
 ↪'+str(round(lamFinite,2)))
ax.plot(x_array,solGaler,label= r'Noisy Reduced' "\n" 'Basis'"\n" '$\lambda$=
 ↪'+str(round(lamGaler,2)),linestyle='dashed')



ax.set_title('Numerical solutions ' r'$\alpha=[q,\kappa]=[7,1]$')
plt.ylabel(r'$\phi(x)$')
plt.xlabel("x")
ax.legend(loc="upper right")

plt.show()
```



Even with a noisy circuit it doesn't do terribly! It's likely we let it run a little too long, in fact, as our standard convergence criteria isn't well suited to a noisy system. Can you think of any ways to better measure convergence in these cases? Give it a try!

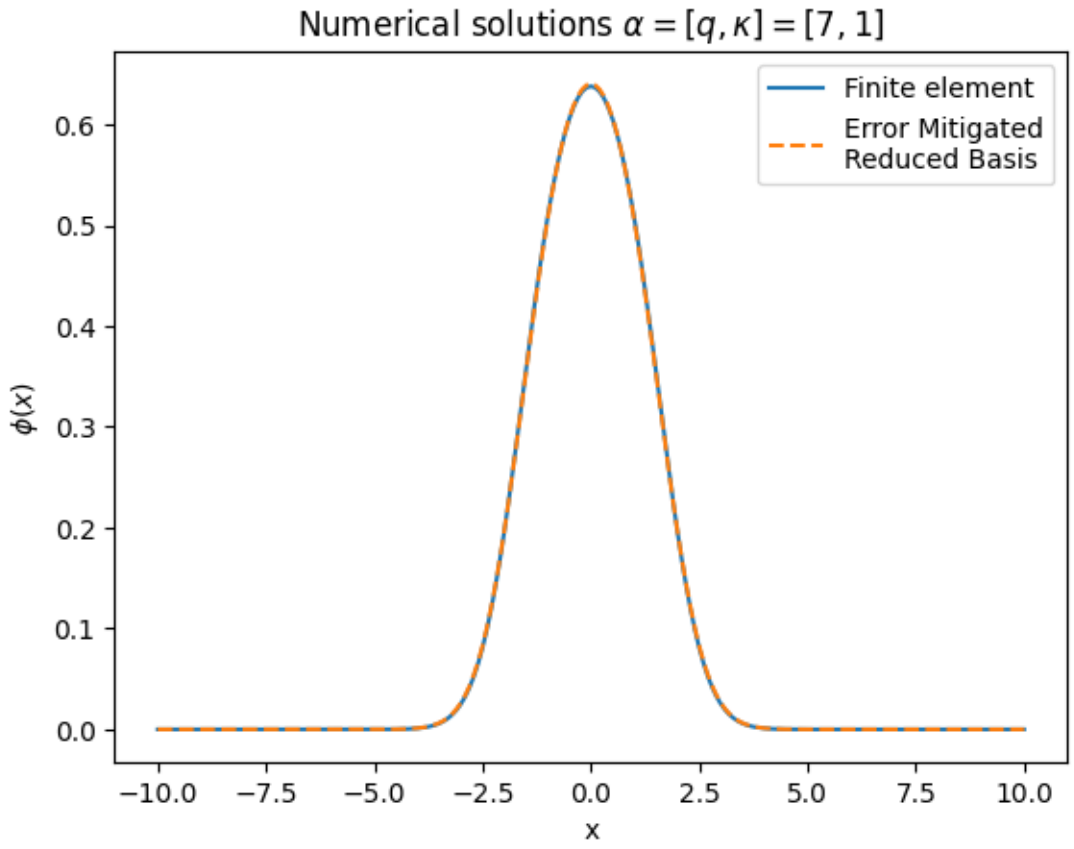
One last thing we can do to get slightly better performance is to leverage zero-noise extrapolation for the coefficients and see if there is a noticeable improvement.

To achieve this we'll first define an extrapolation function to parse the output from our probability function and setup the error mitigated qnode. Then we can run it with three scale factors and extrapolate our coefficients.

Finally, we'll plot the error mitigated solution and see if this was all worth doing!

```
def prob_extrapolate(scale_factors, results, **extrapolate_kwargs):
    prob1 = RichardsonFactory.extrapolate(scale_factors,results[:,0,1])
    prob2 = RichardsonFactory.extrapolate(scale_factors,results[:,0,2])
    return np.array([prob1,prob2])

#set up scale factors to use
scale_factors = [1, 2, 3, 4]

sim_qnode = qml.QNode(noisy_prob_circuit, dev_sim)

#use ZNE to mitigate error
mitigated_qnode = mitigate_with_zne(scale_factors, fold_global, prob_extrapolate)(sim_
 ↪qnode)
zne_result = mitigated_qnode(sol_params,shots=2**14)

mitig_coeffs = np.sqrt(zne_result)
```

```
solGaler=phibuilderFromCoeffs(mitig_coeffs)
lamGaler=solFull[0][0]

# Make plots of the numerical wavefunction
fig, ax = plt.subplots(1,1)

ax.plot(x_array,solFinite,label=r'Finite element')
ax.plot(x_array,solGaler,label= r'Error Mitigated' "\n" 'Reduced Basis',linestyle=
↪'dashed')



ax.set_title('Numerical solutions ' r'$\alpha=[q,\kappa]=[7,1]$')
plt.ylabel(r'$\phi(x)$')
plt.xlabel("x")
ax.legend(loc="upper right")


plt.show()
```



Yet another beautiful result! Compared to the vanilla noisy result a few cells up, this has much better agreement with the 'true' solution provided by the finite element method. And remember, this is the noisy result from a basis of N=2!

I encourage you to try how well we can solve the Gross-Pitaevskii equation with different values of the model parameters, $\alpha = \kappa, q$. While we haven't picked something in our basis, it's possible we just got lucky with the values we chose. Indeed, if you vary the parameters well outside the true solutions that inform the basis, you might start to see big deviations. But these deviations will manifest in the classical computing calculations too, so it's not a computational limit but an

informational one.

If you want one final challenge, you can try running this on real hardware! The required cell is below, but be warned, this will take a long time to run. You may want to instead use the qiskit VQE runner that we used in the *noise chapter*, as this gets priority when running and had built in mitigation. The only thing you'll need to add is your API token in `token` and choose which machine you want to submit to - a good strategy is finding which one has the lightest queue.

Other than that last charge, this brings us to the end of the final challenge problem for our expansive tutorial series on quantum computing applications to many-body problems. The authors had a great time putting this all together (*see Fig. 1*), so we hope you enjoyed going through it :). Come meet our final thoughts and the thanks we want to give to key people in the next and final section of Conclusions and Acknowledgements.

```
token = "XXX"

dev_ibm = qml.device('qiskit.ibmq', wires=2, backend='ibm_nairobi', ibmqx_token=token)

init_params = np.array([0.6,])

def ansatz(params, wires):
    t0 = params[0]
    qml.PauliX(wires=0)
    qml.RY(t0, wires=1)
    qml.CNOT(wires=[1,0])


# generate the prob function
@qml.qnode(dev_ibm)
def noisy_prob_circuit(params):
    ansatz(params, wires=dev_ibm.wires)
    return qml.probs(wires=dev_ibm.wires)

sol_params, energy, n, energy_history = run_vqe(alpha_k,np.array([1,0]),init_params,
 ↪dev_ibm,\
    max_iter=10,debug=True,noisy=True)

probs = noisy_prob_circuit(sol_params)
coeffs = np.sqrt(probs[1:3])

print("Result expected from classical computation: ", classical[0][0])
print("Coefficients expected from classical computation: ",-classical[1][0][0],-
 ↪classical[1][0][1])
print("Result from noisy VQE: ",energy)
print("Coefficients from noisy VQE: ",coeffs[0],coeffs[1])
print("Total Iterations Required: ", n)
```

CHAPTER

SIX

CONCLUSIONS AND ACKNOWLEDGEMENTS

Through this Jupyter book we have gone through various fundamental concepts and tools needed for implementing nuclear model calculations on quantum computers. We explained and developed the Variational Quantum Eigensolver in the first chapter, showcasing an implementation of the deuteron with additional improvements for speeding convergence. In chapter two we went through the intrinsic noise associated with quantum circuits, and addressed ways of dealing with it through the Zero-Noise Extrapolation algorithm. In chapter three we explored the Reduced Basis Method, a dimensionality reduction technique capable of transforming problems involving thousands of degrees of freedom into just a few interacting "quasiparticles". This is a crucial step into translating calculations for quantum computers with access to just few qubits. Finally, in the last chapter we added everything together to tackle the Gross-Pitaevskii equation, a simple nonlinear model for describing dilute Bose-Einstein condensates.

With all these tools in place, you, the reader, now have the ability to implement your own calculations onto a quantum computer. Following a similar workflow as the one we developed in the last chapter, you could find a reduced order model for your system (perhaps using the reduced basis method), which could then be mapped onto a quantum circuit for use with the Variational Quantum Eigensolver algorithm. Don't forget to also use some noise mitigation technique such as the Zero Noise Extrapolation if you are dealing with a noisy circuit. Be sure to let us know if you apply anything you've learned from this book to your own problems of interest and share your successes on this journey! If you'd like, we could even explore adding it to the book - so send any comments or questions to Kyle via `qc at kyle dot ee`.

We will like to thank all of the organizers of the 2022 FRIB Theory Alliance Summer School on Quantum Computing and Nuclear Few- and Many-Body Problems for the incredible opportunity of learning about all these topics from world experts, as well as for raising a "final challenge" that lead to the construction of this Jupyter book.

For the topics found in this book in particular, we would like to thank:

- Ben Hall and Morten Hjorth-Jensen for a fantastic explanation of the Variational Quantum Eigensolver
- Ryan Larose for another fantastic explanation on the Zero Noise Extrapolation algorithm
- Dean Lee for great discussions about the Gross-Pitaevskii equation

Fig. 6.1: Celebratory ice cream the day after the summer school!

CHAPTER

SEVEN

CONTRIBUTORS

- Jingyi Li | Content
- Alexandra Semposki | Content
- Pablo Giuliani | Content
- Kyle Godbey | Content, Webmaster

Many thanks to everyone that has contributed in some way to this living book! If you'd like to add anything or have any questions, contact the team at:

qc <@t> kyle.ee