Estimation of Ro-Vibrational Eigenphases Using Bayesian Quantum Phase Estimation

STAT 447C Final Project: Ethan Rajkumar, #55024616

Introduction

To analyze molecular characteristics, chemists often solve the following eigenvalue equation:

$$\mathbf{H}_{RV}\Psi(\vec{\theta}) = E\Psi(\vec{\theta}) \tag{1}$$

In this equation, $\mathbf{H}_{RV}$represents the Hamiltonian that accounts for the molecule's total ro-vibrational energy. The wavefunction Ψ, parameterized by the vector $\vec{\theta}$, serves as the eigenvector. The energy level E corresponds to the eigenvalue which represents the energy levels electrons can occupy. While simpler molecules like H_2 allow for straightforward single vector decompositions, larger molecules present computational challenges due to the increased size of $\mathbf{H}_{RV}$. This project aims to address these challenges by using quantum phase estimation to estimate a posterior distribution, which can then be used to estimate the eigenvalues of $\mathbf{H}_{RV}$ for the Cr_2 molecule.

Literature Review

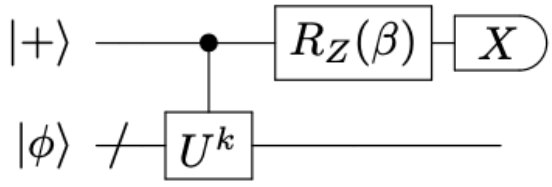
Taking the ro-vibrational Hamiltonian $\mathbf{H}_{RV}$ and performing the following operation to form a matrix $\mathcal{U}$ gives $\mathcal{U} = e^{i\mathbf{H}_{RV}t}$. The expression above allows for the application of the phase estimation algorithm, which estimates the eigenvalues of the unitary operators. The algorithm then uses these eigenvalues to approximate the eigenvalues of the original Hamiltonian. A matrix is denoted to be unitary if it follows the spectral theorem which is listed below.

> **Spectral Theorem:** Let U be a normalized $K \times K$ complex matrix. There exists an orthonormal basis of K-dimensional complex vectors $\{|\psi_1\rangle, \dots, |\psi_K\rangle\}$, along with complex numbers $\lambda_1, \dots, \lambda_K$, such that $U = \lambda_1|\psi_1\rangle\langle\psi_1| + \cdots + \lambda_K|\psi_K\rangle\langle\psi_K|$. This matrix U can be diagonalized in an orthonormal basis consisting of its eigenvectors, with the corresponding eigenvalues on the diagonal.

The main implementation of the phase estimation algorithm is shown below:

> **Quantum Phase Estimation Algorithm:**
>
> - **Input:** An n-qubit quantum state $|\psi\rangle$ and a unitary quantum circuit for an n-qubit operation U.
> - **Promise/Assumptions:** $|\psi\rangle$ is an eigenvector of U.
> - **Output:** An approximation to the number $\theta \in [0, 1)$ satisfying $U|\psi\rangle = e^{2\pi i\theta}|\psi\rangle$.

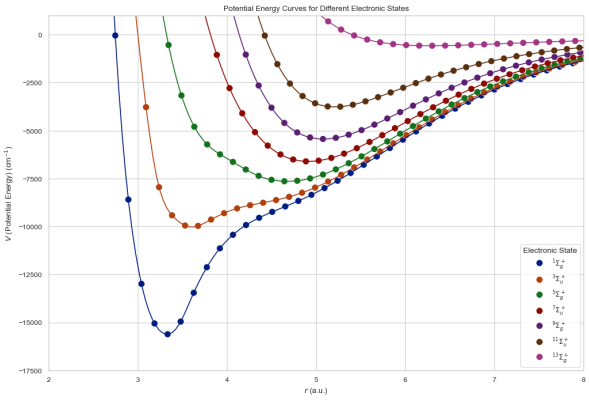
Quantum phase estimation can also take a probabilistic approach in the form of Bayesian optimization. Termed Bayesian Quantum Phase Estimation by Weibe and Granade in 2016[1]. It was made efficient in a highly noisy environment by Yamamoto et al. (2024)[2]. This method uses a quantum circuit to represent the posterior distribution of the phase estimation algorithm[1,2]. For a two-qubit circuit, the authors used a quantum circuit which is an acyclic network of quantum gates connected by wires. The quantum gates are matrices that represent quantum operations while the wires represent the **qubits** (see appendix) on which the gates act. The pictorial representation of the circuit is shown in Figure 1 (below).



The quantum circuit is parameterized by $k \in \mathbb{N}$ and $\beta \in [0, 1]$. The circuit is measured in the Pauli X basis and the measurement is denoted as a probability of states (or a likelihood) in the computational basis such that $p(m|\phi, k, \beta) = \frac{1+\cos(k\phi+\beta)}{2}$ where m is the measurement outcome in the computational basis, ϕ is the unknown where $p(\phi) \sim \text{Unif}[0, 2\pi)$, and RZ is a rotational matrix described in the appendix below. With R possible measurement outcomes and a high number of iterations, a posterior distribution can be computed such that $p(\phi|m, k, \beta) \propto p(\phi) \cdot \Pi_{r=1}^{R} p_r(m|\phi, k, \beta)$. Once converged, the probability distributions are then used to compute a posterior mean with respect to an observable or $\sum_R p(\phi|m, k, \beta)\langle R|\hat{H}|R\rangle$. Note that this work does not compute a posterior mean but instead creates the probabilities that will estimate the eigenvalues of the Hamiltonian matrix.

Problem Formulation

While Asnaashari et al. employed a hybrid quantum-classical computing approach to address an eigenvalue problem, using a greedy induced point sampling algorithm to compute an expectation in their respective basis, they encountered significant scalability issues related to the time complexity of quantum circuit generation, denoted by $\mathcal{O}(\sum_k n \times M_k)$, where M_k represents the time required to generate an expectation value per iteration over n samples, and k denotes the iteration index[3]. Yamamoto and Weibe attempted to use quantum phase estimation of a different type of Hamiltonian to compute the eigenphase/eigenvalues in the computational basis[2,1]. However, their method only uses up to 2 qubits, required 920 gates and is not scalable to a higher amount of qubits. The key challenge that this project aims to address is to create an algorithm that has a minimum of 5 qubits while also having a lower gate count (12) while also being accurate. The project algorithm, which is based on the Bayesian Quantum Phase Estimation algorithm, will be used to estimate the eigenvalues of the Hamiltonian matrix for the dichromium molecule. Further literature review showed that very few papers have been published on the topic and that the Bayesian Quantum Phase Estimation algorithm has not been used to estimate the eigenvalues of the Hamiltonian matrix for the Cr_2 molecule.



Data Preprocessing and Generation of Unitary Matrices

The data used in this project was generated by Asnaashari and is a discrete variable representation of the Hamiltonian[3]. First, an interpotential curve (measures the Coulombic interaction of two atoms) was generated from Fortran for the Cr_2 molecule (graph on the right). Afterwards, a Hamiltonian matrix was selected from the 32 points collected from the curve for each spin state. A combination of the `pennylane` and `phayes` Python 3.11.7 libraries were used to transform the Hamiltonian matrix into a unitary matrix as described in the literature review. The unitary matrix was then used to generate the quantum circuit by applying a transform:

$$U = e^{i\mathbf{H}_{RV}t} \tag{2}$$

The quantum circuit was then used to estimate the phases, ϕ, of each unitary matrix. The phase estimation algorithm itself can be described by the probabilistic model below.

Model

The model used in this project is a Bayesian model that estimates the phases of the unitary matrices. The model is parameterized by the number of qubits, n, and the number of iterations, R. The model is defined as follows (using the $\sim$ notation):

$$\begin{aligned}
&\mathcal{L} = (m|\phi, \beta, k) \sim \text{Fourier Distribution} \\
&\phi \sim \text{Unif}(0, 2\pi) \\
&\beta \sim \text{Unif}(0, 2\pi) \\
&k \sim \mathcal{N}(\alpha, \sqrt{1/\sigma^2}) \\
&\alpha \sim \text{Unif}[J, J_{max}] \text{ where } J \text{ and } J_{max} \in \mathbb{N}
\end{aligned}$$

The quantum circuit provided in the literature review is used to estimate the phases of the unitary matrices. Note that we are trying to estimate ϕ and the error compared to the true value, ϕ_0 which is the angle(s) or phase that we are trying to estimate. The error is denoted as $\epsilon = \phi - \phi_0$. The error is then used to estimate the eigenvalues of the Hamiltonian matrix. Note that the phases will be estimated for multiple spin states of dichromium gas. The likelihood and prior are defined below:

- **Likelihood**: This is defined by the quantum circuit as given by the first figure above. It is parameterized with $k \in \mathbb{N}$ and $\beta \in [0, 2\pi)$.

$$\begin{aligned}
&\mathbb{P}(m|\phi, k, \beta) = \frac{1 + \cos(k\phi + \beta)}{2} \text{ decomposition into the product form gives,} \\
&\mathbb{P}(m|\phi, k, \beta) = \Pi_{r=1}^{R} p(m_r|\phi, k, \beta)
\end{aligned}$$

- **Prior** This is the probability of phi or equivalently ϕ

$$\mathbb{P}(\phi) \sim \text{Unif}[0, 2\pi)$$

Thus the posterior that is being is estimated is described by:

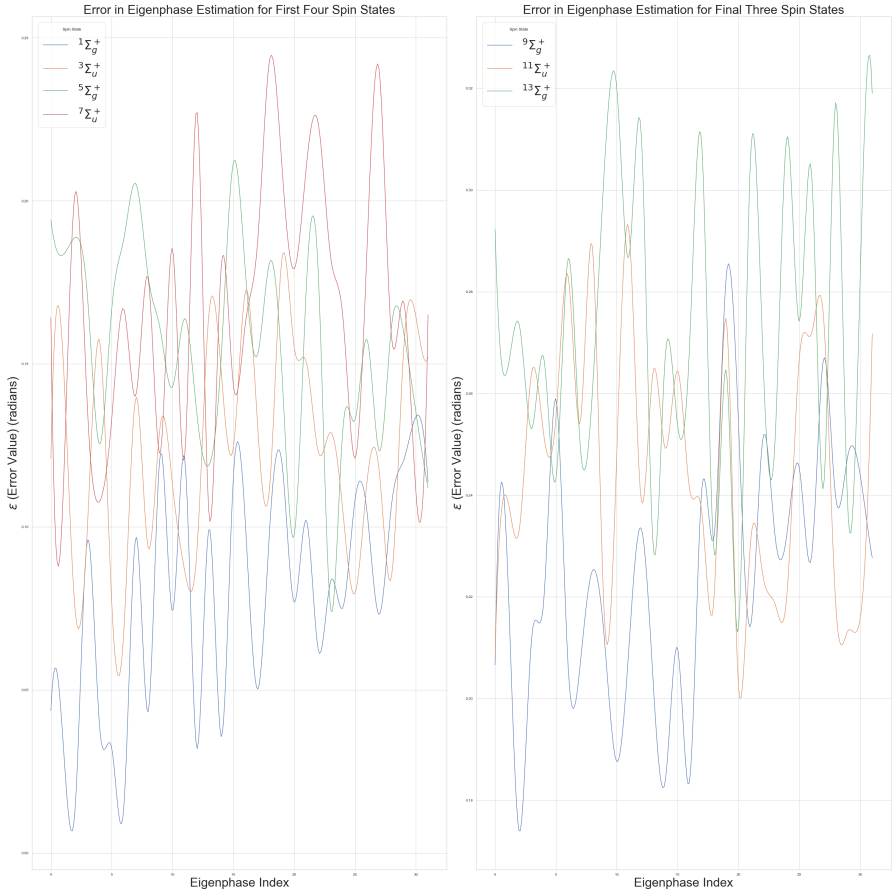
$$\mathbb{P}(\phi|m, k, \beta) \propto \mathbb{P}(\phi) \cdot \Pi_{r=1}^{R} \mathbb{P}(m_r|\phi, k, \beta)$$

The posterior is then used to estimate the angle or phase of the unitary matrix by computation of a posterior mean for the phase. The evaluation metric used for this model is $\epsilon = |\phi - \phi_0|$. This particular error metric is used as the model and the true phases are evaluated in the same basis for each spin state.

Results

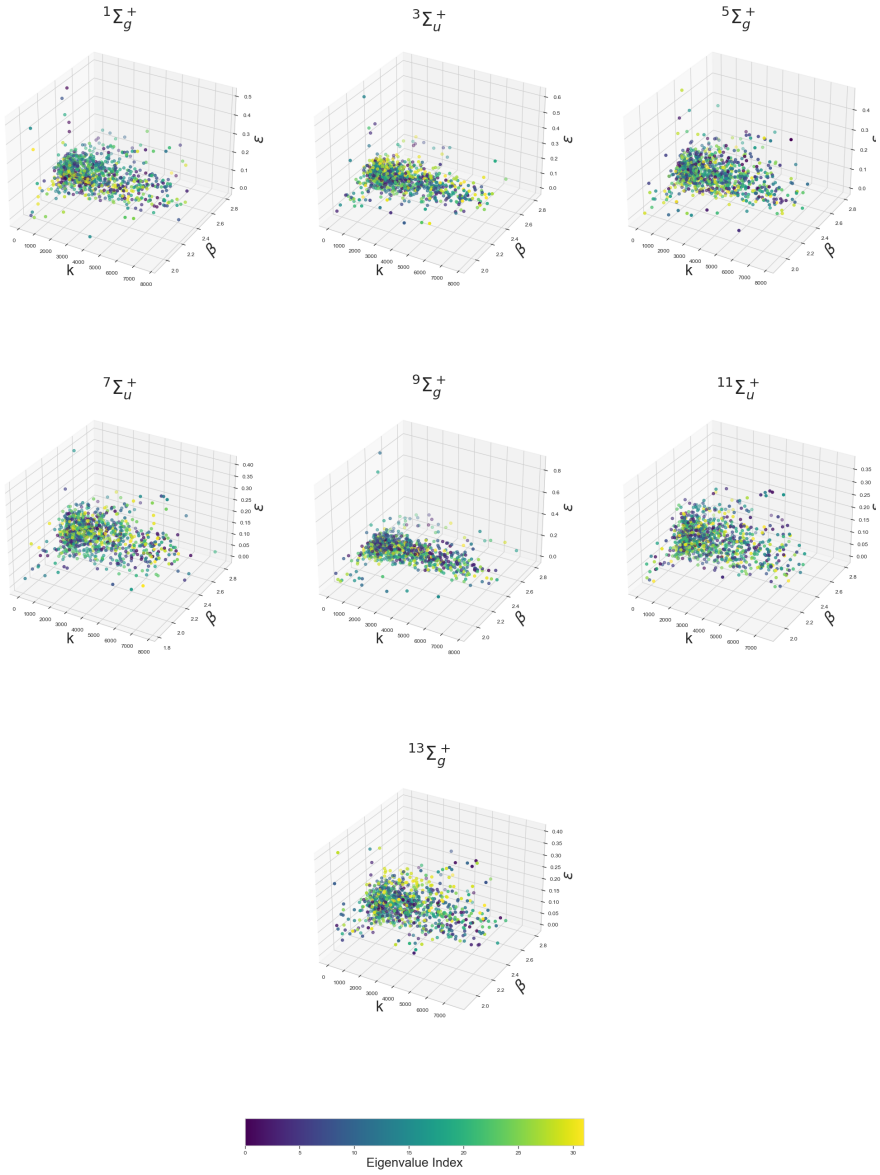
The graph below shows the ϵ values for each of the unitary matrices for all of the spin states. The errors was obtained by taking the synthetic unitary data (100 experiments) and subtracting the true value (for each eigenvalue) from the result. Afterwards, each result was graphed on a cubic spline with an offset of 0.04 radians. This is so the results for all of the spin states can be easily visualized.

Figure 1: Phase Error For Each Spin State (with respect to the Index of the Eigenphase)



The graph indicates that error rates are somewhat low for the 2D graph. The error manifests a sinusoidal pattern with respect to the eigenphase index, which is not uniform across the various spin states. Specifically, the error for the initial spin state ($^1\Sigma_g^+$) averages significantly lower in comparison to that of the subsequent spin states. There is a notable increase in error when progressing from the first to the second spin state. The seventh and thirteenth spin states exhibit the most pronounced errors, which coincide with the greatest fluctuations in the variable ϵ. To determine the extent to which k and β values affect the error, 3D graphs were created to show the relationship between the error and the k and β values.

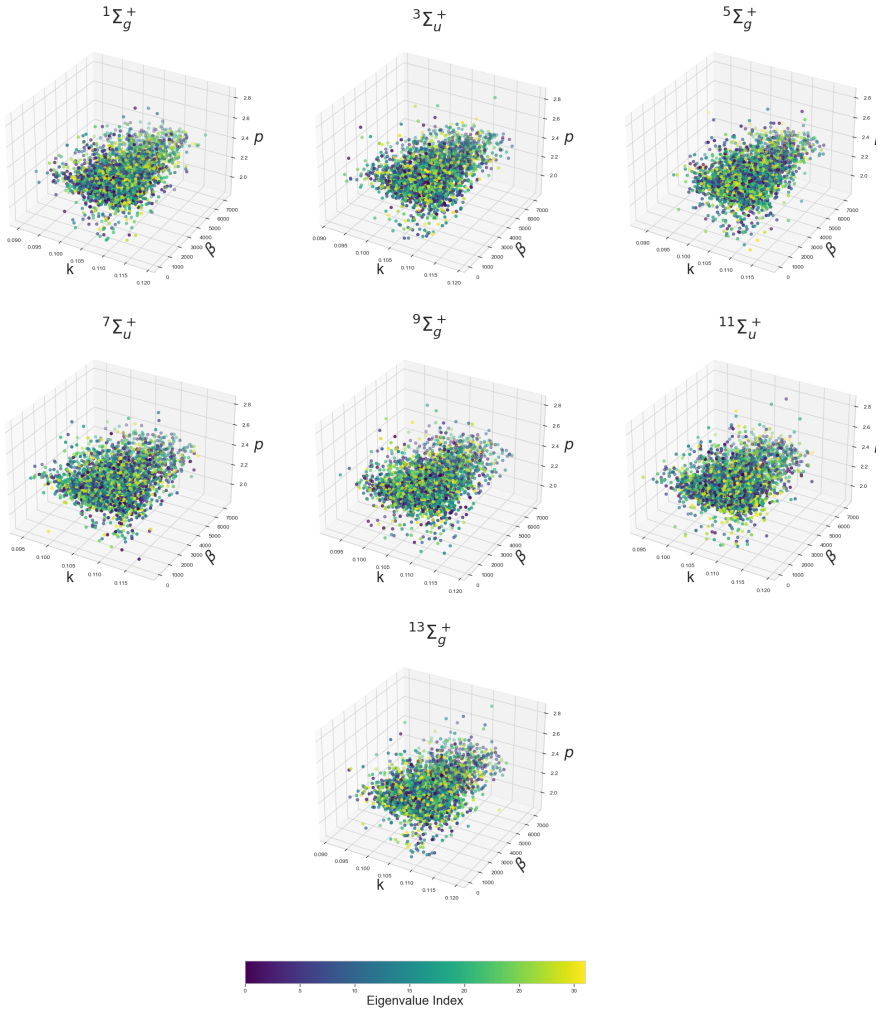
Figure 2: Phase Error For Each Spin State



The 3D plots show a pronounced clustering of data points within specific areas for all states, indicative of a localized aggregation of eigenvalues and possibly contributing to the consistently lower and more predictable error rates for this state. The compactness of these clusters suggests stability in the eigenvalues relative to changes in the parameters k and β. Conversely, the $7^1\Sigma_u^+$ and $13^1\Sigma_g^+$ states demonstrate a wider spread of data points across the 3D space. This greater spread could be symptomatic of higher and more variable error rates, hinting at less stability or precision in these states. The scattered distribution points to a more pronounced variation in eigenvalues in response to k and β. Additionally, the significant overlap of data at elevated k and β values suggests a closeness or even degeneracy in the eigenvalues, further compounding the variability of errors observed. However the graph above suggests that model misspecification is highly unlikely as the error rates are relatively low and the model is robust to the assumptions it holds. The error with respect to the hyperparameters k and β (for all eigenvalues) occupys a approximately normal distribution indicating that it can be improved by adding a constant to the phase estimation.

Finally, a posterior distribution was obtained to determine the distributions of the phases given the hyperparameters, k and β.

Figure 3: Posterior Distribution For Each Spin State



The posterior can be described as an approximately slanted round sheet clustered.The tight clustering of points within the graph suggests a higher certainty about the phase ϕ for the corresponding eigenvalue index, whereas a wider spread of points is indicative of greater uncertainty. When correlated with the results from the 3D phase error graph, it is suggested that lower k values and lower β are associated with higher conditional probabilities of a phase and a greater phase error when compared to the true phase value. A sensitivity analysis could not be performed as the true distribution of the phases is unknown.

Discussion

The results of this study demonstrate that the extrapolation behavior of the model is proficient at determining the phases for a dichromium molecule, with a low error rate observed. The posterior distribution tends to follow a normal distribution across all eigenvalues and eigenvectors, in relation to the hyperparameters k and β. Consequently, these phases can be transformed to estimate the eigenvalues of the Hamiltonian, which in turn correspond to the discrete energy levels accessible to electrons.

However, the model is subject to certain limitations. Its stochastic nature, particularly when it is assumed to be time-independent, introduces variability that can lead to different simulation outcomes despite identical initial conditions, complicating the reproducibility of results and precise behavior prediction. Additionally, as the size of the molecule increases, the assumptions integral to the stochastic model may not remain valid across all scenarios. The inherent randomness of k and β may not accurately reflect the molecule's actual physical attributes, leading to possible model misspecification. Computational issues, such as underflow, present further challenges; as the molecular size escalates, the number of computational basis states expands exponentially. This is predicated on the assumption that the molecule may only uniformly occupy each state, which could amplify the model's sensitivity to initial conditions. Lastly, inherent to the definition of stochastic models is the potential oversight of quantum entanglement (the superposition of states)—an essential aspect of quantum mechanics that, if not accurately represented, could induce significant errors in the modeling of the posterior of ϕ^2.

Conclusion and Future Directions

Despite these limitations, the model does offer a flexible and interpretable approach to Quantum Phase Estimation. By showing a high degree of accuracy for ϕ, the model can be used to estimate the eigenvalues of the Hamiltonian matrix for the Cr_2 molecule. Future work could focus on improving the model's scalability and robustness to larger molecules such as ArHCl. This could involve the development of a more efficient quantum circuit that can handle a higher number of qubits, as well as the implementation of a more sophisticated Bayesian optimization algorithm to improve the model's accuracy and precision. Additionally, the model could be extended to include more complex quantum operations and gates, enhancing its capabilities and applicability to a wider range of molecular systems. Finally the model could be used in other applications than chemistry such as electronics engineering (phases for batteries).

By addressing these challenges and limitations, the model could become a valuable tool for chemists, physicists and engineering seeking to analyze the ro-vibrational properties of complex molecules and materials.

References

[1] Wiebe, N.; Granade, C. Efficient Bayesian Phase Estimation. *Phys. Rev. Lett.* **2016**, *117*, 010503.

[2] Yamamoto, K.; Duffield, S.; Kikuchi, Y.; Muñoz Ramo, D. Demonstrating Bayesian quantum phase estimation with quantum error detection. *Phys. Rev. Res.* **2024**, *6*, 013221.

[3] Asnaashari, K.; Krems, R. V. Compact quantum circuits for variational calculations of ro-vibrational energy levels of molecules on a quantum computer. *arXiv.org* **2023**,

Acknowledgements

I would like to thank Professor Roman Krems and Kasra Asnaashari for the data used, their guidance and support throughout this project. I would also like to thank the University of British Columbia for providing the resources necessary to complete this project.

Appendix

A: Glossary of Some Quantum Computing Terms

Bra A row vector defined by:

$$\langle\psi| = [\psi_1^\dagger, \psi_2^\dagger, ..., \psi_K^\dagger] \tag{3}$$

where $\psi^\dagger$ is the complex conjugate of ψ.

Ket A column vector defined by:

$$|\psi\rangle = \left[\psi_1, \psi_2, ..., \psi_K\right]^T$$

Qubit Represented by a complex-valued, 2×1 vector. It can be written as a linear combination of $|0\rangle$ and $|1\rangle$ as follows:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$$

where α and β are complex numbers, and $|\alpha|^2 + |\beta|^2 = 1$. Note that

$$|0\rangle = [1, 0]^T, |1\rangle = [0, 1]^T \tag{4}$$

One does not observe α and β. Instead $|\alpha|^2$ and $|\beta|^2$ are the coefficients that we observe. Each of $|\alpha|^2$ and $|\beta|^2$ and beta correspond to the probability of collapsing to a specific state.

Computational Basis States Refers to the standard orthonormal basis for the state space of n qubits and represents a complete set of orthonormal vectors for this space. For a single qubit, the computational basis consists of the two states $|0\rangle$ and $|1\rangle$. Each computational basis state for a system of n qubits can be described as a tensor product of n single-qubit states, where each qubit is independently in either $|0\rangle$ or $|1\rangle$. The computational basis states form an orthonormal basis, which implies that they are mutually orthogonal and each has unit norm. The completeness of this basis set means that any state $|\psi\rangle$ of the system can be uniquely expressed as a linear combination of these basis states, and no other states need to be added to fully describe the state space. Each basis state corresponds to a unique binary string of length n, which is particularly useful for encoding and manipulating information in quantum computation. The general form of a computational basis state for an n-qubit system is:

$$|i_1 i_2 \dots i_n\rangle = |i_1\rangle \otimes |i_2\rangle \otimes \cdots \otimes |i_n\rangle$$

where i_k is either 0 or 1, representing the state of the k-th qubit. For example, in a two-qubit system, the computational basis states are $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$, corresponding to the vector representations $[1,0,0,0]^T$, $[0,1,0,0]^T$, $[0,0,1,0]^T$, and $[0,0,0,1]^T$ respectively. Note that $|00\rangle = |1\rangle$, $|01\rangle = |2\rangle$, $|10\rangle = |3\rangle$, and $|11\rangle = |4\rangle$. This is because the number of computational basis states (k) is proportional to the number of qubits (N) such that

$$k = 2^N \tag{5}$$

Hardware Basis States Refers to the set of gates or operations that a quantum computer can physically interpret and execute. The hardware basis

B: List of Quantum Gates Used in this Work

Pauli-Z Gate (Z)

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

RZ Gate

$$R_{z,x_k}(\theta) = \begin{bmatrix} e^{-i\theta \cdot x_k} & 0 \\ 0 & e^{i\theta \cdot x_k} \end{bmatrix}$$

C: Python Implementation

Authors: Ethan Rajkumar

Last Modified: Sunday April 21st, 2024 by Ethan Rajkumar

In [ ]:
```
#importing the necessary libraries
import numpy as np
from lib import *
import pandas as pd
import pennylane as qml
from lib.cr2dataset import hartree, get_pot_cr2
from lib.cr2dataset import cr2_params
from lib.qpe import QPE
import jax
from jax import random
from lib.phase_estimation import *
import phayes
import jax.numpy as jnp
import random as r
import seaborn as sns
import matplotlib.pyplot as plt
from scipy.interpolate import make_interp_spline
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
```

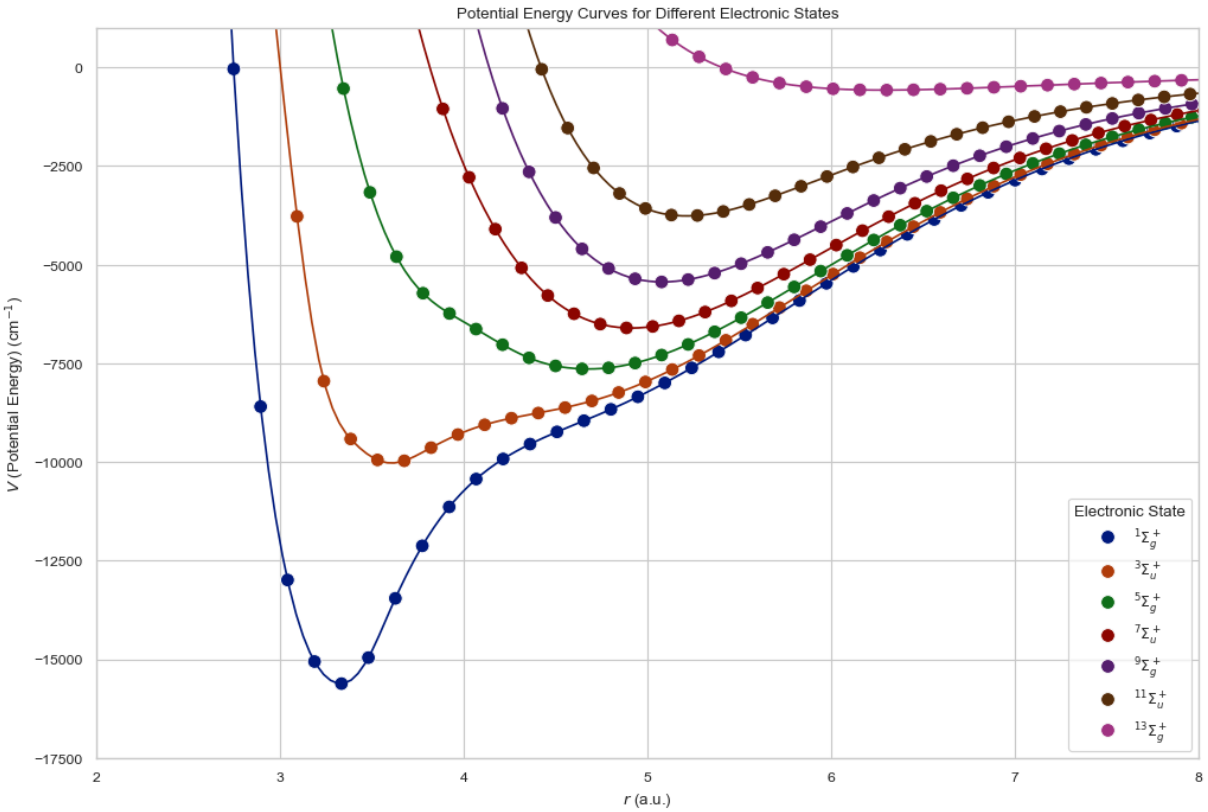
```
/var/folders/k9/jxn4k2_s1v50xzyqggp1kft40000gn/T/ipykernel_85851/2955796516.
py:4: DeprecationWarning:
Pyarrow will become a required dependency of pandas in the next major releas
e of pandas (pandas 3.0),
(to allow more performant data types, such as the Arrow string type, and bet
ter interoperability with other libraries)
but was not found to be installed on your system.
If this would cause problems for you,
please provide us feedback at https://github.com/pandas-dev/pandas/issues/54
466

  import pandas as pd
```

Part 1 Hamiltonian Generation

1.1 : Interpotential Curve Point Selection

In [ ]:
```
# Set the Seaborn theme and color palette
sns.set_theme(context='paper', style='whitegrid')
dark_palette = sns.color_palette("dark", n_colors=7)  # Using a darker palet

# Initialize DataFrame to store the potential data
data_32 = pd.DataFrame()
data_1000 = pd.DataFrame()
```

```
states = [1, 3, 5, 7, 9, 11, 13]
labels = ['${}^1\Sigma_g^+$', '${}^3\Sigma_u^+$', '${}^5\Sigma_g^+$', '${}^7
          '${}^9\Sigma_g^+$', '${}^{11}\Sigma_u^+$', '${}^{13}\Sigma_g^+$']

# Fetch potential data for each state and accumulate it in the DataFrames
for i, state in enumerate(states):
    pot, lims = get_pot_cr2(state)
    rs_1000 = np.linspace(lims[0], lims[1], 1000)
    rs_32 = np.linspace(start=lims[0], stop=lims[1], num=256)
    potentials_32 = pot(rs_32) * hartree
    potentials_1000 = pot(rs_1000) * hartree
    temp_df_32 = pd.DataFrame({'r (a.u.)': rs_32, 'V (cm^-1)': potentials_32
    temp_df_1000 = pd.DataFrame({'r (a.u.)': rs_1000, 'V (cm^-1)': potential
    data_32 = pd.concat([data_32, temp_df_32], ignore_index=True)
    data_1000 = pd.concat([data_1000, temp_df_1000], ignore_index=True)


plt.figure(figsize=(12, 8))
sns.lineplot(
    data=data_1000,
    x='r (a.u.)',
    y='V (cm^-1)',
    hue='State',
    palette=dark_palette,
    legend=None
)


sns.scatterplot(
    data=data_32,
    x='r (a.u.)',
    y='V (cm^-1)',
    hue='State',
    palette=dark_palette,
    s=72,  # Adjust the size of the scatter plot markers
    edgecolor="w",
    legend='full'
)

# Configure plot aesthetics
plt.ylim((-17500, 1000))
plt.xlim((2, 8))
plt.xlabel(r'$r$ (a.u.)')
plt.ylabel(r'$V$ (Potential Energy) (cm$^{-1}$)')
plt.title('Potential Energy Curves for Different Electronic States')
plt.legend(title='Electronic State', loc='lower right')

# Display the plot
plt.show()
```



1.2: Generating the Hamiltonians for all Spin States

In [ ]:
```
mol_params =  cr2_params
list_of_spins = [1, 3, 5, 7, 9, 11, 13]
list_of_params32 = [[2.8, 4], [3.2, 4.5], [4, 6.5], [4.3, 6.8], [4.5, 7], [4

hamiltonians = create_hamiltonians(list_of_spins, list_of_params32, mol_para

for i, spin in enumerate(list_of_spins):
    params = list_of_params32[i]
    variable_name = f"h_dvr_{spin}"
    decomposed_hamiltonian = qml.pauli_decompose(hamiltonians[(spin, tuple(p
    globals()[variable_name] = decomposed_hamiltonian
```

1.3 Conversion of the hamiltonians into a unitary matrix

In [ ]:
```
wire_order = [0, 1, 2, 3, 4]
n = 5
for spin in list_of_spins:
    variable_name = f"ufss{spin}"
    hamiltonian = globals()[f"h_dvr_{spin}"]
    unitary = convert_to_unitary(hamiltonian, n, wire_order)
    globals()[variable_name] = unitary
```

1.4 Find the true phases of the unitary matrices

In [ ]:
```
for spin in list_of_spins:
    unitary_matrix = globals()[f"ufss{spin}"]
    phase = find_true_phases(unitary_matrix)
    phase = phase/np.linalg.norm(phase)
    phase = np.abs(phase/(2*np.pi))
    globals()[f"phase{spin}"] = phase
```

In [ ]:
```
for spin in list_of_spins:
    phase = globals()[f"phase{spin}"]
    print(phase.shape)
    print(f"True phases for spin state {spin}: {phase}")
```

```
(32,)
True phases for spin state 1: [0.04096298 0.04450851 0.03373529 0.03641698
0.04883094 0.04707422
 0.04618917 0.04335729 0.04236002 0.04286075 0.03153212 0.02933257
 0.02824815 0.02153788 0.02175125 0.0180878  0.01859339 0.0256471
 0.0104862  0.00672655 0.00572107 0.00376777 0.00121464 0.00027203
 0.02241888 0.02029003 0.01805162 0.01533049 0.01363865 0.00616677
 0.00840116 0.00825458]
(32,)
True phases for spin state 3: [0.02901343 0.02373451 0.03744477 0.04043268
0.0427488  0.04435596
 0.04636712 0.04968681 0.04959865 0.04161587 0.04204605 0.03526964
 0.03264829 0.01655096 0.01354258 0.01365763 0.01005915 0.02640197
 0.00565433 0.00484661 0.00216334 0.02348992 0.02325409 0.02170404
 0.02199635 0.01558653 0.01279555 0.00834406 0.0092766  0.00902679
 0.00247604 0.00232195]
(32,)
True phases for spin state 5: [0.01227631 0.00621999 0.00261552 0.00198135
0.00339383 0.00250196
 0.00124663 0.00148712 0.02469586 0.02625868 0.02744886 0.01272442
 0.01362003 0.01481766 0.0317335  0.02972174 0.02082199 0.02099297
 0.02160365 0.02358665 0.02330392 0.03152144 0.03432793 0.03763654
 0.03866702 0.03931646 0.0414154  0.0411701  0.04617502 0.04629184
 0.04383206 0.04384084]
(32,)
True phases for spin state 7: [0.03247615 0.04127042 0.04582127 0.04682656
0.04935615 0.04803262
 0.04830488 0.04015521 0.03836336 0.03455383 0.02986265 0.02907656
 0.02588974 0.02264514 0.01966771 0.02723006 0.02580416 0.02644467
 0.01410045 0.01252247 0.00749492 0.00653779 0.00323278 0.00115246
 0.01668291 0.01541079 0.01466869 0.00436287 0.00975986 0.00909482
 0.00761252 0.0077013 ]
(32,)
True phases for spin state 9: [0.03644479 0.03410099 0.03235278 0.02972791
0.02487595 0.02248715
 0.02213779 0.01563024 0.04440149 0.04709256 0.04630769 0.04367538
 0.04300372 0.04214087 0.03814891 0.03733337 0.03145188 0.03176127
 0.03271724 0.01129369 0.0081831  0.01781276 0.01595272 0.01556459
 0.00298519 0.00221959 0.00064878 0.00018968 0.00709408 0.00770577
 0.00807828 0.00825916]
(32,)
True phases for spin state 11: [0.02388759 0.02166877 0.02048656 0.01556992
0.01169694 0.00879841
 0.00615253 0.00500117 0.00131725 0.00297208 0.00586808 0.00400173
 0.00446676 0.01710126 0.02665297 0.02037801 0.02125365 0.02352124
 0.02249257 0.03355426 0.03469166 0.03594763 0.04107166 0.04480059
 0.04601715 0.04757934 0.04908886 0.03792824 0.0332597  0.0341589
 0.03558577 0.035445  ]
(32,)
True phases for spin state 13: [0.02667305 0.02834227 0.03292579 0.03407857
0.03630076 0.03544317
 0.04465175 0.04655567 0.04831635 0.01516329 0.01289853 0.01196757
 0.00946927 0.04294896 0.03566375 0.03503588 0.03888334 0.03964266
 0.03955065 0.02637651 0.02455309 0.02127476 0.02116977 0.01578449
 0.01294445 0.00698745 0.01036515 0.00980925 0.00114507 0.00114902
 0.00204674 0.00204323]
```

Part 2 Quantum Circuit Generation for Phase Estimation in the Fourier Basis

2.1 Obtaining Prior Distribution and Setting Up for Bayesian Estimation

In [ ]:
```
unitaries = [ufss1, ufss3, ufss5, ufss7, ufss9, ufss11, ufss13]
J_max = 2000
num_experiments = 100
c = np.random.random_integers(low=7000, high=8000, size=None)
J_values = [c for i in range(num_experiments)]
n_shots_per_experiment = 1/(1-(1/(2**5)))
ks = []
betas = []
```

```
/var/folders/k9/jxn4k2_s1v50xzyqggp1kft40000gn/T/ipykernel_85851/2573652648.
py:4: DeprecationWarning: This function is deprecated. Please call randint(7
000, 8000 + 1) instead
  c = np.random.random_integers(low=7000, high=8000, size=None)
```

2.3 Obtaining the Bayesian Estimation

In [ ]:
```
results = []
for i in range(len(unitaries)):
    U = unitaries[i]
    unitary_results = []
    for l in range(len(U)):
        # Generate a new seed for each l, ensuring each unitary eigenvalue h
        eigenvalue_results = []
        phase_estimates = []
        true_eigvals, true_eigvecs = jnp.linalg.eig(U)
        phi_vec = true_eigvecs[:, l]
        n_qubits_U = int(np.log2(U.shape[0]))
        for j in range(num_experiments):
            random_keys = random.split(random.PRNGKey(0), num_experiments)
            prior = initial_fourier_state(J_values[j])
            k1= np.random.random_integers(low=20, high=8000, size=None)
            beta, k, new_state = phase_estimation_iteration(prior, k1, U, ph
            fourier_basis_probs = new_state/np.linalg.norm(new_state)
            # print(fourier_basis_probs)
            phase = np.log(np.abs(np.arccos(fourier_basis_probs) - k - beta)
            nphase = np.mean(phase/ (2*np.pi))
            # print(f"Unitary {i+1}, Eigenvalue {l}: Experiment {j + 1} of {
            average_phase = np.mean(nphase)
            phase_estimates.append(average_phase)
            eigenvalue_results.append((k, beta, new_state, fourier_basis_prc
        unitary_results.append(eigenvalue_results)
    results.append(unitary_results)
```

```
/var/folders/k9/jxn4k2_s1v50xzyqggp1kft40000gn/T/ipykernel_85851/1510552935.
py:15: DeprecationWarning: This function is deprecated. Please call randint
(20, 8000 + 1) instead
  k1= np.random.random_integers(low=20, high=8000, size=None)
/var/folders/k9/jxn4k2_s1v50xzyqggp1kft40000gn/T/ipykernel_85851/1510552935.
py:15: DeprecationWarning: This function is deprecated. Please call randint
(20, 8000 + 1) instead
  k1= np.random.random_integers(low=20, high=8000, size=None)
```

Part 2.4 Obtaining Synthetic Unitary Phase Matrices

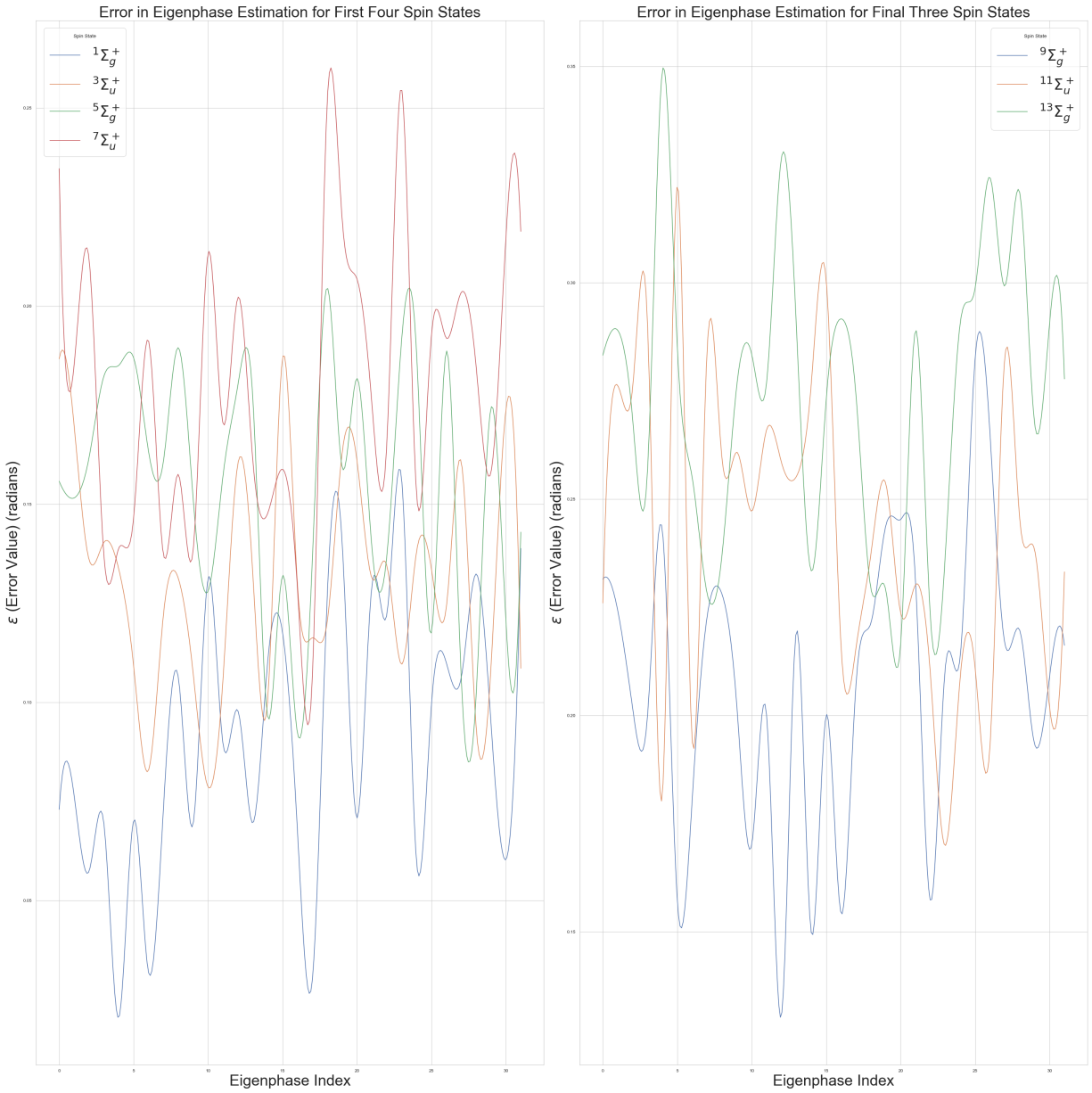
In [ ]:
```
num_unitaries = 7
num_eigenvalues = 32
average_phases = np.zeros((num_unitaries, num_eigenvalues))
for i in range(num_unitaries):
    for l in range(num_eigenvalues):
        all_phases = [experiment_data[-1] for experiment_data in results[i][
        average_phases[i, l] = np.mean(all_phases)
print(average_phases)
```

```
[[0.11395961 0.12111673 0.09104102 0.10592369 0.06922828 0.11720211
  0.07807143 0.11638102 0.14868714 0.11226954 0.16276877 0.11888868
  0.12617953 0.09108213 0.13188769 0.1357094  0.08626869 0.0554593
  0.137017   0.14726561 0.0765395  0.13311245 0.12331188 0.15654524
  0.08213437 0.11966629 0.12801765 0.12155099 0.14604159 0.10084592
  0.06871746 0.1470913 ]
 [0.18573934 0.16375986 0.14356634 0.15079993 0.14638789 0.12351263
  0.0992226  0.14173636 0.15090252 0.11885538 0.09070852 0.11103068
  0.16285193 0.11917759 0.08407069 0.17046222 0.10978495 0.11269349
  0.09596249 0.1377196  0.13314468 0.12483709 0.1278009  0.10131795
  0.13002306 0.11919911 0.10682631 0.13901824 0.07309473 0.08805895
  0.14624996 0.08091483]
 [0.10806192 0.09779248 0.10425775 0.12427528 0.128428   0.12945777
  0.10461319 0.10227016 0.15420115 0.11755738 0.09549206 0.10984612
  0.13364519 0.13193598 0.06819494 0.10165607 0.0530957  0.09676424
  0.16625066 0.12299077 0.14505373 0.11065858 0.11115564 0.16919513
  0.16693483 0.09699918 0.16998944 0.08743015 0.08789979 0.16059481
  0.10449243 0.12679575]
 [0.17716509 0.13619228 0.16797274 0.09405906 0.09825454 0.10463986
  0.14958811 0.08751243 0.10587692 0.08299973 0.15311566 0.10951629
  0.13801961 0.0939096  0.07733445 0.09607178 0.06458823 0.04262562
  0.17555495 0.14465526 0.1243597  0.09188099 0.07839951 0.16588652
  0.07927408 0.1180156  0.11658533 0.11781942 0.1045507  0.07716891
  0.13486521 0.13656639]
 [0.14793463 0.13836208 0.11404443 0.10957082 0.14761616 0.05982174
  0.08269269 0.11839959 0.15246695 0.12924878 0.09642825 0.12358081
  0.05365925 0.14161673 0.06788171 0.11734984 0.06567408 0.11884755
  0.13404033 0.13510743 0.13346457 0.12877452 0.0530991  0.10668735
  0.09660047 0.16468582 0.14577103 0.096764   0.10673943 0.08101795
  0.09704457 0.10447072]
 [0.09987066 0.14792761 0.14617535 0.15611042 0.04339428 0.18131077
  0.05039941 0.13925551 0.11078427 0.11381692 0.10308405 0.12007491
  0.11288551 0.12232678 0.15336508 0.17098401 0.08713941 0.08911204
  0.10972044 0.13716879 0.10840355 0.11615657 0.10048097 0.06469432
  0.10481496 0.10861614 0.09201349 0.17088619 0.12856022 0.1223587
  0.08747496 0.1186164 ]
 [0.1299143  0.13737421 0.12151149 0.11061744 0.20528017 0.14068857
  0.11967447 0.09478872 0.10526848 0.1115399  0.11636737 0.10901012
  0.15848495 0.15273784 0.0890598  0.12799093 0.15053999 0.13369177
  0.08890244 0.07540163 0.06099636 0.13024588 0.0637094  0.07304776
  0.12360635 0.12570602 0.15468851 0.12906601 0.14191705 0.08853123
  0.11261807 0.09982844]]
```

2.4 Graphing the Estimated Phase vs the True Phases

In [ ]:
```
true_phases = [phase1, phase3, phase5, phase7, phase9, phase11, phase13]

error_values = []
for syn_phase, true_phase in zip(average_phases, true_phases):
    error = np.abs(syn_phase - true_phase)
    error_values.append(error)

phase_indices = np.arange(len(error_values[0]))

labels = ['${}^1\Sigma_g^+$', '${}^3\Sigma_u^+$', '${}^5\Sigma_g^+$', '${}^7
```

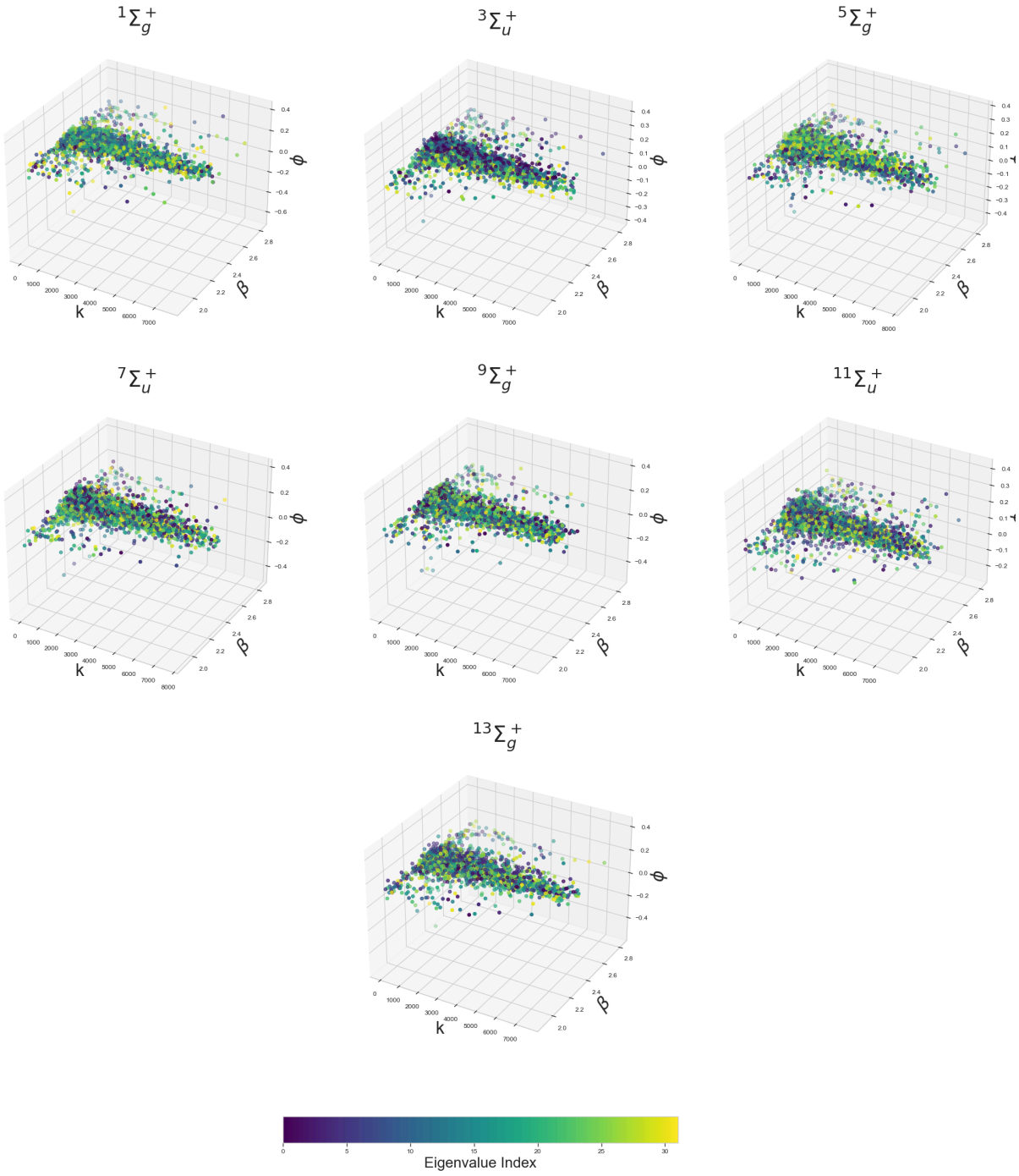
```
          '${}^9\Sigma_g^+$', '${}^{11}\Sigma_u^+$', '${}^{13}\Sigma_g^+$']

offset_increment = 0.03
n = 30
fig, axs = plt.subplots(1, 2, figsize=(n, n))
for index, errors in enumerate(error_values[:4]):
    spline = make_interp_spline(phase_indices, errors, k=3)
    smooth_indices = np.linspace(phase_indices.min(), phase_indices.max(), 3
    smooth_errors = spline(smooth_indices)
    offset_errors = smooth_errors + offset_increment * index
    sns.lineplot(x=smooth_indices, y=offset_errors, label=f'{labels[index]}'

axs[0].set_title('Error in Eigenphase Estimation for First Four Spin States'
axs[0].set_xlabel('Eigenphase Index', fontsize=n)
axs[0].set_ylabel(r'$\epsilon$ (Error Value) (radians)', fontsize=n)
axs[0].legend(title='Spin State', fontsize=n)

for index, errors in enumerate(error_values[4:7]):
    spline = make_interp_spline(phase_indices, errors, k=3)
    smooth_indices = np.linspace(phase_indices.min(), phase_indices.max(), 3
    smooth_errors = spline(smooth_indices)
    offset_errors = smooth_errors + offset_increment * (index + 4)
    sns.lineplot(x=smooth_indices, y=offset_errors, label=f'{labels[index +

axs[1].set_title('Error in Eigenphase Estimation for Final Three Spin States
axs[1].set_xlabel('Eigenphase Index', fontsize=n)
axs[1].set_ylabel(r'$\epsilon$ (Error Value) (radians)', fontsize=n)
axs[1].legend(title='Spin State', fontsize=n)

plt.tight_layout()
plt.show()
```



In [ ]:

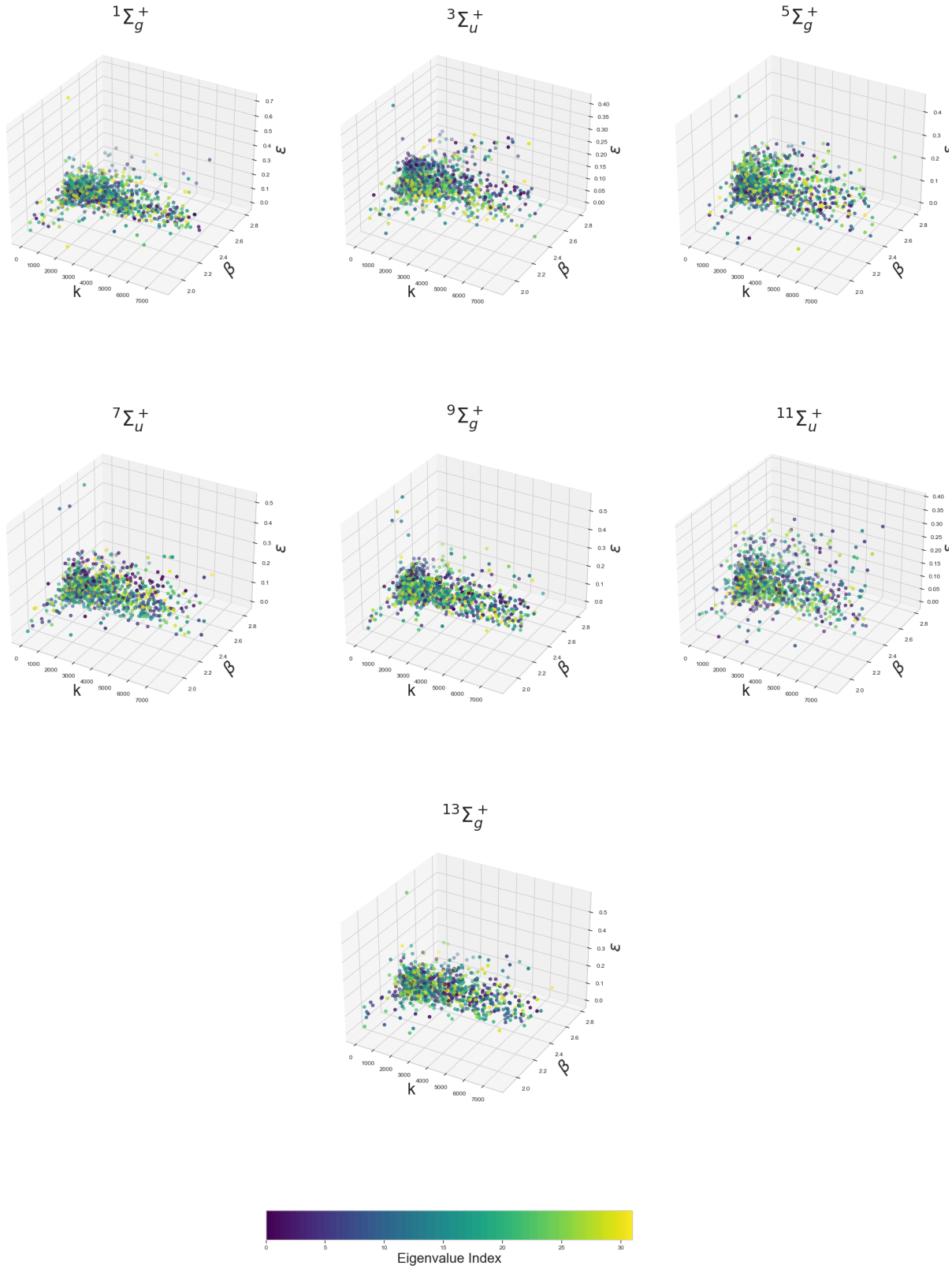
In [ ]:
```
fig = plt.figure(figsize=(25, 25))
labels = ['${}^1\Sigma_g^+$', '${}^3\Sigma_u^+$', '${}^5\Sigma_g^+$', '${}^7
          '${}^9\Sigma_g^+$', '${}^{11}\Sigma_u^+$', '${}^{13}\Sigma_g^+$']

for i in range(7):
    if i < 6:
        ax = fig.add_subplot(3, 3, i + 1, projection='3d')
    else:
        ax = fig.add_subplot(3, 3, 8, projection='3d')

    all_ks = []
    all_betas = []
    all_phases = []
    all_eigenvalue_indices = []

    unitary_results = results[i]
    for j, eigenvalue_results in enumerate(unitary_results):
```

```
        ks = []
        betas = []
        average_phases = []

        for experiment_result in eigenvalue_results:
            k, beta, _, _, average_phase = experiment_result
            ks.append(k)
            betas.append(beta)
            average_phases.append(average_phase)

        all_ks.extend(ks)
        all_betas.extend(betas)
        all_phases.extend(average_phases)
        all_eigenvalue_indices.extend([j] * len(ks))

    all_ks = np.array(all_ks)
    all_betas = np.array(all_betas)
    all_phases = np.array(all_phases)
    all_eigenvalue_indices = np.array(all_eigenvalue_indices)

    scatter = ax.scatter(all_ks, all_betas, all_phases, c=all_eigenvalue_inc
    ax.set_xlabel('k',  fontsize=24)
    ax.set_ylabel(r'$\beta$', fontsize=24)
    ax.set_zlabel(r'$\phi$', fontsize=24)
    ax.set_title(labels[i], fontsize=30)

cbar_ax = fig.add_axes([0.35, 0.04, 0.3, 0.02])
cbar = fig.colorbar(scatter, cax=cbar_ax, orientation='horizontal')
cbar.set_label('Eigenvalue Index', fontsize=20)


plt.show()
```



In [ ]:
```
# Assuming 'results' and 'true_phases' are defined somewhere before this coc

fig = plt.figure(figsize=(25, 30))
labels = ['${}^1\Sigma_g^+$', '${}^3\Sigma_u^+$', '${}^5\Sigma_g^+$', '${}^7
          '${}^9\Sigma_g^+$', '${}^{11}\Sigma_u^+$', '${}^{13}\Sigma_g^+$']

# Assuming 'results' and 'true_phases' are defined somewhere before this coc

for i in range(7):
    if i < 6:
        ax = fig.add_subplot(3, 3, i + 1, projection='3d')
    else:
        ax = fig.add_subplot(3, 3, 8, projection='3d')
```

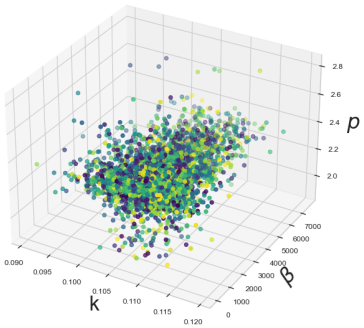
```
    all_ks = []
    all_betas = []
    all_errors = []
    all_eigenvalue_indices = []

    unitary_results = results[i]
    true_phase = true_phases[i]

    for j, eigenvalue_results in enumerate(unitary_results):
        ks = []
        betas = []
        errors = []
        for experiment_result, true_phase_value in zip(eigenvalue_results, t
            k, beta, _, _, average_phase = experiment_result
            ks.append(k)
            betas.append(beta)
            errors.append(np.abs(average_phase - true_phase_value))

        all_ks.extend(ks)
        all_betas.extend(betas)
        all_errors.extend(errors)
        all_eigenvalue_indices.extend([j] * len(ks))

    all_ks = np.array(all_ks)
    all_betas = np.array(all_betas)
    all_errors = np.array(all_errors)
    all_eigenvalue_indices = np.array(all_eigenvalue_indices)

    scatter = ax.scatter(all_ks, all_betas, all_errors, c=all_eigenvalue_inc
    ax.set_xlabel('k', fontsize=24)
    ax.set_ylabel(r'$\beta$', fontsize=24)
    ax.set_zlabel(r'$\epsilon$', fontsize=24)
    ax.set_title(labels[i], fontsize=30)

# Add color bar
cbar_ax = fig.add_axes([0.35, 0.04, 0.3, 0.02])
cbar = fig.colorbar(scatter, cax=cbar_ax, orientation='horizontal')
cbar.set_label('Eigenvalue Index', fontsize=20)

plt.show()
```



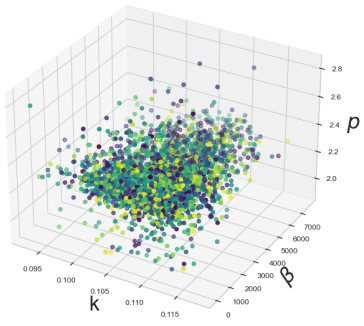
In [ ]:
```
fig = plt.figure(figsize=(25, 25))
labels = ['${}^1\Sigma_g^+$', '${}^3\Sigma_u^+$', '${}^5\Sigma_g^+$', '${}^7
          '${}^9\Sigma_g^+$', '${}^{11}\Sigma_u^+$', '${}^{13}\Sigma_g^+$']

for i in range(7):
    if i < 6:
        ax = fig.add_subplot(3, 3, i + 1, projection='3d')
    else:
```

```
        ax = fig.add_subplot(3, 3, 8, projection='3d')

    all_ks = []
    all_betas = []
    all_phases = []
    all_eigenvalue_indices = []

    unitary_results = results[i]
    for j, eigenvalue_results in enumerate(unitary_results):
        ks = []
        betas = []
        average_phases = []

        for experiment_result in eigenvalue_results:
            k, beta, _, fourier_basis_probs, _, = experiment_result
            ks.append(k)
            betas.append(beta)
            average_phases.append(np.mean(fourier_basis_probs))

        all_ks.extend(ks)
        all_betas.extend(betas)
        all_phases.extend(average_phases)
        all_eigenvalue_indices.extend([j] * len(ks))

    all_ks = np.array(all_ks)
    all_betas = np.array(all_betas)
    all_phases = np.array(all_phases)
    all_eigenvalue_indices = np.array(all_eigenvalue_indices)

    scatter = ax.scatter(all_phases, all_ks, all_betas, c=all_eigenvalue_inc
    ax.set_xlabel('k',  fontsize=24)
    ax.set_ylabel(r'$\beta$', fontsize=24)
    ax.set_zlabel(r'$p$', fontsize=24)
    ax.set_title(labels[i], fontsize=30)

cbar_ax = fig.add_axes([0.35, 0.04, 0.3, 0.02])
cbar = fig.colorbar(scatter, cax=cbar_ax, orientation='horizontal')
cbar.set_label('Eigenvalue Index', fontsize=20)


plt.show()
```

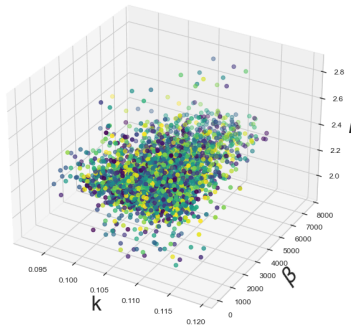
$^{1}\Sigma_{g}^{+}$



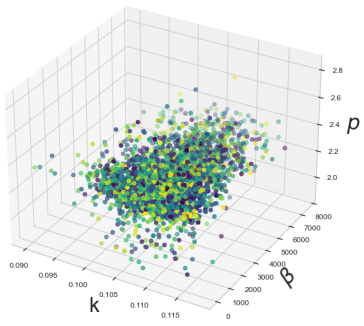
$^{3}\Sigma_{u}^{+}$



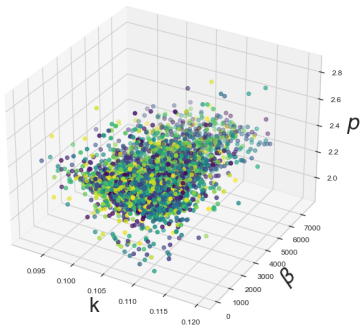
$^{5}\Sigma_{g}^{+}$



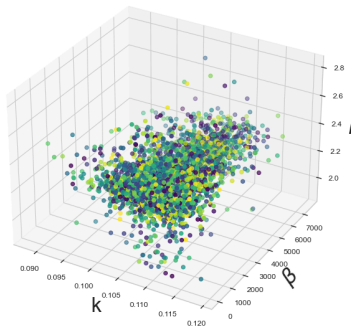
$^{7}\Sigma_{u}^{+}$



$^{9}\Sigma_{g}^{+}$



$^{11}\Sigma_{u}^{+}$



$^{13}\Sigma_{g}^{+}$