QUANTUM SYSTEMS ANALYSIS

OUR TEAM

Divyansh Kumar
(23B1833)

Pulin Tanmay Beck
(23B1822)

Soumik Sahoo
(23B1825)

Divyansh Patel
(23B1803)

PROBLEM STATEMENT

Understanding and predicting quantum mechanical properties, such as energy levels and wavefunctions, is crucial in physics and engineering. Analytical solutions to the Schrödinger equation exist for only a few idealized systems, while numerical methods solves this problem but it can be computationally expensive and challenging to scale for complex systems.

CHALLENGES AND SOLUTIONS

Computing energy levels for quantum systems using numerical methods requires solving matrix equations that become computationally expensive as complexity increases. Machine learning models can approximate these relationships once trained, enabling rapid predictions without recalculating.

CHALLENGES AND SOLUTIONS

For systems where analytical solutions do not exist (e.g., with nonlinear potentials or multi-particle systems), machine learning models provide a way to generalize and predict results. By training on data generated from known physical systems, machine learning models can learn the underlying physics relationships (e.g., the dependence of energy levels on mass and spring constant for 1D harmonic oscillator).

CHALLENGES AND SOLUTIONS

This project aims to explore the use of machine learning techniques, such as Physics-Informed Neural Networks (PINNs), Random Forest (RF), and XGBoost (XGB), to efficiently predict quantum properties, including energy levels and wavefunctions, for one-dimensional systems (specifically 1D harmonic oscillators). The ultimate goal is to provide a faster, scalable, and more versatile alternative to traditional numerical solvers while maintaining high accuracy and physics consistency.

CHALLENGES AND SOLUTIONS

So we are comparing the performances of GBR, RF, and Physics Induced Neural Network, aiming to:-

- Identify the most accurate method for predicting energy levels.
- Evaluate trade-offs in computational efficiency and prediction accuracy.

UNDERSTANDING THE METHODS

1. Gradient Boosting Regressor (GBR):-

- Builds models sequentially, with each model correcting the errors of the previous one.
- Uses decision trees as base learners and optimizes predictions by minimizing a loss function (like MSE).
- Handles nonlinear relationships well.
- Effective for small datasets and accurate for regression tasks.
- Predicts multiple energy levels of the quantum system using MultiOutputRegressor.

UNDERSTANDING THE METHODS

2. Random Forest Regressor (RF):-

- An ensemble of decision trees trained on random subsets of the data and features.
- Aggregates results from all trees to make predictions, reducing overfitting.
- Robust and interpretable.
- Provides a baseline for comparing ensemble methods.
- Also used for multi-output regression to predict energy levels.

UNDERSTANDING THE METHODS

3. Physics-Informed Neural Networks (PINN):-

- Combines neural networks with physical laws (Here Schrödinger equation).
- Trained to minimize a loss function that includes both data fitting and compliance with physics equations.
- Captures complex, nonlinear relationships.
- Incorporates domain knowledge for better generalization.
- Predicts energy levels with high flexibility and integrates quantum mechanical constraints.

THEORY

$$i\hbar\frac{\partial}{\partial t}\psi(x,t) = \hat{H}\psi(x,t)$$

Schordinger Equation

$$\hat{A}\phi = \lambda\phi$$

Eigen value Representation

THEORY

$$E_n = \hbar\omega(n + \frac{1}{2})$$

Energy of 1D Harmonic Oscillator

$$\omega = \sqrt{\frac{k}{m}}$$

Relation with mass and spring constant

THEORY

First, we derive the difference equation for 1-D harmonic oscillators.
This is done by sampling the Schrodinger equation at N points and converting the Hamiltonian into an $N \times N$ matrix.
This transformation converts the differential equation into a matrix equation,
where the solutions correspond to the eigenvalues and eigenvectors of the Hamiltonian matrix.

Kinetic Energy Matrix (T):

- Derived from the second derivative term in the Schrödinger equation:

$$T = -\frac{\hbar^2}{2m}\frac{d^2}{dx^2}.$$

- Numerically, this is implemented using finite differences:

$$T_{i,j} = \begin{cases} -\frac{\hbar^2}{2m\Delta x^2}, & \text{if } i = j+1 \text{ or } i = j-1, \\ \frac{\hbar^2}{m\Delta x^2}, & \text{if } i = j, \\ 0, & \text{otherwise.} \end{cases}$$

- This matrix is tridiagonal, reflecting the local nature of the second derivative operator.

Potential Energy Matrix (V):

- Derived from the potential energy term $V(x) = \frac{1}{2}kx^2$.
- This is a diagonal matrix:

$$V_{i,j} = \begin{cases} \frac{1}{2}kx_i^2, & \text{if } i = j, \\ 0, & \text{otherwise.} \end{cases}$$

$$H = \begin{bmatrix} V_1 + T_{11} & T_{12} & 0 & \cdots & 0 \\ T_{21} & V_2 + T_{22} & T_{23} & \cdots & 0 \\ 0 & T_{32} & V_3 + T_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & T_{N-1,N} \\ 0 & 0 & 0 & T_{N,N-1} & V_N + T_{NN} \end{bmatrix}.$$

THEORY

$$H = \begin{bmatrix} V_1 + T_{11} & T_{12} & 0 & \cdots & 0 \\ T_{21} & V_2 + T_{22} & T_{23} & \cdots & 0 \\ 0 & T_{32} & V_3 + T_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & T_{N-1,N} \\ 0 & 0 & 0 & T_{N,N-1} & V_N + T_{NN} \end{bmatrix}.$$

One of our teammate had read this book learning about the numerical soultions to differential equations. The idea to make a model to calculate these numerical solutions more efficienntly came from there

Reference for the theory was taken from the following book
Quantum Transport: Atom to Transistor by Supriyo Datta

Eigenproblem of the Hamiltonian

To find the energy levels, we solve the eigenvalue problem:

$$H\psi = E\psi,$$

where:

- E are the eigenvalues (quantized energy levels of the oscillator).
- ψ are the eigenvectors (wavefunctions of the energy states).

DEFINING 1-D SCHRODINGER EQUATION

```python
# Constants
hbar = 1.0  # Reduced Planck's constant
N = 100     # Number of grid points
L = 10.0    # Length of the 1D box
x = np.linspace(-L / 2, L / 2, N)  # Spatial grid
dx = x[1] - x[0]  # Grid spacing

def potential_harmonic(x, k):
    #Harmonic potential: V(x) = 0.5 * k * x^2
    return 0.5 * k * x**2

def solve_schrodinger_1d(m, k):
    #Solve the 1D Schrödinger equation for a harmonic oscillator.
    V = potential_harmonic(x, k)  # Potential
    kinetic = -0.5 * (hbar**2 / m) * (-2 * np.eye(N) + np.eye(N, k=1) + np.eye(N, k=-1)) / dx**2
    H = kinetic + np.diag(V)  # Hamiltonian

    # Solve eigenvalue problem
    eigenvalues, _ = eigh(H)
    return eigenvalues[:5]  # Return first 5 energy levels
```

This code converts the Hamiltonian for a 1-D Harmonic Oscillator into a matrix, transforming the Schrodinger equation from a differential to a difference equation for numerical solutions. Using $H(\psi) = E(\psi)$, the eigenvalues of H represent the allowed energy levels. The first 5 energy levels for the given (m, k) combination were calculated.

GENERATING OUR OWN DATASET

```python
# Generate dataset
data = []
m_values = np.arange(0.1, 50, 0.2) # Range of masses
k_values = np.arange(0.1, 50, 0.2)  # Range of spring constants

for m in m_values:
    for k in k_values:
        energies = solve_schrodinger_1d(m, k)
        data.append([m, k] + list(energies))
```

As we decided to undertake this project independently from scratch,
finding a large dataset with mass, spring constant,
and labeled numerical solutions for the 1D harmonic oscillator was quite challenging.
Therefore, we decided to create our own dataset for training the ML models.

```python
# Convert to a Pandas DataFrame
columns = ['mass', 'spring_constant'] + [f'energy_{i+1}' for i in range(5)]
df = pd.DataFrame(data, columns=columns)
print(df.head())

# Save dataset
df.to_csv('quantum_data.csv', index=False)
```

	mass	spring_constant	energy_1	energy_2	energy_3	energy_4	energy_5
0	0.1	0.1	0.634374	2.253990	4.669856	7.996048	12.254691
1	0.1	0.3	0.907159	2.915330	5.479934	8.847789	13.117653
2	0.1	0.5	1.134268	3.511434	6.272234	9.708487	13.994845
3	0.1	0.7	1.330011	4.051910	7.035849	10.569426	14.882987
4	0.1	0.9	1.503287	4.545650	7.765934	11.422959	15.778276

CREATING TRAINING AND TESTING DATA

```
# Load data
data = pd.read_csv('quantum_data.csv')

# Define features and targets
X = data[['mass', 'spring_constant']]
y = data[[f'energy_{i+1}' for i in range(5)]]

# Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Divinding the dataset into 80% training and 20% test.

As explained above we are using Gradient Booster and Random Forest

```
# Initialize a multi-output regressor
multi_output_model = MultiOutputRegressor(GradientBoostingRegressor())

# Train the model
multi_output_model.fit(X_train, y_train)
```

MultiOutputRegressor

estimator: GradientBoostingRegressor

GradientBoostingRegressor

```
# Initialize the Random Forest model
rf_model = RandomForestRegressor(n_estimators=100, random_state=42)

# Train the model
rf_model.fit(X_train, y_train)
```

RandomForestRegressor

RandomForestRegressor(random_state=42)

CREATING TRAINING AND TESTING DATA

```python
# Predictions from XGBoost
y_pred_xgb = multi_output_model.predict(X_test)

# Predictions from Random Forest
y_pred_rf = rf_model.predict(X_test)
```

```python
# Evaluate XGBoost
mse_xgb = mean_squared_error(y_test, y_pred_xgb)
r2_xgb = r2_score(y_test, y_pred_xgb)

# Evaluate Random Forest
mse_rf = mean_squared_error(y_test, y_pred_rf)
r2_rf = r2_score(y_test, y_pred_rf)

# Print results
print("XGBoost Results:")
print(f"Mean Squared Error: {mse_xgb}")
print(f"R^2 Score: {r2_xgb}")

print("\nRandom Forest Results:")
print(f"Mean Squared Error: {mse_rf}")
print(f"R^2 Score: {r2_rf}")
```

```
XGBoost Results:
Mean Squared Error: 0.013016638013818593
R^2 Score: 0.9991812899868193

Random Forest Results:
Mean Squared Error: 0.00033504664801053643
R^2 Score: 0.9999782679376749
```

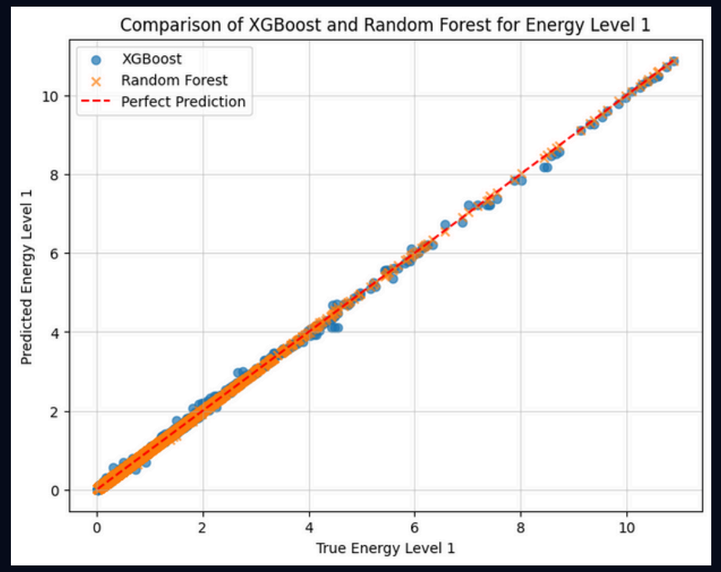
As we can see, the Random Forest model outperformed XGBoost significantly in both MSE and R^2 score.

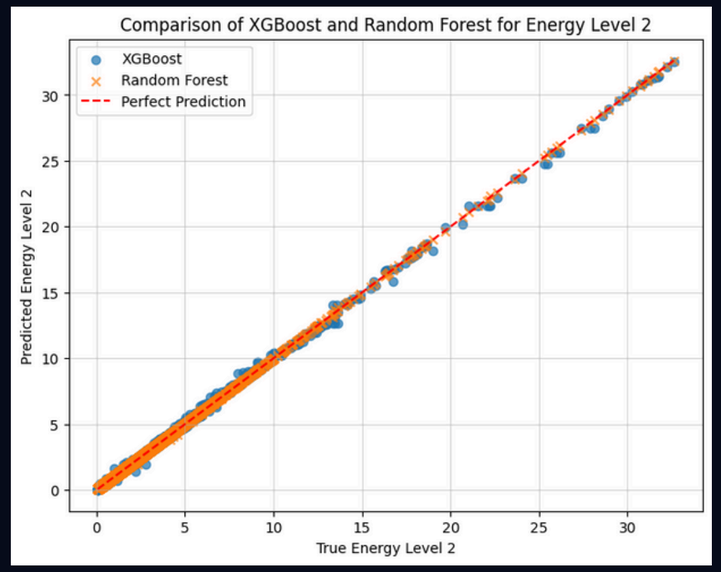
The MSE values of the Random Forest model were approximately $\frac{1}{100}$ of that of the XGBoost model.

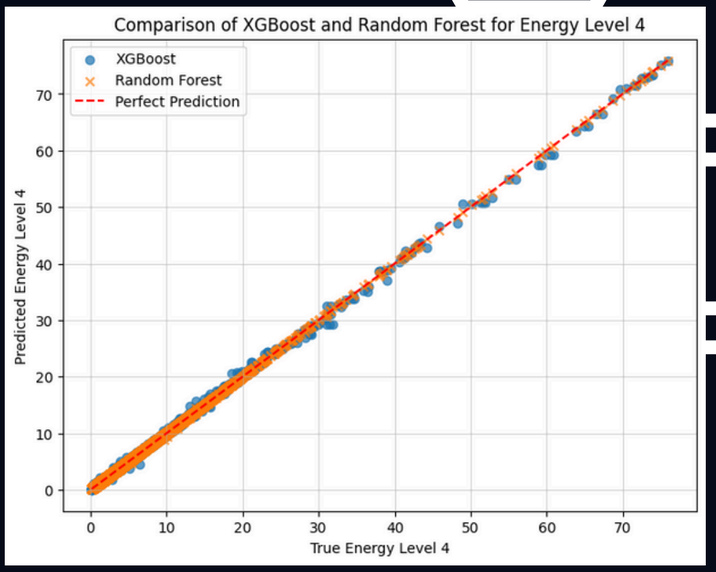
Meanwhile, the R^2 score, which measures the explained variance by the models,
is also greater for Random Forest, although not by much.

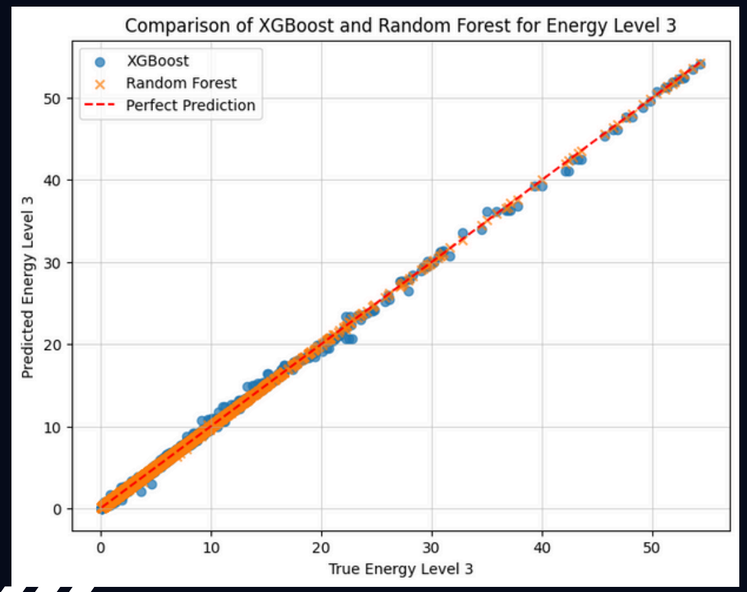
PLOTTING FIRST 5 ENERGY LEVELS

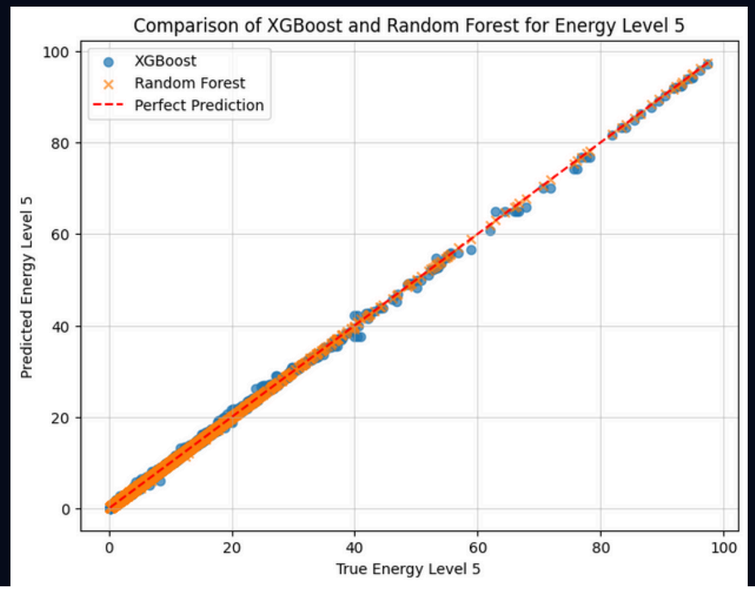
```python
for i in range(5):
    plt.figure(figsize=(8, 6))
    true_values = y_test[f'energy_{i+1}']
    plt.scatter(true_values, y_pred_xgb[:, i], label='XGBoost', alpha=0.7, marker='o')
    plt.scatter(true_values, y_pred_rf[:, i], label='Random Forest', alpha=0.7, marker='x')
    plt.plot(
        [min(true_values), max(true_values)],
        [min(true_values), max(true_values)],
        color='red', linestyle='--', label='Perfect Prediction'
    )
    plt.xlabel(f"True Energy Level {i+1}")
    plt.ylabel(f"Predicted Energy Level {i+1}")
    plt.title(f"Comparison of XGBoost and Random Forest for Energy Level {i+1}")
    plt.legend()
    plt.grid(alpha=0.5)
    plt.show()
```


Comparison of XGBoost and Random Forest for Energy Level 1
XGBoost
Random Forest
Perfect Prediction
Predicted Energy Level 1
True Energy Level 1


Comparison of XGBoost and Random Forest for Energy Level 2
XGBoost
Random Forest
Perfect Prediction
Predicted Energy Level 2
True Energy Level 2


Comparison of XGBoost and Random Forest for Energy Level 4
XGBoost
Random Forest
Perfect Prediction
Predicted Energy Level 4
True Energy Level 4


Comparison of XGBoost and Random Forest for Energy Level 3
XGBoost
Random Forest
Perfect Prediction
Predicted Energy Level 3
True Energy Level 3


Comparison of XGBoost and Random Forest for Energy Level 5
XGBoost
Random Forest
Perfect Prediction
Predicted Energy Level 5
True Energy Level 5

PLOTTING AND COMPARISION

One common theme observable in all the above plots is that, at lower energy values for all 5 levels,
XGBoost and Random Forest closely follow the perfect prediction line.
However, as the energy increases, XGBoost tends to deviate slightly from the perfect prediction line,
whereas Random Forest continues to closely follow it even at higher energies.
This can be attributed to the synthetic dataset having fewer training cases for very large k and very small m.

Since the numerical solution for the energy levels approximately changes linearly with $\omega = \sqrt{\frac{k}{m}}$, as the analytical solutions are given by

$$E = \left(n + \frac{1}{2}\right)\hbar\omega,$$

the lower number of training cases causes XGBoost to underperform for higher energy cases (large k, small m).
On the other hand, Random Forest, which trains by building trees independently (in parallel)
and uses simple averaging, trains faster and performs well even at higher energies, unlike XGBoost, which builds trees sequentially.

PLOTTING AND COMPARISION

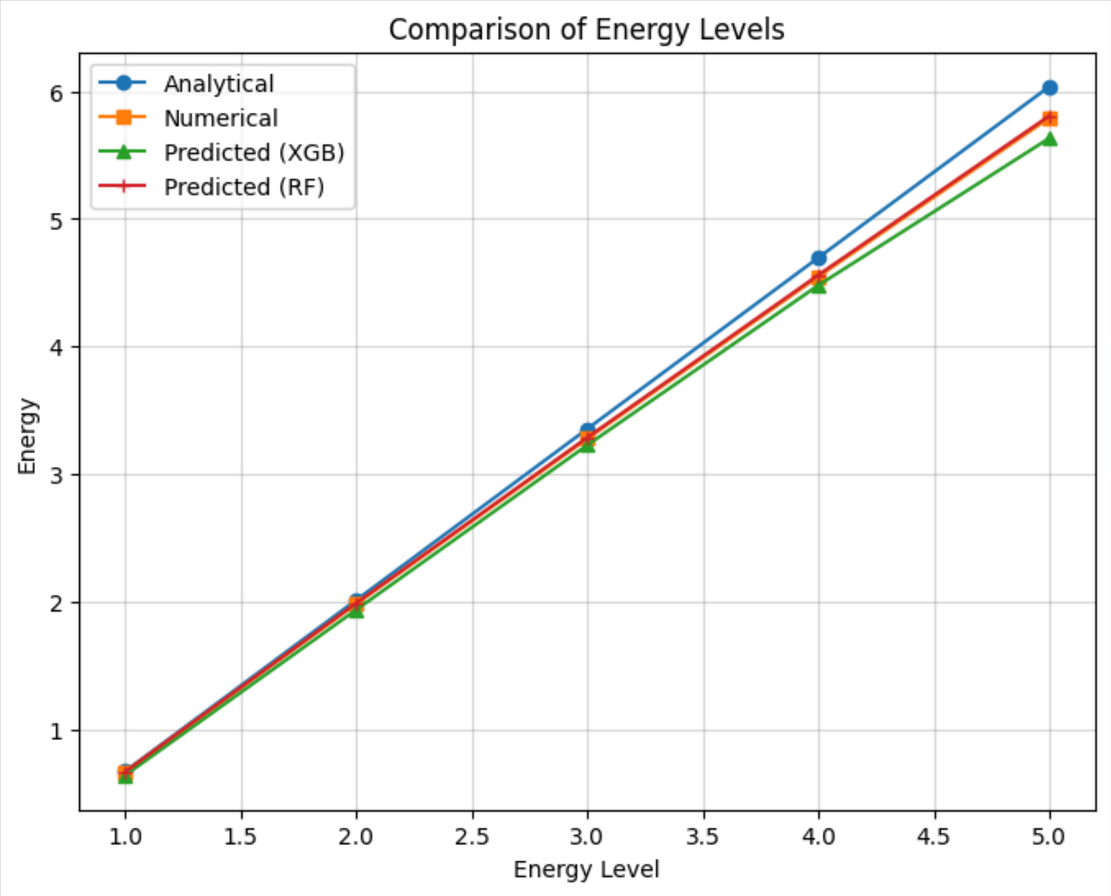
```python
def analytical_energy_levels(m, k, hbar=1):
    omega = np.sqrt(k / m)
    return [hbar * omega * (n + 0.5) for n in range(5)]
```

```python
# Define a sample system
m = 10.0  # Mass
k = 18.0  # Spring constant

feature_names = ['mass', 'spring_constant']
sample_input = pd.DataFrame([[m, k]], columns=feature_names)

# Get results
analytical = analytical_energy_levels(m, k)
numerical = solve_schrodinger_1d(m, k)
predicted_XGB = multi_output_model.predict(sample_input)[0]
predicted_RF = rf_model.predict(sample_input)[0]

# Print comparison
print("Energy Level Comparison:")
print(f"Analytical: {analytical}")
print(f"Numerical:  {numerical}")
print(f"Predicted:  {predicted_XGB}")
print(f"Predicted:  {predicted_RF}")
```

```python
# Plot comparison
levels = range(1, 6)
plt.figure(figsize=(8, 6))
plt.plot(levels, analytical, label='Analytical', marker='o')
plt.plot(levels, numerical, label='Numerical', marker='s')
plt.plot(levels, predicted_XGB, label='Predicted (XGB)', marker='^')
plt.plot(levels, predicted_RF, label='Predicted (RF)', marker='+')
plt.xlabel("Energy Level")
plt.ylabel("Energy")
plt.title("Comparison of Energy Levels")
plt.legend()
plt.grid(alpha=0.5)
plt.show()
```

```
Energy Level Comparison:
Analytical: [0.6708203932499369, 2.0124611797498106, 3.3541019662496847, 4.695742752749559, 6.037383539249432]
Numerical:  [0.66503077 1.98330308 3.27765928 4.54738988 5.79171074]
Predicted:  [0.63970839 1.93694922 3.22714183 4.47945204 5.63249457]
Predicted:  [0.66671791 1.98842992 3.28636142 4.55981327 5.8080135 ]
```

PLOTTING AND COMPARISION



Here also we can confirm that Random Forest follows closely the numrical solutions while XGBoost deviates a little from it.
The numerical solution itself deviates from analytical solutions which was expected as numerical solutions are only an approximation of the analytical ones

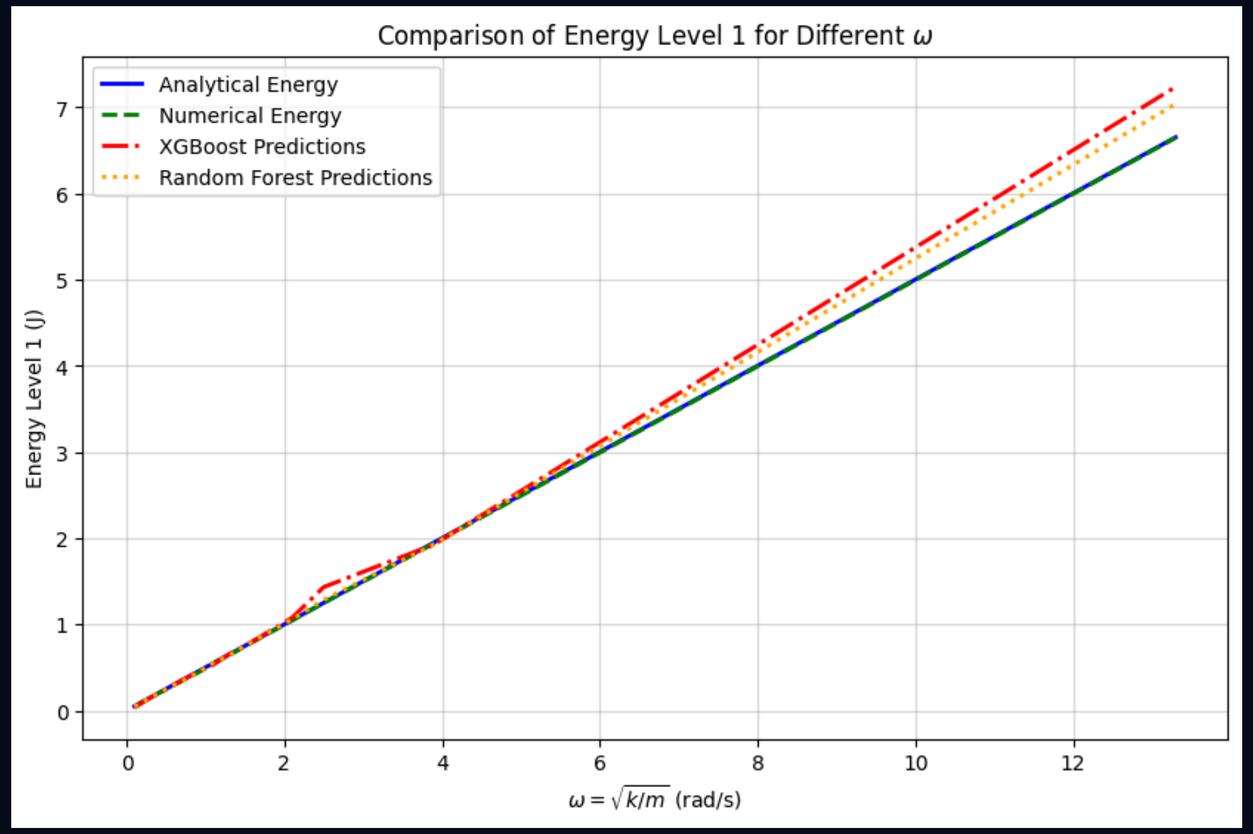
PLOTTING THE GROUND STATE ENERGY LEVEL PREDICTION FOR DIFFERENT OMEGA

```python
# Generate omega values
omega_values = np.arange(0.1, 50, 0.2)

# Generate valid m and k for all omega values
m_values = np.random.uniform(0.1, 50, len(omega_values))
k_values = m_values * omega_values**2
valid_indices = (k_values >= 0.1) & (k_values <= 50)

# Filter valid omega, m, and k
omega_values = omega_values[valid_indices]
m_values = m_values[valid_indices]
k_values = k_values[valid_indices]

# Prepare input features for predictions
feature_names = ['mass', 'spring_constant']
input_features = pd.DataFrame({'mass': m_values, 'spring_constant': k_values})

# Calculate analytical, numerical, and predicted energies
analytical_energy = [analytical_energy_levels(m, k)[0] for m, k in zip(m_values, k_values)]
numerical_energy = [solve_schrodinger_1d(m, k)[0] for m, k in zip(m_values, k_values)]
XGB_energy = multi_output_model.predict(input_features)[:, 0]  # Energy level 1 predictions
RF_energy = rf_model.predict(input_features)[:, 0]             # Energy level 1 predictions
```

```python
plt.figure(figsize=(10, 6))
plt.plot(omega_values, analytical_energy, label="Analytical Energy", color='blue', linewidth=2)
plt.plot(omega_values, numerical_energy, label="Numerical Energy", color='green', linestyle='--', linewidth=2)
plt.plot(omega_values, XGB_energy, label="XGBoost Predictions", color='red', linestyle='-.', linewidth=2)
plt.plot(omega_values, RF_energy, label="Random Forest Predictions", color='orange', linestyle=':', linewidth=2)

plt.xlabel(r"$\omega = \sqrt{k/m}$ (rad/s)")
plt.ylabel("Energy Level 1 (J)")
plt.title("Comparison of Energy Level 1 for Different $\omega$")
plt.legend()
plt.grid(alpha=0.5)
plt.show()
```

Creating a test dataset with one case of k and m for each ω to avoid redundancy (e.g., $k = 1, m = 1$ and $k = 2, m = 2$ give the same ω). This dataset will be used to analyze the performance of the models for varying ω in the ground state alone.

PLOT FOR GROUND STATE VS. OMEGA



Both models closely follow the numerical line at lower ω values, with Random Forest being more accurate.

As $\omega = \sqrt{\frac{k}{m}}$ increases, the lack of training data in the higher ω range causes both models to deviate.

However, Random Forest remains closer to the numerical line compared to XGBoost.

PHYSICS INDUCED NEURAL NETWORK

After exploring simpler models like Random Forest and XGBoost,
we now focus on Physics-Informed Neural Networks (PINNs).
PINNs excel in physics regression problems by embedding governing equations
into the loss function, ensuring solutions obey physical laws.
Traditional models like XGBoost and Random Forest are purely data-driven and lack this capability.
This often leads to physically invalid predictions, particularly in regions with
sparse training data,such as at larger ω values.

PHYSICS INDUCED NEURAL NETWORK

```python
# Define the PINN class
class EnergyPINN(Model):
    def __init__(self):
        super(EnergyPINN, self).__init__()
        self.hidden_layers = [
            layers.Dense(64, activation='tanh'),
            layers.Dense(64, activation='tanh'),
            layers.Dense(64, activation='tanh'),
            layers.Dense(1, activation=None)  # Output
        ]

    def call(self, inputs):
        x = inputs
        for layer in self.hidden_layers:
            x = layer(x)
        return x
```

Here we have defined a fully connected neural network with 3 hidden layers, each having 64 neurons and a 'tanh' activation with the final layer having 1 neuron giving the energy level.

PHYSICS INDUCED NEURAL NETWORK

As PINNs are computationally very expensive as compare to tradional ML models
hence we cannot use the existing quantum_data.csv file
which we created for the training of XGBoost and Random Forest.
Hence in this cell we have created a lighter dataset,
for the ground state energy level only, for PINN to train on.
Of course if we would have used quantum_data.csv file then
the model would have been even better
but it is computationally very expensive especially for my laptop specs :)

```python
# Generate synthetic training data
def generate_data(num_samples=1000):
    masses = np.random.uniform(0.1, 50, num_samples)
    spring_constants = np.random.uniform(0.1, 50, num_samples)
    omega = np.sqrt(spring_constants / masses)
    energies = 0.5 * omega  # Analytical formula for the first energy level
    inputs = np.vstack((masses, spring_constants)).T
    return inputs, energies

# Prepare data
X_train, y_train = generate_data(1000)
X_test, y_test = generate_data(200)

# Convert data to float32 before creating TensorFlow datasets
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
y_train = y_train.astype('float32')
y_test = y_test.astype('float32')

# Convert to TensorFlow datasets
train_dataset = tf.data.Dataset.from_tensor_slices((X_train, y_train)).batch(32)
test_dataset = tf.data.Dataset.from_tensor_slices((X_test, y_test)).batch(32)
```

PHYSICS INDUCED NEURAL NETWORK

```python
# Initialize the model and optimizer
pinn = EnergyPINN()
optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
loss_fn = tf.keras.losses.MeanSquaredError()
```

```
Epoch 0: Loss = 0.003642
Epoch 500: Loss = 0.000007
Epoch 1000: Loss = 0.000008
Epoch 1500: Loss = 0.000003
Epoch 2000: Loss = 0.000003
Epoch 2500: Loss = 0.000038
```

```python
# Training step
@tf.function
def train_step(inputs, targets):
    with tf.GradientTape() as tape:
        predictions = pinn(inputs)
        loss = loss_fn(targets, predictions)
    gradients = tape.gradient(loss, pinn.trainable_variables)
    optimizer.apply_gradients(zip(gradients, pinn.trainable_variables))
    return loss

# Training loop
epochs = 3000
for epoch in range(epochs):
    for inputs, targets in train_dataset:
        loss = train_step(tf.convert_to_tensor(inputs, dtype=tf.float32),
                          tf.convert_to_tensor(targets, dtype=tf.float32))
    if epoch % 500 == 0:
        print(f"Epoch {epoch}: Loss = {loss.numpy():.6f}")
```

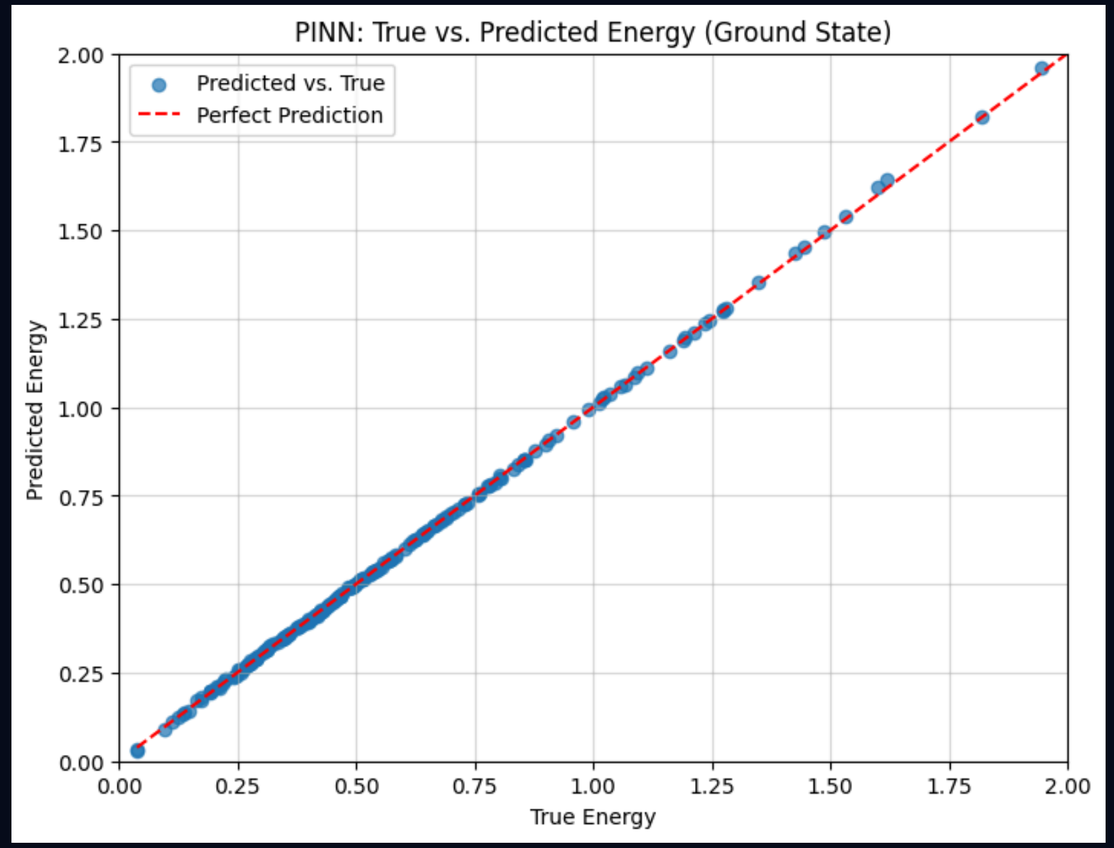
Here we have used gradient descent optimization for the training process of PINN.

TRUE VS. PREDICTED ENERGY

```python
# Evaluate on test data
predictions = []
true_energies = []
for inputs, targets in test_dataset:
    preds = pinn(tf.convert_to_tensor(inputs, dtype=tf.float32))
    predictions.extend(preds.numpy())
    true_energies.extend(targets.numpy())

# Convert results to numpy arrays
predictions = np.array(predictions).flatten()
true_energies = np.array(true_energies)
```

```python
# Plot the results
plt.figure(figsize=(8, 6))
plt.scatter(true_energies, predictions, alpha=0.7, label="Predicted vs. True")
plt.plot([min(true_energies), max(true_energies)],
         [min(true_energies), max(true_energies)],
         color='red', linestyle='--', label='Perfect Prediction')
plt.xlim(0, 2)
plt.ylim(0, 2)
plt.xlabel("True Energy")
plt.ylabel("Predicted Energy")
plt.title("PINN: True vs. Predicted Energy (Ground State)")
plt.legend()
plt.grid(alpha=0.5)
plt.show()
```

TRUE VS. PREDICTED ENERGY

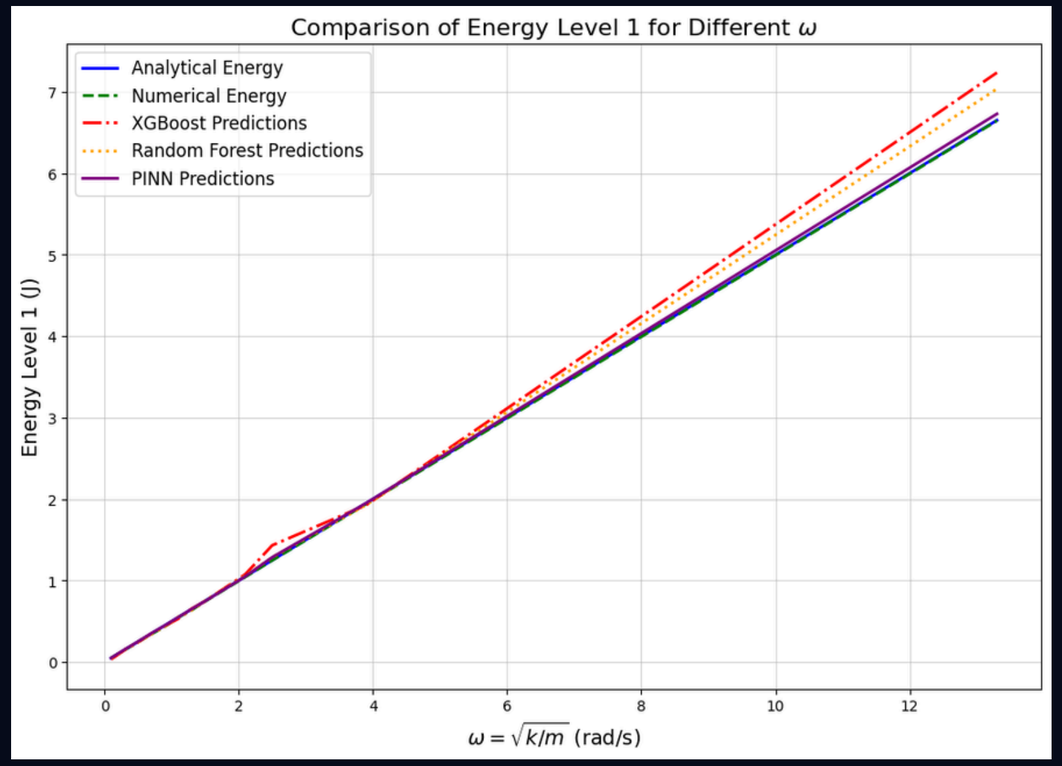


As we can see that even at the higher energy values of the ground state,
PINN is closely following the perfect prediction line,
which was expected from the model as
it is particulary accurate even in those areas where training data is sparse.

COMPARISION ACROSS METHODS

```
# Predictions using PINN
pinn_predictions = []
for m, k in zip(m_values, k_values):
    sample_input = tf.convert_to_tensor([[m, k]], dtype=tf.float32)
    pinn_prediction = pinn(sample_input).numpy()[0]  # Predict energy level 1
    pinn_predictions.append(pinn_prediction)
pinn_energy = np.array(pinn_predictions)

# Plot results
import matplotlib.pyplot as plt

plt.figure(figsize=(12, 8))
plt.plot(omega_values, analytical_energy, label="Analytical Energy", color='blue', linewidth=2)
plt.plot(omega_values, numerical_energy, label="Numerical Energy", color='green', linestyle='--', linewidth=2)
plt.plot(omega_values, XGB_energy, label="XGBoost Predictions", color='red', linestyle='-.', linewidth=2)
plt.plot(omega_values, RF_energy, label="Random Forest Predictions", color='orange', linestyle=':', linewidth=2)
plt.plot(omega_values, pinn_energy, label="PINN Predictions", color='purple', linestyle='-', linewidth=2)

plt.xlabel(r"$\omega = \sqrt{k/m}$ (rad/s)", fontsize=14)
plt.ylabel("Energy Level 1 (J)", fontsize=14)
plt.title("Comparison of Energy Level 1 for Different $\omega$", fontsize=16)
plt.legend(fontsize=12)
plt.grid(alpha=0.5)
plt.show()
```

PLOT FOR GROUND STATE VS. OMEGA



As we can see what we expected earlier came true in our results also.
PINN is able to out perform both
the traditional machine learning models XGBoost and Random Forest.
It is able to perform better in higher values of
ω and closely follow the numerical data line.
This is the situation when we have used the lighter training dataset
with lesser number of training cases,
if we would have used a better training dataset
then the performance of the PINN model would have been even better.
Hence we can conclude that PINNs can be used for predicting the energy levels
from m and k values relieving us from doing the computationally expensive work
of finding the numerical solutions again and again.

CALCULATING ERRORS

```
mse_rf = mean_squared_error(numerical_energy,RF_energy)
mse_xgb = mean_squared_error(numerical_energy,XGB_energy)
mse_pinn = mean_squared_error(numerical_energy,pinn_energy)

# Data for the table
column_1 = ['Physics Informed NN', 'Random Forest Regression', 'Extreme Gradient Boosting']
column_2 = [round(mse_pinn,5), round(mse_rf,5), round(mse_xgb,5)]  # Replace with your values

# Combine the data into a list of rows
data = list(zip(column_1, column_2))

fig, ax = plt.subplots(figsize=(6, 6))  # Adjust the size here

# Hide axes as we don't need them for the table
ax.axis('off')

# Create the table
table = ax.table(cellText=data, colLabels=['Models', 'Mean Squared Error'], loc='center')

plt.figure(figsize=(6, 1))

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

Models	Mean Squared Error
Physics Informed NN	0.00077
Random Forest Regression	0.01295
Extreme Gradient Boosting	0.03213

As we can see the data of Mean Squared error of the models agrees with our interpretation of the above plot.

WAVE FUNCTION PREDICTION!!!

PINNs performed exceptionally well in predicting the energy levels
of a harmonic oscillator, but can we extend this further?
The wavefunction is a vital component of quantum systems,
providing a complete description of a particle's state.
It allows us to compute probabilities, expectation values, and physical observables ,
which are crucial for interpreting experimental results.
Predicting wavefunctions also enables the analysis of phenomena like
tunneling, interference, and energy quantization.
In the following code cells, we use PINNs to predict the wavefunction of
a 1-D harmonic oscillator in its ground state.

WAVE FUNCTION PREDICTION!!!

We define a neural network with 2 hidden layers, each having 64 neurons,
and output $c \cdot \psi(x)$, where c is a constant.
The Schrodinger equation satisfied by $\psi(x)$ also holds for $c \cdot \psi(x)$, hence the inclusion of c.
The loss function incorporates the Schrodinger equation
and the boundary condition $\psi(x) \to 0$ at boundaries.
These conditions ensure physics-consistent solutions. Unlike standard neural networks,
PINNs embed physical laws into the loss function, enabling physics-aware learning.

WAVE FUNCTION PREDICTION!!!

```python
from tensorflow.keras import models

hbar = 1.0
m = 1.0
k = 1.0
omega = np.sqrt(k / m)

x_values = np.linspace(-5, 5, 200).reshape(-1, 1)

# Define the Neural Network
def create_pinn():
    model = models.Sequential([
        layers.InputLayer(input_shape=(1,)),
        layers.Dense(64, activation='tanh'),
        layers.Dense(64, activation='tanh'),
        layers.Dense(1, activation=None)  # Output: ψ(x)
    ])
    return model
```

```python
model = create_pinn()
E = 0.5 * hbar * omega  # Example energy level (ground state)
# Physics-Informed Loss Function
def pinn_loss(model, x):
    with tf.GradientTape() as tape1:
        tape1.watch(x)
        with tf.GradientTape() as tape2:
            tape2.watch(x)
            psi = model(x)  # ψ(x)
        dpsi_dx = tape2.gradient(psi, x)  # ∂ψ/∂x
    d2psi_dx2 = tape1.gradient(dpsi_dx, x)  # ∂²ψ/∂x²
```

```python
# Schrödinger equation residual
potential = 0.5 * k * x**2
schrodinger_residual = -hbar**2 / (2 * m) * d2psi_dx2 + potential * psi - E * psi

# Physics Loss
physics_loss = tf.reduce_mean(tf.square(schrodinger_residual))

# Boundary Condition Loss (ψ → 0 at boundaries)
boundary_loss = tf.reduce_mean(tf.square(model(tf.constant([[-5.0], [5.0]]))))

return physics_loss + boundary_loss
```

WAVE FUNCTION PREDICTION!!!

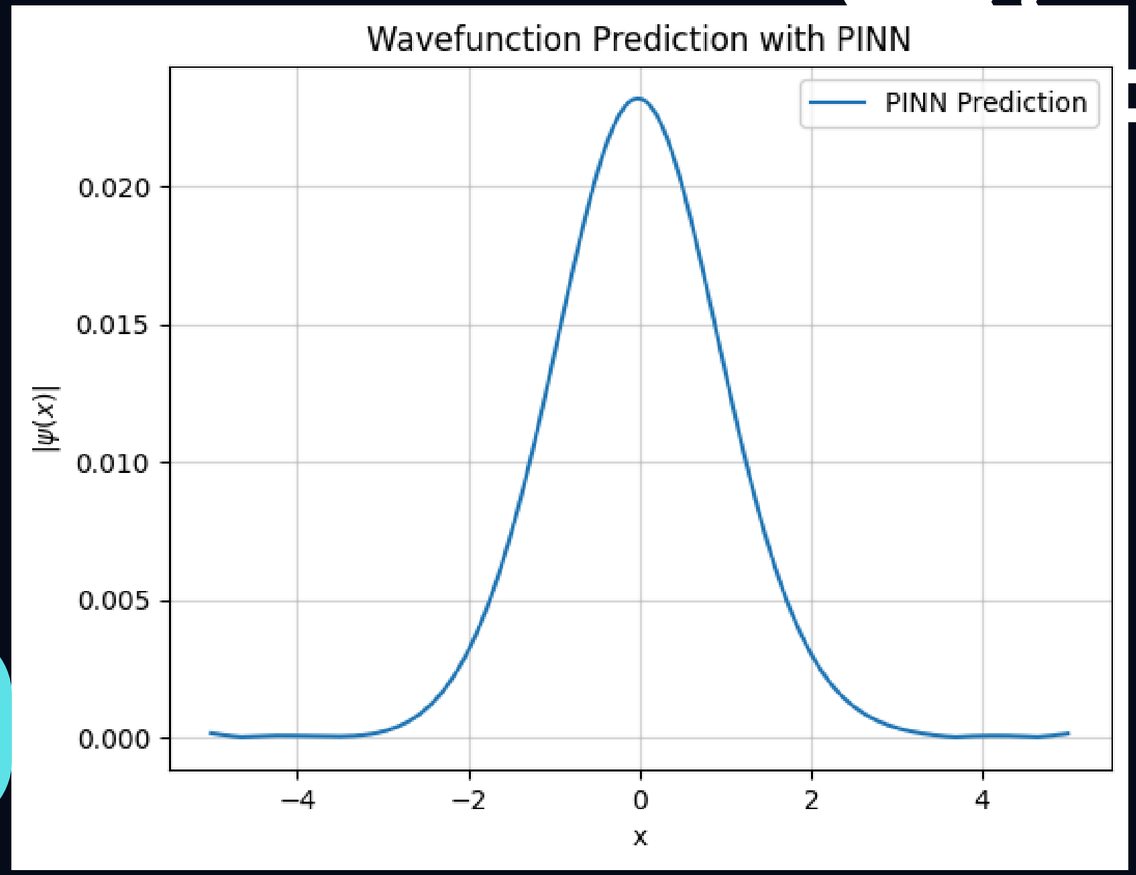
```python
# Training Loop
E = 0.5 * hbar * omega  # Example energy level (ground state)
optimizer = tf.keras.optimizers.Adam(learning_rate=0.01)

@tf.function
def train_step(x):
    with tf.GradientTape() as tape:
        loss = pinn_loss(model, x)
    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))
    return loss

# Training
epochs = 5000
for epoch in range(epochs):
    loss = train_step(tf.constant(x_values, dtype=tf.float32))
    if epoch % 500 == 0:
        print(f"Epoch {epoch}: Loss = {loss.numpy():.6f}")
```

```
Epoch 0: Loss = 2.279564
Epoch 500: Loss = 0.000013
Epoch 1000: Loss = 0.000005
Epoch 1500: Loss = 0.000003
Epoch 2000: Loss = 0.000002
Epoch 2500: Loss = 0.000001
Epoch 3000: Loss = 0.000001
Epoch 3500: Loss = 0.000001
Epoch 4000: Loss = 0.000001
Epoch 4500: Loss = 0.000006
```

Here, we set E to the ground state energy of a 1D harmonic oscillator. We use gradient descent to train the PINN model.

WAVE FUNCTION PREDICTION!!!

```
psi_pred = model.predict(x_values)

plt.plot(x_values, abs(psi_pred), label='PINN Prediction')
plt.title("Wavefunction Prediction with PINN")
plt.xlabel("x")
plt.ylabel(r"$|\psi(x)|$")
plt.legend()
plt.grid(alpha=0.5)
plt.show()
```

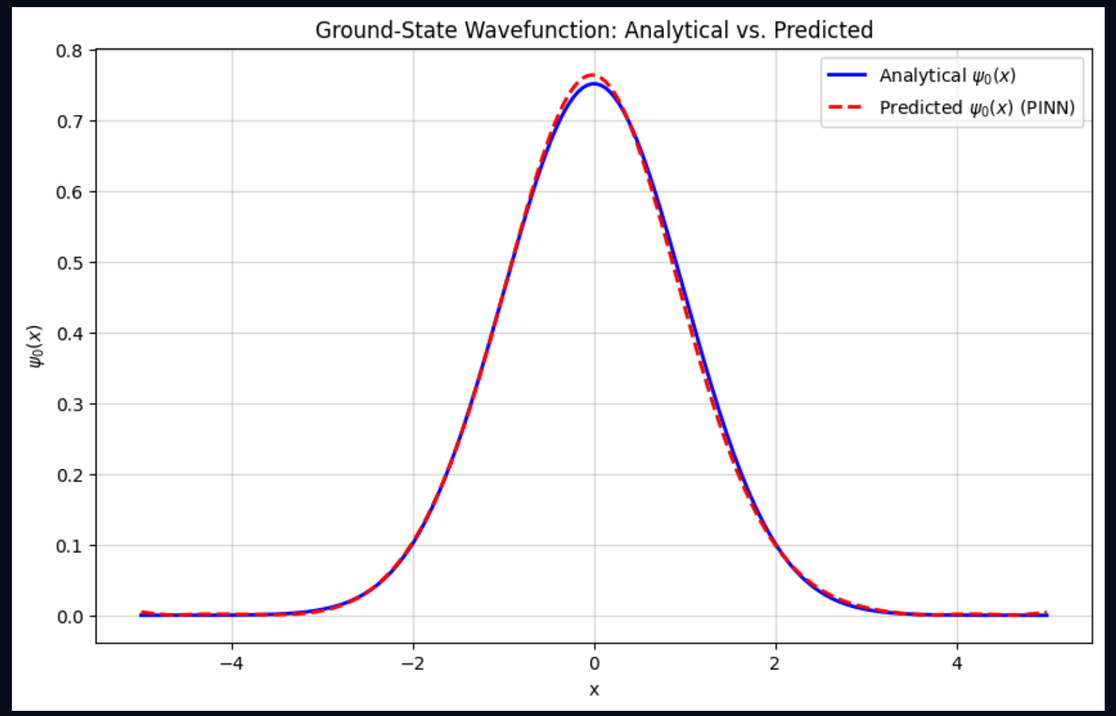


This is the predicted wave function by the PINN model.
Now, let's compare it with the actual solution of the Schrodinger equation for the ground state of a 1D harmonic oscillator, which involves Hermite polynomials.

ANALYTICAL VS. PREDICTED (WF)

```python
# Analytical ground-state wavefunction
def analytical_psi_0(x):
    normalization = (m * omega / (np.pi * hbar))**0.25
    return normalization * np.exp(-m * omega * x**2 / (2 * hbar))

# Predict with PINN
psi_pred = model.predict(x_values).flatten()

# Normalize the predicted wavefunction
psi_pred_normalized = psi_pred / np.sqrt(np.sum(psi_pred**2) * (x_values[1] - x_values[0]))

# Analytical wavefunction
psi_actual = analytical_psi_0(x_values.flatten())

# Plot the results
plt.figure(figsize=(10, 6))
plt.plot(x_values, abs(psi_actual), label="Analytical $\psi_0(x)$", color="blue", linewidth=2)
plt.plot(x_values, abs(psi_pred_normalized), label="Predicted $\psi_0(x)$ (PINN)", color="red", linestyle="--", linewidth=2)
plt.title("Ground-State Wavefunction: Analytical vs. Predicted")
plt.xlabel("x")
plt.ylabel(r"$\psi_0(x)$")
plt.legend()
plt.grid(alpha=0.5)
plt.show()
```

ANALYTICAL VS. PREDICTED (WF)



First, we normalized the predicted $\psi(x)$ to eliminate the constant c factor discussed earlier. Then, we plotted $|\psi(x)|$ vs x.

As seen, the predicted $\psi(x)$ closely follows the analytical $\psi(x)$ throughout the domain. This demonstrates the PINN model's ability to predict wavefunctions for quantum systems, enabling the analytical analysis of even complex systems using ML models.

CONCLUSION

This project demonstrated the use of machine learning models,
including Random Forest, XGBoost, and Physics-Informed Neural Networks (PINNs),
to predict quantum energy levels and wavefunctions.
The results highlight the potential of ML models to efficiently
approximate numerical solutions in quantum mechanics,
reducing computational costs compared to traditional methods.
This work paves the way for extending ML applications to
more complex quantum systems and other physics-driven problems,
offering a scalable approach to tackling scientific challenges.

GITHUB LINK

https://github.com/Soumik969/AIDS

THANK YOU