Levitating Rotor

Objectives in the past week

➲ Full conversion of scipy numerical solver into JAX

➲ Continue exploring simulation parameters (driving frequency, sampling points) to understand physical system

➲ Minor explorations of last week's neural network

➲ Advance simple neural network that predicts the torque of the entire system given varying frequency and theta.

➲ Brief preview of updated physical system (after meeting with collaborators)

FINAL GOAL: Develop a neural network that identifies missing physics of the system. Run experiments with collaborators. Room for creative extensions of project.

Levitating Rotor – Conversion to JAX

Use ode_int

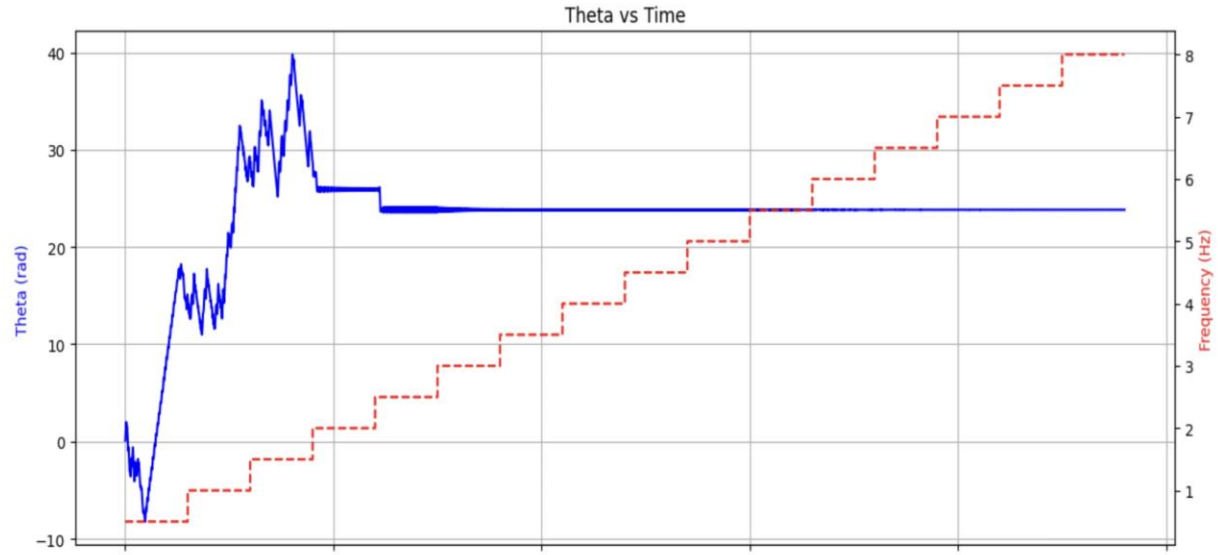
```
import jax
import jax.numpy as jnp
from jax.experimental.ode import odeint
import matplotlib.pyplot as plt
```

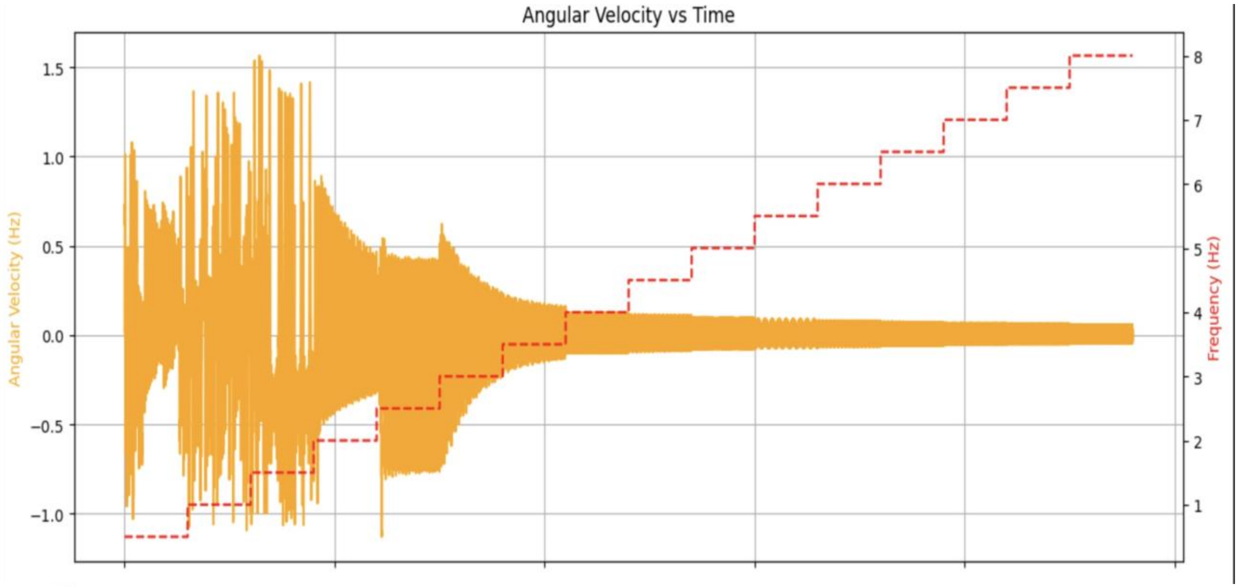
```
@jax.jit
def dynamics(y, t, f):
    theta, theta_dot = y
    torque = compute_torque(theta, t, f)
    theta_ddot = (torque - c * theta_dot) / I
    return jnp.array([theta_dot, theta_ddot])

# Time and frequency sweep
t_span = jnp.linspace(0, 30, 1000)
frequencies = jnp.arange(0.5, 8.5, 0.5)
```

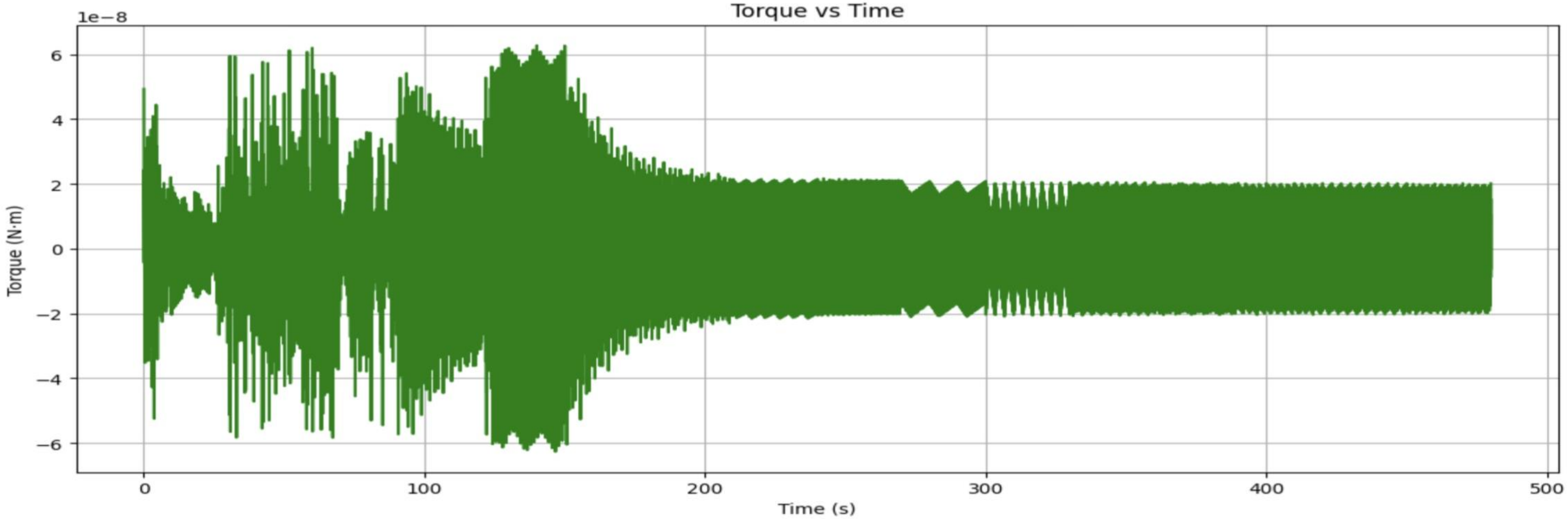
```
y0 = jnp.array([theta_0, theta_dot_0])
current_time_offset = 0.0

for f in frequencies:
    times = t_span + current_time_offset

    sol = odeint(dynamics, y0, t_span, f)

    theta_vals = sol[:, 0]
    theta_dot_vals = sol[:, 1]

    # Compute torques
    torque_vals = jax.vmap(lambda th, t: compute_torque(th, t, f))(theta_vals, t_span)

    all_theta.append(theta_vals)
    all_theta_dot.append(theta_dot_vals)
    all_torque.append(torque_vals)
    all_time.append(times)
    all_frequency.append(jnp.full_like(times, f))

    y0 = sol[-1]
    current_time_offset += 30.0
```

Levitating Rotor – Conversion to JAX

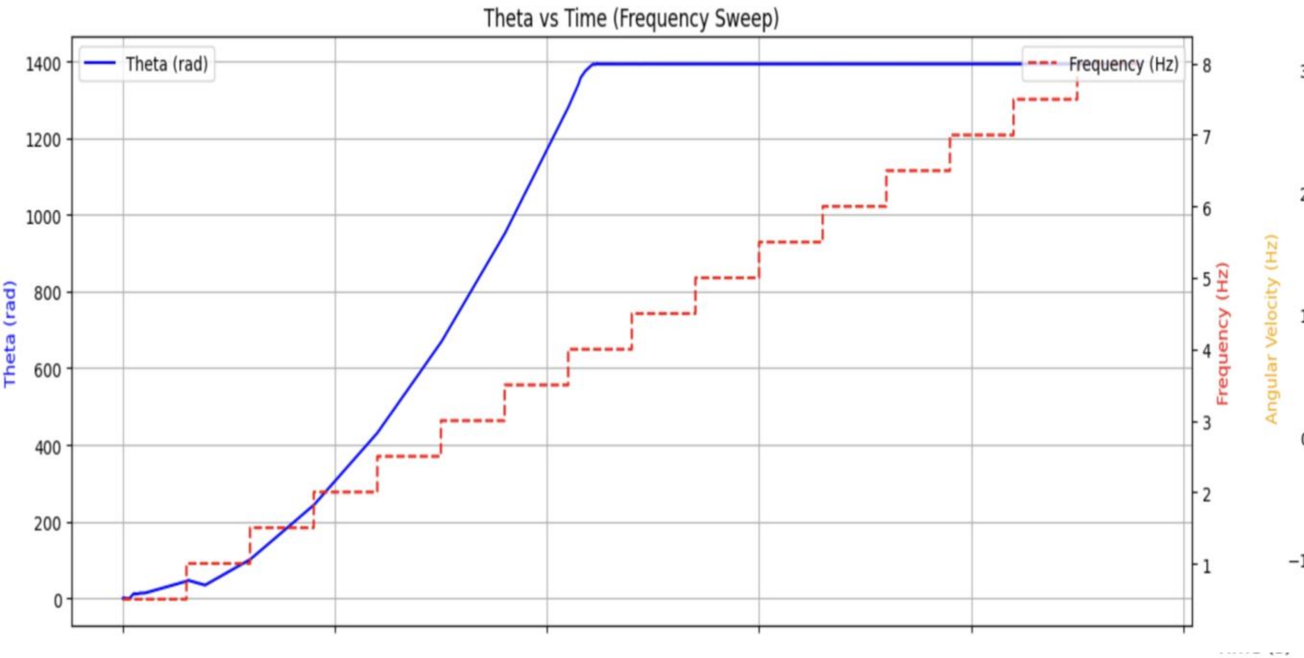
Use ode_int. Runtime = 5 mins

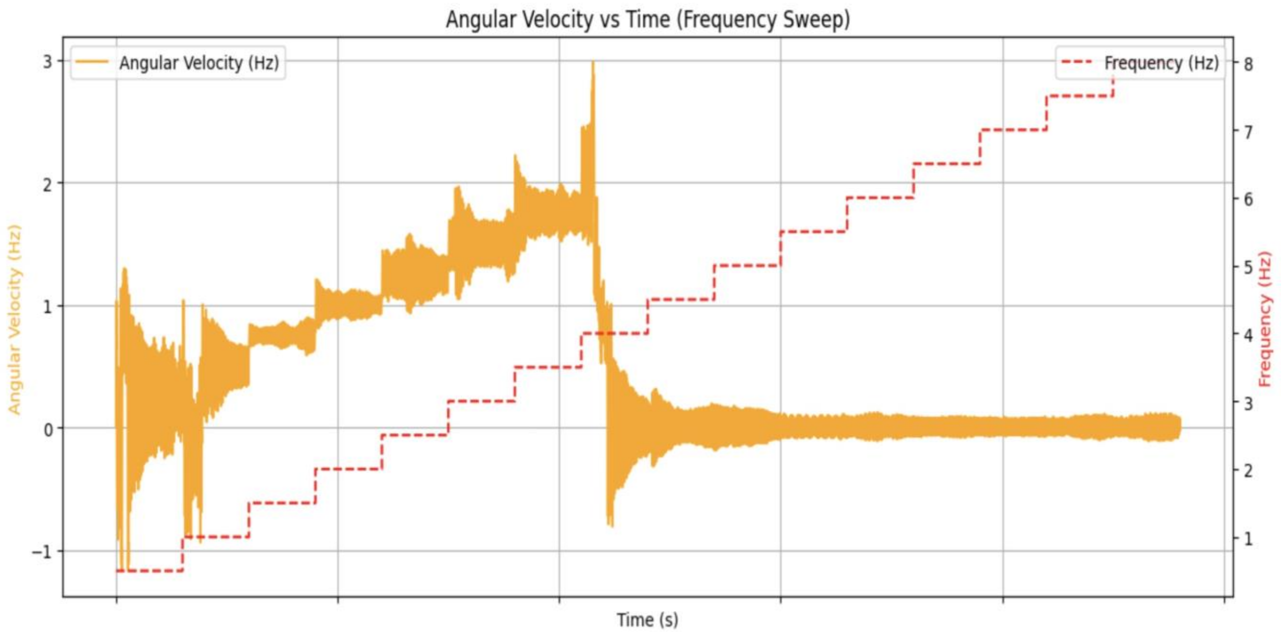


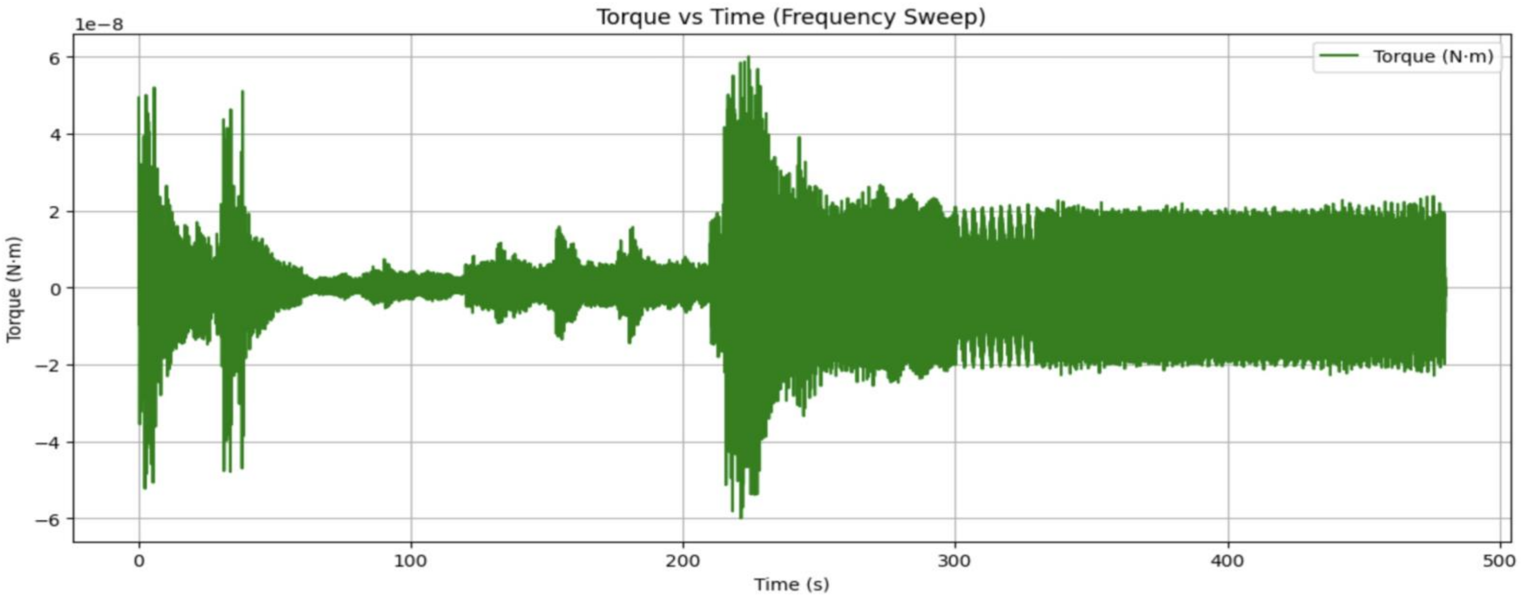




Original Scipy Outputs. Runtime = 4 mins







Levitating Rotor – Conversion to JAX

Use RK4. Runtime = 13s

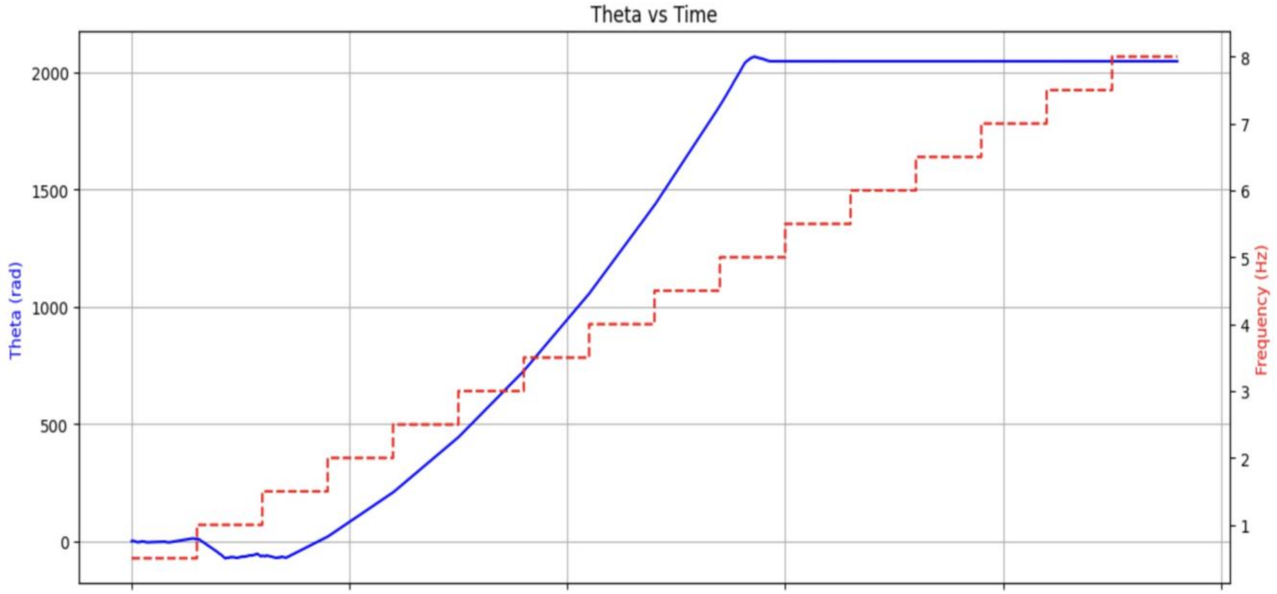
```
import jax
import jax.numpy as jnp
from jax import jit, vmap
from jax.lax import scan
import matplotlib.pyplot as plt
```

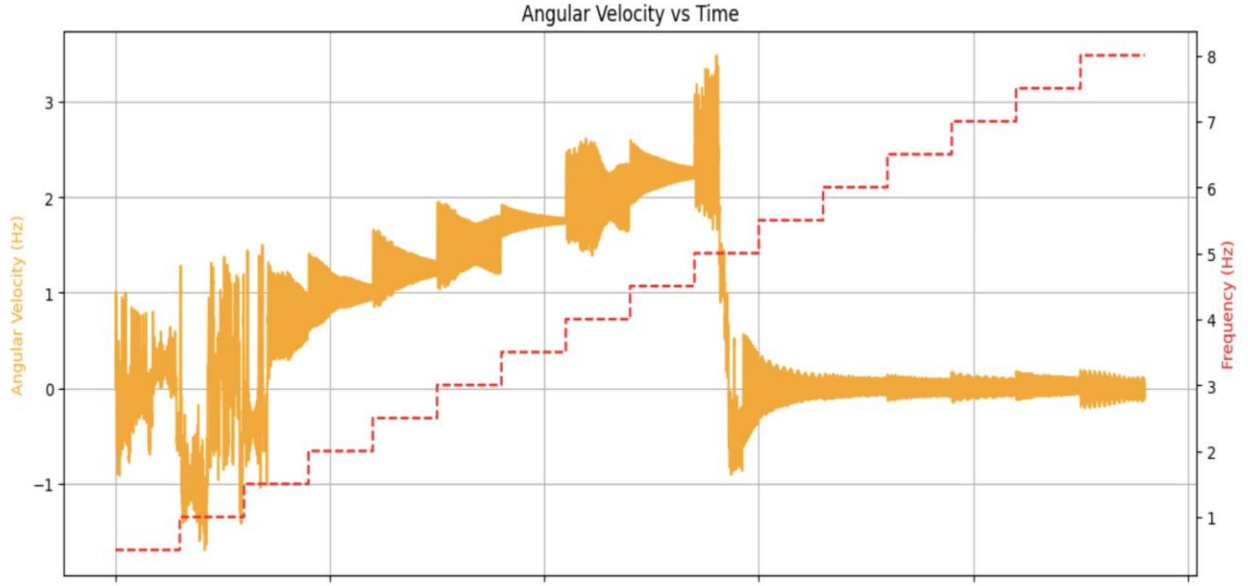
```
# Time and frequency sweep setup
t_segment = jnp.linspace(0, 30, 1000)
frequencies = jnp.arange(0.5, 8.5, 0.5)
y0 = jnp.array([0.1, 0.1])  # Initial [theta, theta_dot]

# Storage for full results
all_theta = []
all_theta_dot = []
all_torque = []
all_time = []
all_frequency = []

offset_time = 0.0

for freq in frequencies:
    times = t_segment + offset_time
    ys = rk4_integrate(y0, t_segment, freq)
    thetas = ys[:, 0]
    theta_dots = ys[:, 1]
    torques = vmap(lambda th, t: compute_torque(th, t, freq))(thetas, t_segment)

    all_theta.append(thetas)
    all_theta_dot.append(theta_dots)
    all_torque.append(torques)
    all_time.append(times)
    all_frequency.append(jnp.full_like(t_segment, freq))

    y0 = ys[-1]
    offset_time += 30.0
```

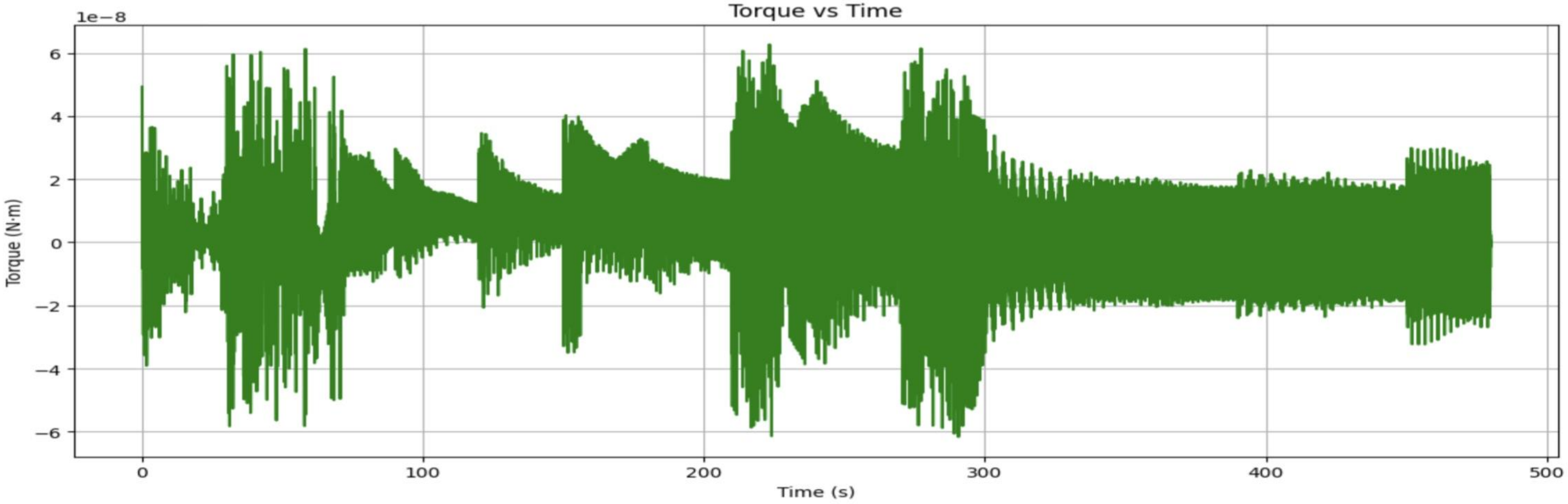
```
@jit
def dynamics(y, t, f):
    theta, theta_dot = y
    tau = compute_torque(theta, t, f)
    theta_ddot = (tau - c * theta_dot) / I
    return jnp.array([theta_dot, theta_ddot])

# Not jitted — uses a closure to avoid dynamic argument issues
def rk4_step(dynamics_fn, y, t, dt):
    k1 = dynamics_fn(y, t)
    k2 = dynamics_fn(y + 0.5 * dt * k1, t + 0.5 * dt)
    k3 = dynamics_fn(y + 0.5 * dt * k2, t + 0.5 * dt)
    k4 = dynamics_fn(y + dt * k3, t + dt)
    return y + dt / 6.0 * (k1 + 2*k2 + 2*k3 + k4)

@jit
def rk4_integrate(y0, t_array, frequency):
    dt = t_array[1] - t_array[0]

    def dyn(y, t): return dynamics(y, t, frequency)

    def step_fn(y, t):
        y_next = rk4_step(dyn, y, t, dt)
        return y_next, y

    _, ys = scan(step_fn, y0, t_array[:-1])
    ys = jnp.vstack([y0, ys])
    return ys
```

Levitating Rotor – Conversion to JAX

Use RK4. Runtime = 13s







Levitating Rotor – 2nd Round of Tests

First Round of Tests

Since angular velocity corresponds to frotor = (fvolt)/2, maximum rotor frequency we should go up to is 4.25Hz (1/2 of fvolt = 8.5Hz). Default t_span = 30s with 1000 points sampled achieves highest rotor frequency of 2.5Hz. This is 33.33 points per sec and 13.33 points sampled per cycle for frotor = 2.5Hz. So ideal sampling is 15 points per cycle. That is 4.25 x 15 = 63.75 sampling points per sec. Which is 1913 points per 30 secs.

Modification 1: 0.5Hz step-up in frequencies AND 1900 sampling points across 30s. (5 mins runtime) Result -- No difference.

Modification 2: 0.05Hz step-up in frequencies AND 1000 sampling points across 30s (>10 mins runtime).

Modification 3: 0.1Hz step-up in frequencies AND 1000 sampling points across 30s (>10 mins runtime).

Modification 4: 0.2Hz step-up in frequencies AND 1000 sampling points across 30s (>10 mins runtime).

Modification 5: 0.2Hz step-up in frequencies AND 500 sampling points over 30s. (10 mins runtime) Result – Max angular velocity near 4.0Hz. Max theta above 5000 rad.

Modification 6: 0.2Hz step-up in frequencies AND 500 sampling points over 20s. (6 mins runtime) Result – Max angular velocity > 4.0Hz. Max theta above 4000 rad.

Modification 7: 0.2Hz step-up in frequencies AND 500 sampling points over 10s. (2 mins runtime) Result – Max angular velocity > 12.0Hz. Max theta above 8000 rad.

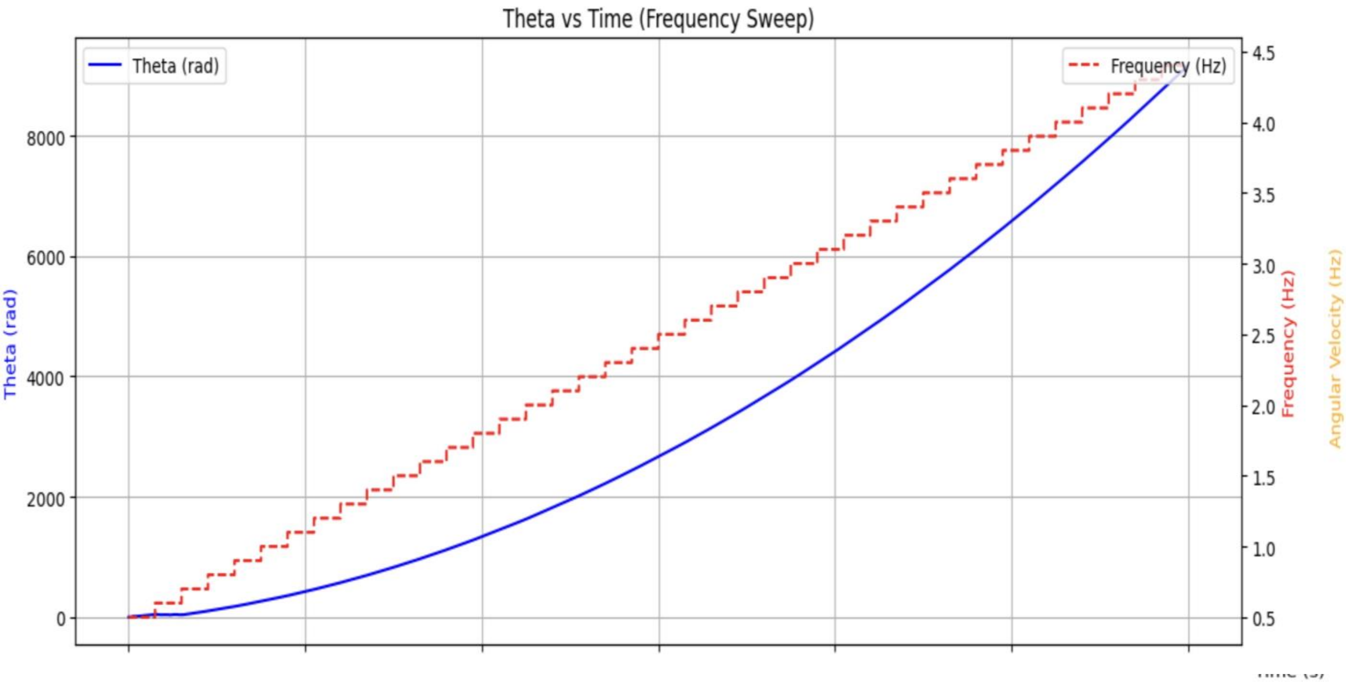
Modification 8: 0.2Hz step-up in frequencies AND 500 sampling points over 5s. Result -- Max angular velocity > 4.0Hz. Max theta above 1200 rad.

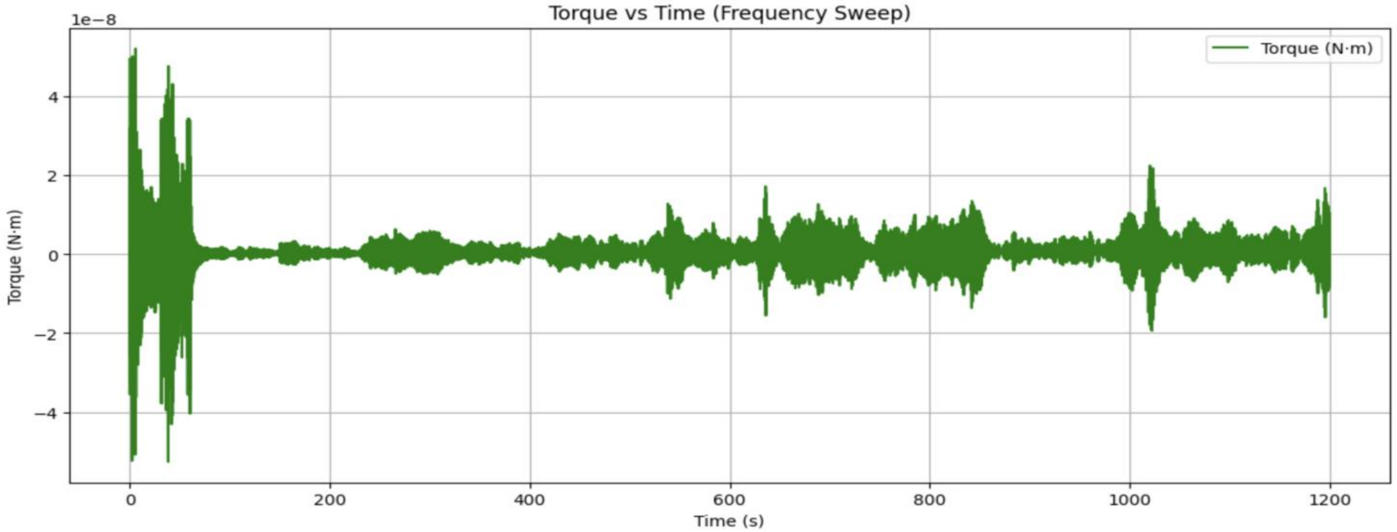
Modification 9: 0.1Hz step-up in frequencies AND 500 sampling points over 5s. (3 mins runtime) Result – Max angular velocity > 4.0Hz. Max theta > 2500rad

Modification 10: 0.1Hz step-up in frequencies AND 500 sampling points over 10s. (8mins runtime) Result – Max angular velocity around 2.5Hz. Max theta > 1750rad.

```
**2nd Round of Tests**

Test 1: 0.5Hz step-up in frequencies AND 2200 sampling points across 30s.
(runtime 5 mins)
Result -- Max angular velocity near 5Hz. Max theta > 1750 rad.
Test 2: 0.2Hz step-up in frequencies AND 1000 sampling points across 10s. Max
driving f = 4.5Hz. (runtime 1 min)
Result -- Max angular velocity >2.0Hz. Max theta near 1000 rad.
Test 3: 0.2Hz step-up in frequencies AND 1000 sampling points across 5s. Max
driving f = 4.5Hz. (runtime 1 min)
Result -- Max angular velocity >8Hz. Max theta >1000 rad.
Test 4: 0.1Hz step-up AND 1000 sampling points across 30s for driving
frequency up to 4.5Hz. To test runtime. (runtime 2 mins)
Result -- Linear increase in angular velocity. Max theta > 8000rad. Max ang
vel > 2.5 Hz
Test 5: 0.1Hz step-up AND 1000 sampling points across 20s for driving
frequency up to 4.5Hz. (runtime 2 mins)
Result -- Flatter linear increase in angular velocity. Max theta > 5000 rad.
Max ang vel > 3Hz

Inference: Denser points higher accuracy
```

Levitating Rotor – 2nd Round of Tests (Test 4)

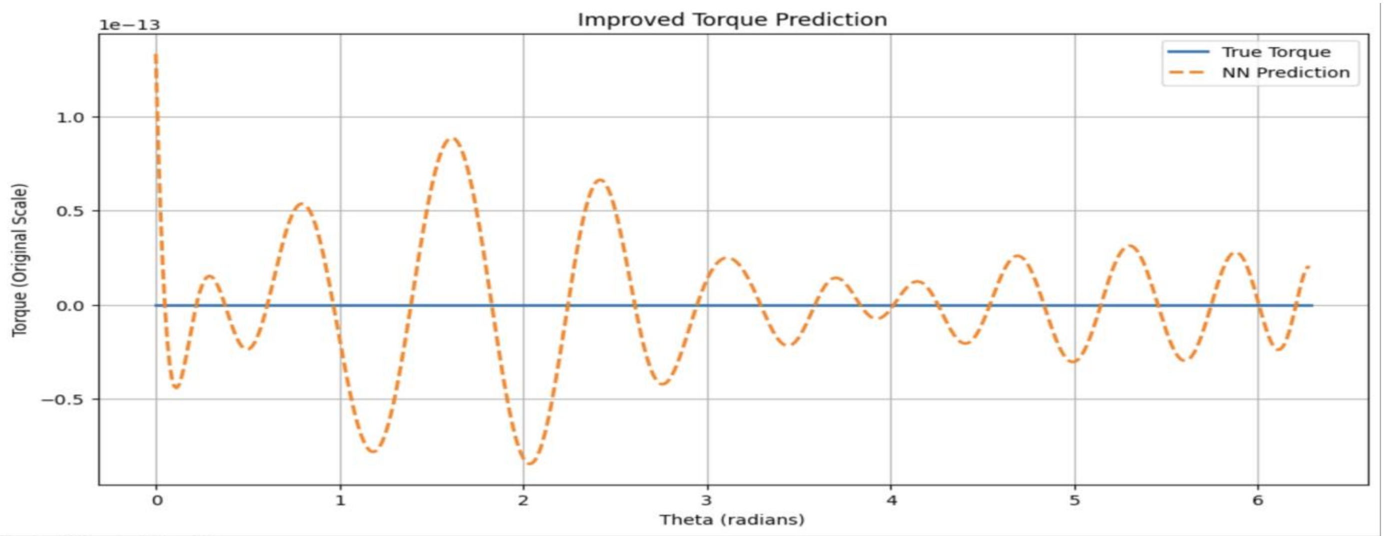




Levitating Rotor – Last Week's NN

```
Epoch 0, Loss: 0.3449
Epoch 100, Loss: 0.0002
Epoch 200, Loss: 0.0000
Epoch 300, Loss: 0.0000
Epoch 400, Loss: 0.0000
Epoch 500, Loss: 0.0000
Epoch 600, Loss: 0.0000
Epoch 700, Loss: 0.0000
Epoch 800, Loss: 0.0000
Epoch 900, Loss: 0.0000
Epoch 1000, Loss: 0.0000
Epoch 1100, Loss: 0.0000
Epoch 1200, Loss: 0.0000
Epoch 1300, Loss: 0.0000
Epoch 1400, Loss: 0.0000
Epoch 1500, Loss: 0.0000
Epoch 1600, Loss: 0.0000
Epoch 1700, Loss: 0.0000
Epoch 1800, Loss: 0.0000
Epoch 1900, Loss: 0.0000
Epoch 2000, Loss: 0.0000
Epoch 2100, Loss: 0.0000
Epoch 2200, Loss: 0.0000
Epoch 2300, Loss: 0.0000
Epoch 2400, Loss: 0.0000
Epoch 2500, Loss: 0.0000
Epoch 2600, Loss: 0.0000
Epoch 2700, Loss: 0.0000
Epoch 2800, Loss: 0.0000
Epoch 2900, Loss: 0.0000
```

```python
class TorquePredictor(nn.Module):
    @nn.compact
    def __call__(self, x):
        # Rich periodic encoding applied within layers
        x = nn.Dense(256)(x)
        x = nn.swish(x)

        # Add sinusoidal transformations at this stage
        x = x + jnp.sin(x) + jnp.cos(x)  # Apply sin and cos to output of first layer

        # Second layer with periodic encoding
        x = nn.Dense(128)(x)
        x = nn.swish(x)

        # Add sinusoidal transformations again
        x = x + jnp.sin(x) + jnp.cos(x)  # Apply sin and cos to output of second layer

        # Third layer with periodic encoding
        x = nn.Dense(64)(x)
        x = nn.swish(x)

        # Final sinusoidal transformation before output
        x = x + jnp.sin(x) + jnp.cos(x)  # Apply sin and cos to output of third layer

        # Final output layer
        return nn.Dense(1)(x)  # Linear output
```

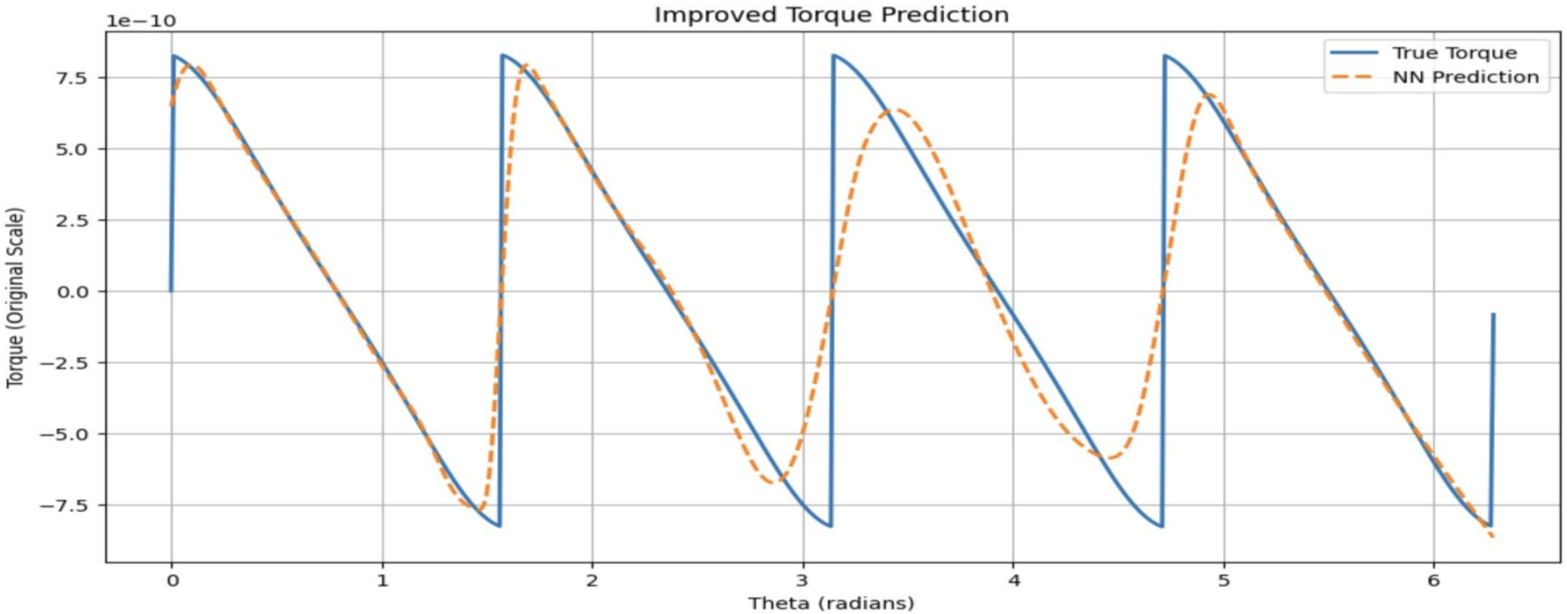



```
Test MSE: 1.216e-27
Max Error: 1.331e-13
```

Levitating Rotor – Last Week's NN

```
Epoch 0, Loss: 27.0334
Epoch 100, Loss: 25.9287
Epoch 200, Loss: 25.3520
Epoch 300, Loss: 24.0382
Epoch 400, Loss: 21.9584
Epoch 500, Loss: 21.0280
Epoch 600, Loss: 20.0743
Epoch 700, Loss: 18.7347
Epoch 800, Loss: 17.2689
Epoch 900, Loss: 15.8151
Epoch 1000, Loss: 14.5447
Epoch 1100, Loss: 13.4509
Epoch 1200, Loss: 12.3741
Epoch 1300, Loss: 11.1195
Epoch 1400, Loss: 9.7432
Epoch 1500, Loss: 8.4855
Epoch 1600, Loss: 7.4175
Epoch 1700, Loss: 6.5567
Epoch 1800, Loss: 5.8855
Epoch 1900, Loss: 5.3845
Epoch 2000, Loss: 4.9970
Epoch 2100, Loss: 4.7092
Epoch 2200, Loss: 4.4915
Epoch 2300, Loss: 4.3178
Epoch 2400, Loss: 4.1726
Epoch 2500, Loss: 4.0485
Epoch 2600, Loss: 3.9397
Epoch 2700, Loss: 3.8416
Epoch 2800, Loss: 3.7512
Epoch 2900, Loss: 3.6636
```

```python
# ============================================
# Enhanced Model Architecture
# ============================================
class TorquePredictor(nn.Module):
    @nn.compact
    def __call__(self, x):
        # Rich periodic encoding
        #x = jnp.concatenate([
            #x,
            #jnp.sin(x), jnp.cos(x),
            #jnp.sin(2*x), jnp.cos(2*x),
            #jnp.sin(4*x), jnp.cos(4*x)
        #], axis=-1)

        # Larger network with residual connections
        x = nn.Dense(256)(x)
        x = nn.swish(x) #Swish is better than ReLU for physics problems.
        x = nn.Dense(128)(x)
        x = nn.swish(x)
        x = nn.Dense(64)(x)
        x = nn.swish(x)
        return nn.Dense(1)(x)  # Linear output
```

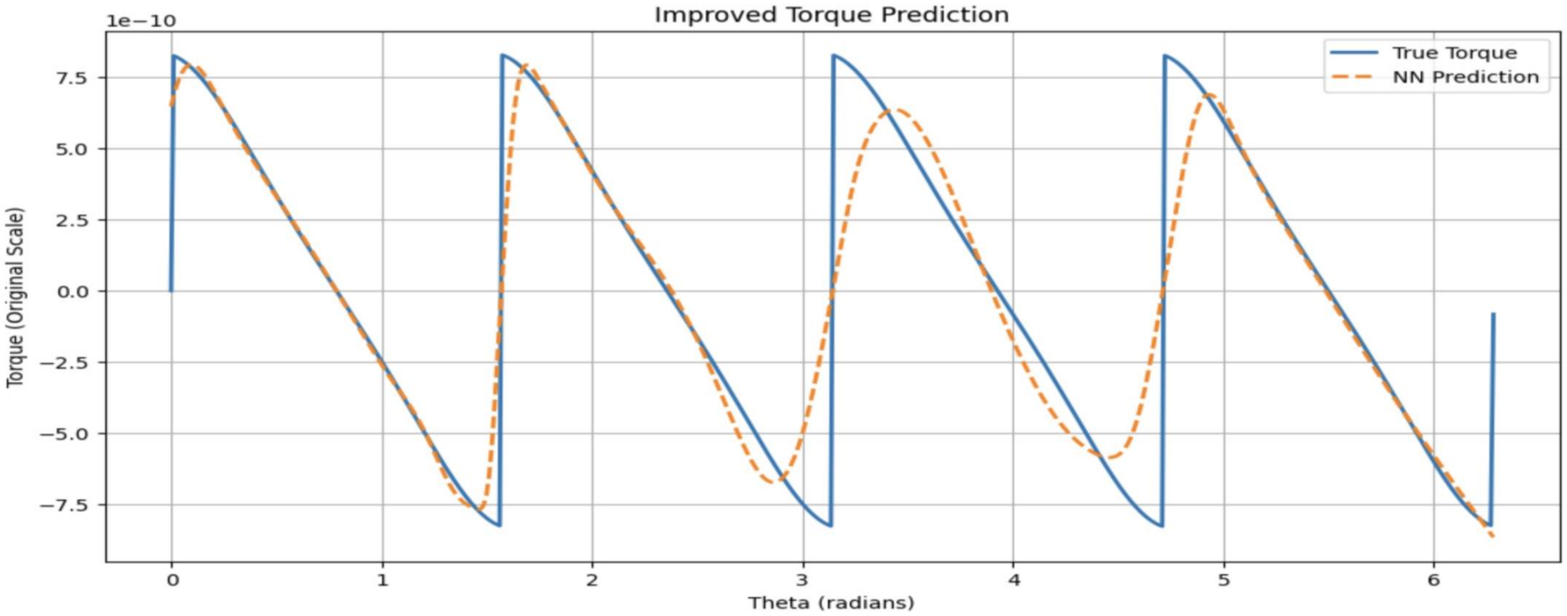


Levitating Rotor – Last Week's NN

Both scaling and swish

```
Epoch 0, Loss: 27.0334
Epoch 100, Loss: 25.9287
Epoch 200, Loss: 25.3520
Epoch 300, Loss: 24.0382
Epoch 400, Loss: 21.9584
Epoch 500, Loss: 21.0280
Epoch 600, Loss: 20.0743
Epoch 700, Loss: 18.7347
Epoch 800, Loss: 17.2689
Epoch 900, Loss: 15.8151
Epoch 1000, Loss: 14.5447
Epoch 1100, Loss: 13.4509
Epoch 1200, Loss: 12.3741
Epoch 1300, Loss: 11.1195
Epoch 1400, Loss: 9.7432
Epoch 1500, Loss: 8.4855
Epoch 1600, Loss: 7.4175
Epoch 1700, Loss: 6.5567
Epoch 1800, Loss: 5.8855
Epoch 1900, Loss: 5.3845
Epoch 2000, Loss: 4.9970
Epoch 2100, Loss: 4.7092
Epoch 2200, Loss: 4.4915
Epoch 2300, Loss: 4.3178
Epoch 2400, Loss: 4.1726
Epoch 2500, Loss: 4.0485
Epoch 2600, Loss: 3.9397
Epoch 2700, Loss: 3.8416
Epoch 2800, Loss: 3.7512
Epoch 2900, Loss: 3.6636
```

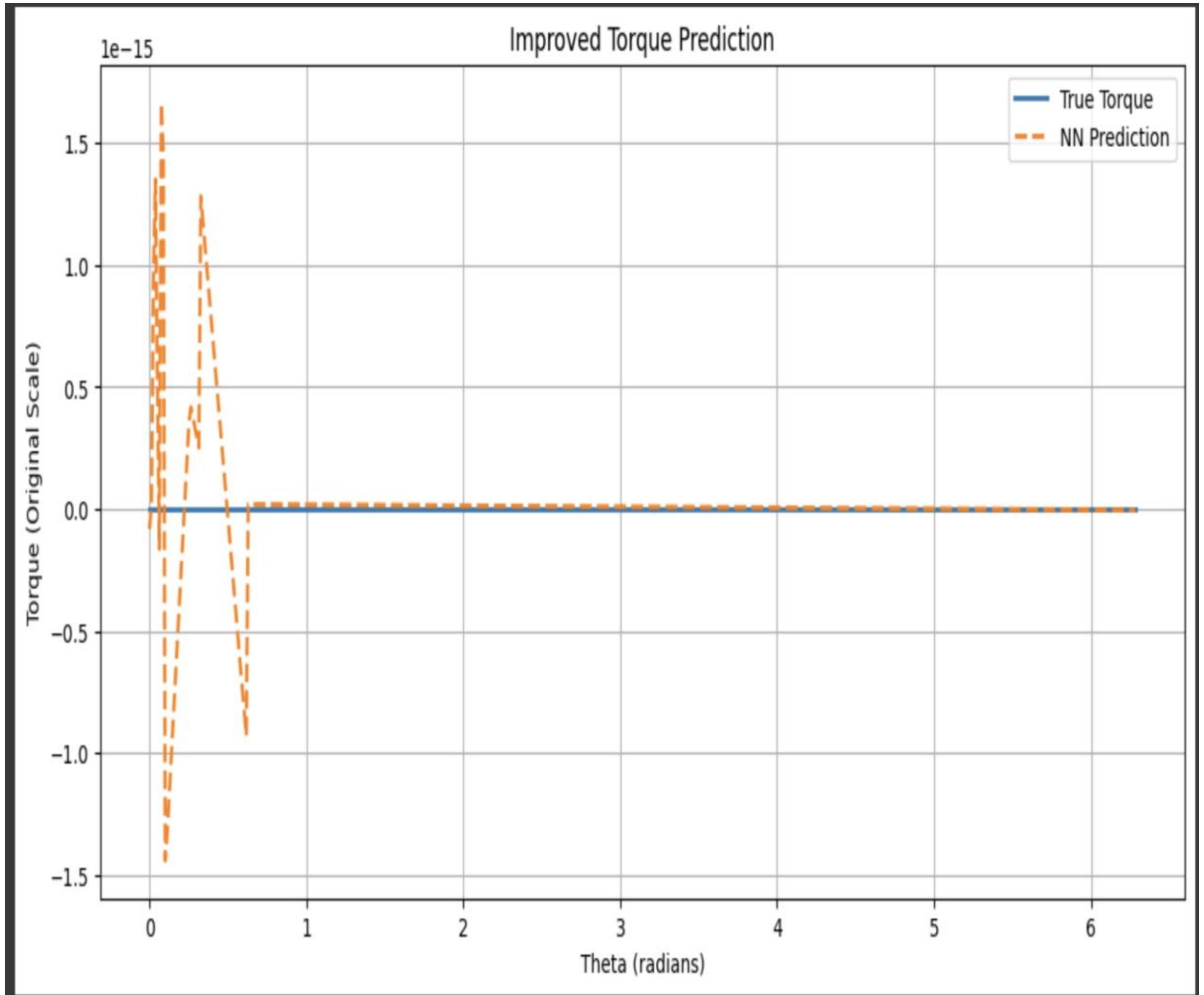
```python
# ============================================
# Enhanced Model Architecture
# ============================================
class TorquePredictor(nn.Module):
    @nn.compact
    def __call__(self, x):
        # Rich periodic encoding
        #x = jnp.concatenate([
            #x,
            #jnp.sin(x), jnp.cos(x),
            #jnp.sin(2*x), jnp.cos(2*x),
            #jnp.sin(4*x), jnp.cos(4*x)
        #], axis=-1)

        # Larger network with residual connections
        x = nn.Dense(256)(x)
        x = nn.swish(x) #Swish is better than ReLU for physics problems.
        x = nn.Dense(128)(x)
        x = nn.swish(x)
        x = nn.Dense(64)(x)
        x = nn.swish(x)
        return nn.Dense(1)(x)  # Linear output
```



Levitating Rotor – Last Week's NN

Only scaling

```
# ==============================================
# Enhanced Model Architecture
# ==============================================
class TorquePredictor(nn.Module):
    @nn.compact
    def __call__(self, x):
        # Rich periodic encoding
        #x = jnp.concatenate([
            #x,
            #jnp.sin(x), jnp.cos(x),
            #jnp.sin(2*x), jnp.cos(2*x),
            #jnp.sin(4*x), jnp.cos(4*x)
        #], axis=-1)

        # Larger network with residual connections
        x = nn.Dense(256)(x)
        x = nn.relu(x) #Swish is better than ReLU for physics problems.
        x = nn.Dense(128)(x)
        x = nn.relu(x)
        x = nn.Dense(64)(x)
        x = nn.relu(x)
        return nn.Dense(1)(x)  # Linear output
```



Test MSE: 5.641e-32
Max Error: 1.667e-15

Levitating Rotor – 2D NN

```
import jax
import jax.numpy as jnp
from flax import linen as nn
from flax.training import train_state
import optax
from functools import partial
from jax import config
config.update("jax_enable_x64", True) #Critical for tiny values
import matplotlib.pyplot as plt
```

```
def compute_torques_scalar(theta, t, f):
    angle_diff = theta + arm_angles[:, jnp.newaxis] - electrode_angles[jnp.newaxis, :]
    sin_half_angle_diff = jnp.sin(angle_diff / 2)
    distances_squared = d0_squared + (R2 * sin_half_angle_diff)**2
    voltages_squared = electrode_voltages_precomputed(t, f)**2
    torques = R_epsilon_A * voltages_squared / distances_squared * jnp.sign(jnp.sin(angle_diff))
    return jnp.sum(torques)

# Define a function that takes theta and f and uses the fixed t
def compute_torques(theta, f):
    return compute_torques_scalar(theta, t=20.0, f=f)
```

```
# ==========================================
# Enhanced Scaling (Handles Tiny Values)
# ==========================================
def scale_torque(torque):
    log_torque = jnp.log10(jnp.abs(torque) + 1e-20)  # Adjust based on your torque magnitude
    scaled = log_torque / jnp.max(jnp.abs(jnp.log10(jnp.abs(torque) + 1e-20))) #Normalize to [-1, 1]
    return scaled, None

# Generate the training data for theta
theta_train = jnp.linspace(0, 2 * jnp.pi, 1000)

# Generate the training data for frequency f_train
f_train = jnp.linspace(0.5, 8.5, 1000)

# Use vmap to apply compute_torques over the inputs
torque_train = jax.vmap(compute_torques, in_axes=(0, 0))(theta_train, f_train)

# Scale torque values
torque_train_scaled, torque_scaled = scale_torque(torque_train)

# Reshape data
theta_train = theta_train.reshape(-1, 1)
torque_train_scaled = torque_train_scaled.reshape(-1, 1)
f_train = f_train.reshape(-1, 1)  # Reshape f_train
```

Levitating Rotor – 2D NN

```python
# ==============================================
# Evaluation
# ==============================================

# Generate theta_test values
theta_test = jnp.linspace(0, 2 * jnp.pi, 5).reshape(-1, 1)
f_test = jnp.linspace(0.5, 8.5, 5).reshape(-1, 1)

# Compute true torque with a specific frequency for comparison
torque_true = jax.vmap((compute_torques))(theta_test.squeeze(), f_test.squeeze())

print(torque_true)

# Predict using the trained model, including f_test
torque_pred_scaled = model.apply(state.params, theta_test, f_test)

# Reverse scaling to get the predicted torque
torque_pred = torque_pred_scaled / torque_scale # Reverse scaling
```

```
[ 0.00000000e+00  1.41501113e-08  1.13951685e-07 -8.28734414e-10
 -8.97167183e-07]
```

```python
# ==============================================
# Evaluation
# ==============================================

# Generate theta_test values
theta_test = jnp.linspace(0, 2 * jnp.pi, 10).reshape(-1, 1)
f_test = jnp.linspace(0.5, 3.0, 10).reshape(-1, 1)

# Compute true torque with a specific frequency for comparison
torque_true = jax.vmap((compute_torques))(theta_test.squeeze(), f_test.squeeze())

print(torque_true)

# Predict using the trained model, including f_test
torque_pred_log = model.apply(state.params, theta_test, f_test)
torque_pred = 10**torque_pred_log
```

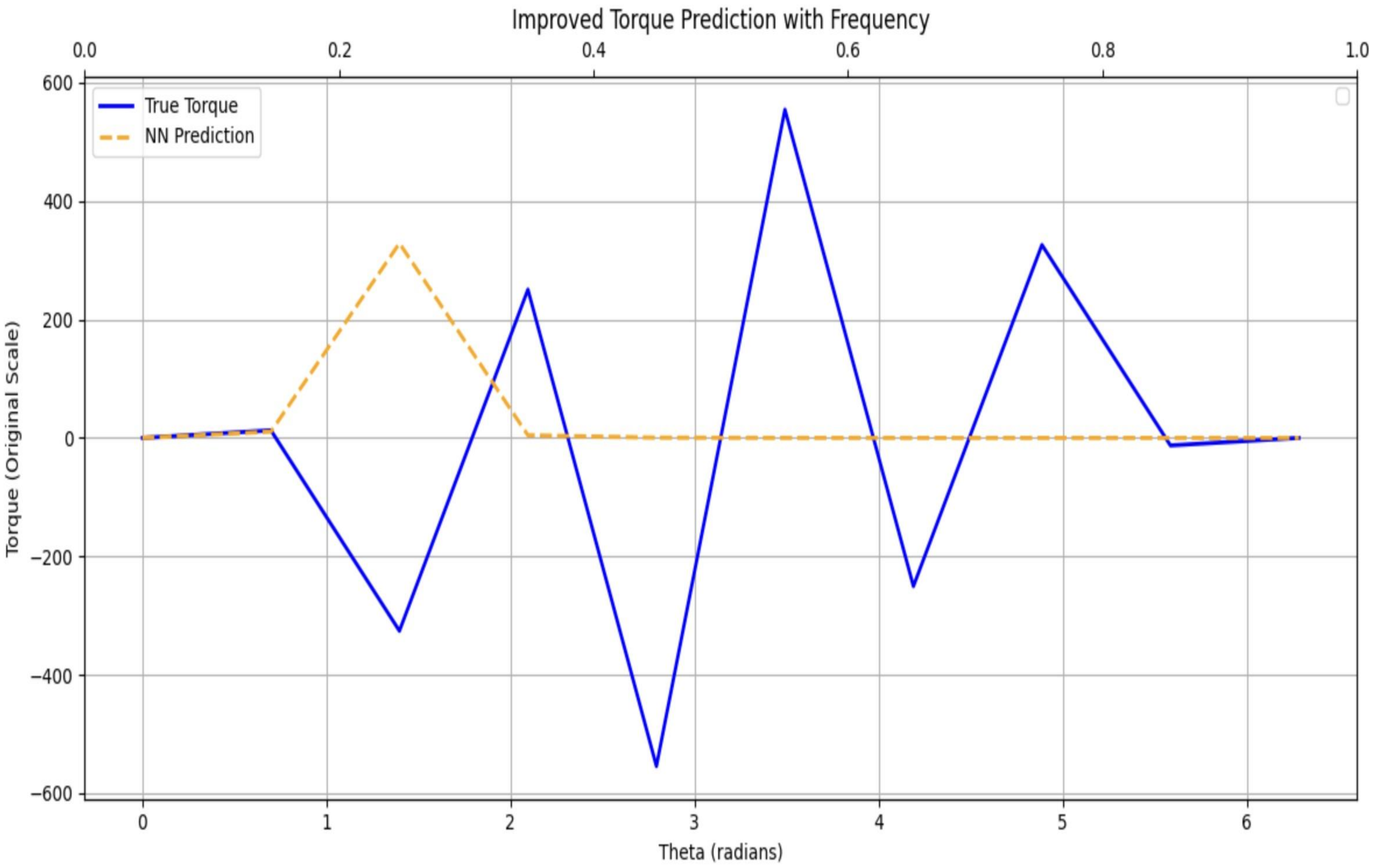
```
[ 1.79366203e-43  1.27214798e+01 -3.25827637e+02  2.50482364e+02
 -5.54214168e+02  5.54214168e+02 -2.50482364e+02  3.25827637e+02
 -1.27214798e+01 -1.69010573e-24]
```

```python
# ==============================================
# Evaluation
# ==============================================

# Generate theta_test values
theta_test = jnp.linspace(0, 2 * jnp.pi, 7).reshape(-1, 1)
f_test = jnp.linspace(0.5, 8.5, 7).reshape(-1, 1)

# Compute true torque with a specific frequency for comparison
torque_true = jax.vmap((compute_torques))(theta_test.squeeze(), f_test.squeeze())

print(torque_true)

# Predict using the trained model, including f_test
torque_pred_log = model.apply(state.params, theta_test, f_test)
torque_pred = 10**torque_pred_log
```

```
[ 1.79366203e-43 -2.50482364e+02  2.50482364e+02 -2.50987756e-26
 -2.50482364e+02  2.50482364e+02 -6.83861649e-25]
```

Levitating Rotor – 2D NN

```
Epoch 0, Loss: 0.7552
Epoch 100, Loss: 0.0042
Epoch 200, Loss: 0.0041
Epoch 300, Loss: 0.0040
Epoch 400, Loss: 0.0040
Epoch 500, Loss: 0.0039
Epoch 600, Loss: 0.0039
Epoch 700, Loss: 0.0039
Epoch 800, Loss: 0.0039
Epoch 900, Loss: 0.0039
Epoch 1000, Loss: 0.0039
Epoch 1100, Loss: 0.0039
Epoch 1200, Loss: 0.0039
Epoch 1300, Loss: 0.0038
Epoch 1400, Loss: 0.0038
Epoch 1500, Loss: 0.0038
Epoch 1600, Loss: 0.0038
Epoch 1700, Loss: 0.0038
Epoch 1800, Loss: 0.0038
Epoch 1900, Loss: 0.0038
Epoch 2000, Loss: 0.0038
Epoch 2100, Loss: 0.0038
Epoch 2200, Loss: 0.0038
Epoch 2300, Loss: 0.0037
Epoch 2400, Loss: 0.0037
Epoch 2500, Loss: 0.0037
Epoch 2600, Loss: 0.0037
Epoch 2700, Loss: 0.0037
Epoch 2800, Loss: 0.0037
Epoch 2900, Loss: 0.0036
```




```
Test MSE: 1.061e+05
Max Error: 8.828e+02
```

Levitating Rotor – Updated System

- Controlled dependence between electrode driving frequencies and rotor's frequency. Use of laser detector.
- Accommodate for more oscillatory patterns of driving frequencies (not limited to triangular waveforms).
- Collaborators found varying modes of vibration of rotor arms. X, Y modes etc. Intensity of modes affect stability of system.
- Require NN to identify missing physics causing coupling of modes (e.g. are there energy transfers between modes) to find ways to stabilize vibrations.