Electrodes-Driven Rotor System

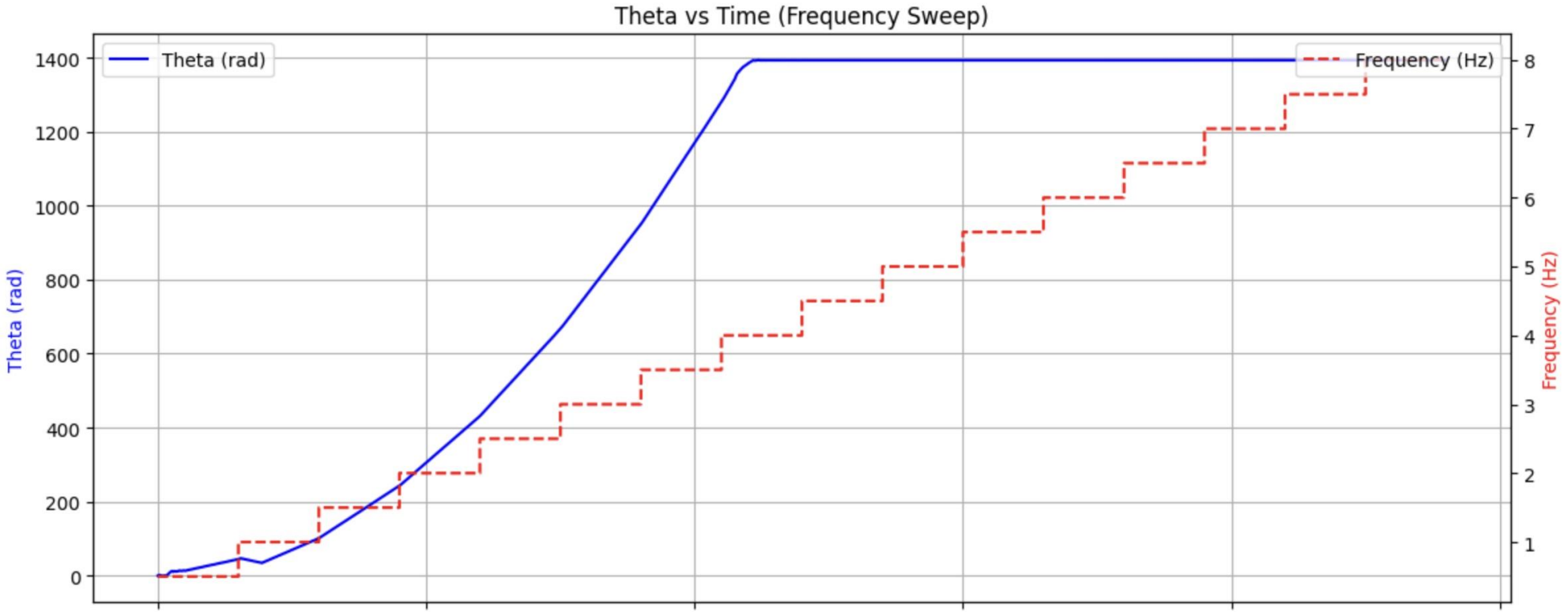
Objectives in the past week

- ✓ Experiment with simulation parameters (driving frequency, sampling points) to explore the physical system.
- ✓ Build a simple neural network that predicts the torque of the entire system given fixed time, frequency with varying theta.
- ✓ Ideas for next step in implementing the APHYNITY Model

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

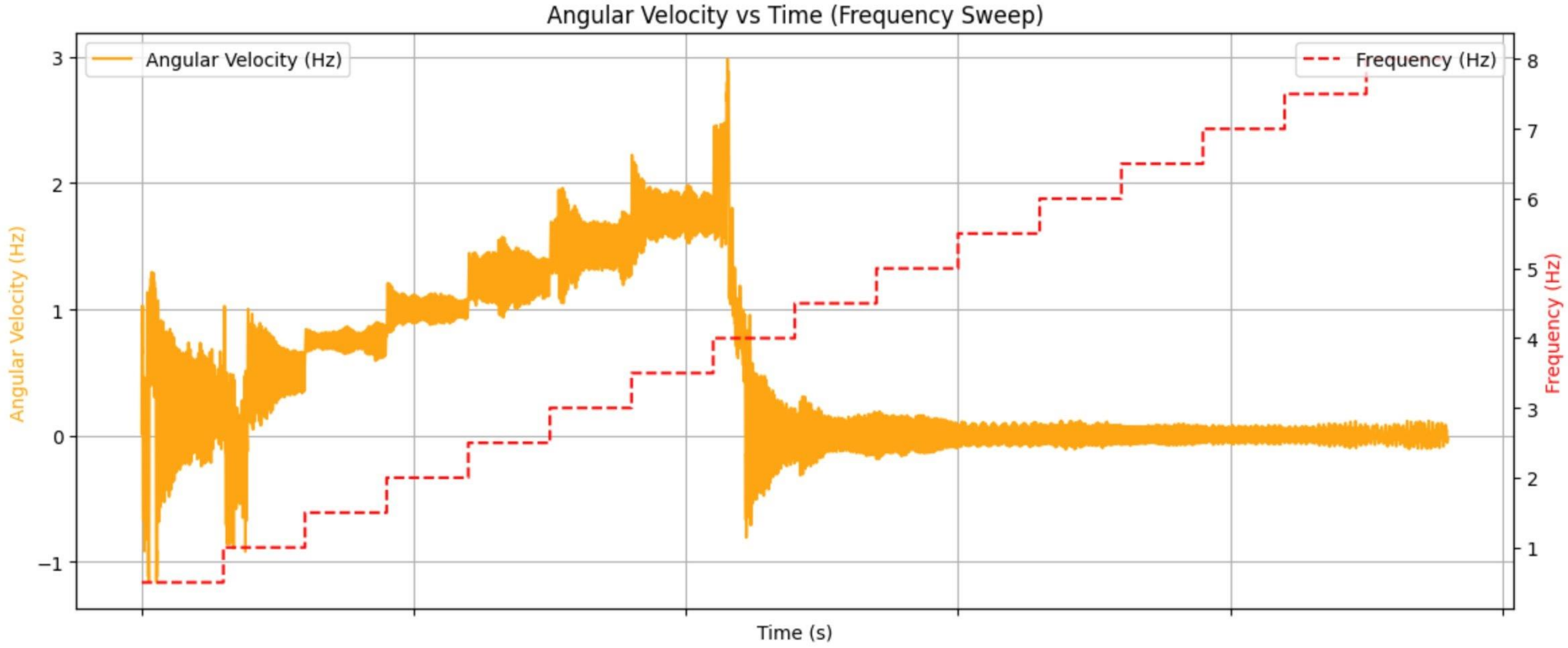
Experimenting with Parameters of Rotor System

```
# Simulation parameters
t_span = (0, 30)  # Time range(s)
t_eval = jnp.linspace(t_span[0],t_span[1], 1000) #numerical solver determines how quickly data is sampled
frequencies = jnp.arange(0.5, 8.5, 0.5)  # Frequencies from 0.5 Hz to 8 Hz
```



Experimenting with Parameters of Rotor System

```
# Simulation parameters
t_span = (0, 30)  # Time range(s)
t_eval = jnp.linspace(t_span[0],t_span[1], 1000) #numerical solver determines how quickly data is sampled
frequencies = jnp.arange(0.5, 8.5, 0.5)  # Frequencies from 0.5 Hz to 8 Hz
```



Experimenting with Parameters of Rotor System

2500 sampled points in 1 sec too expensive. 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.

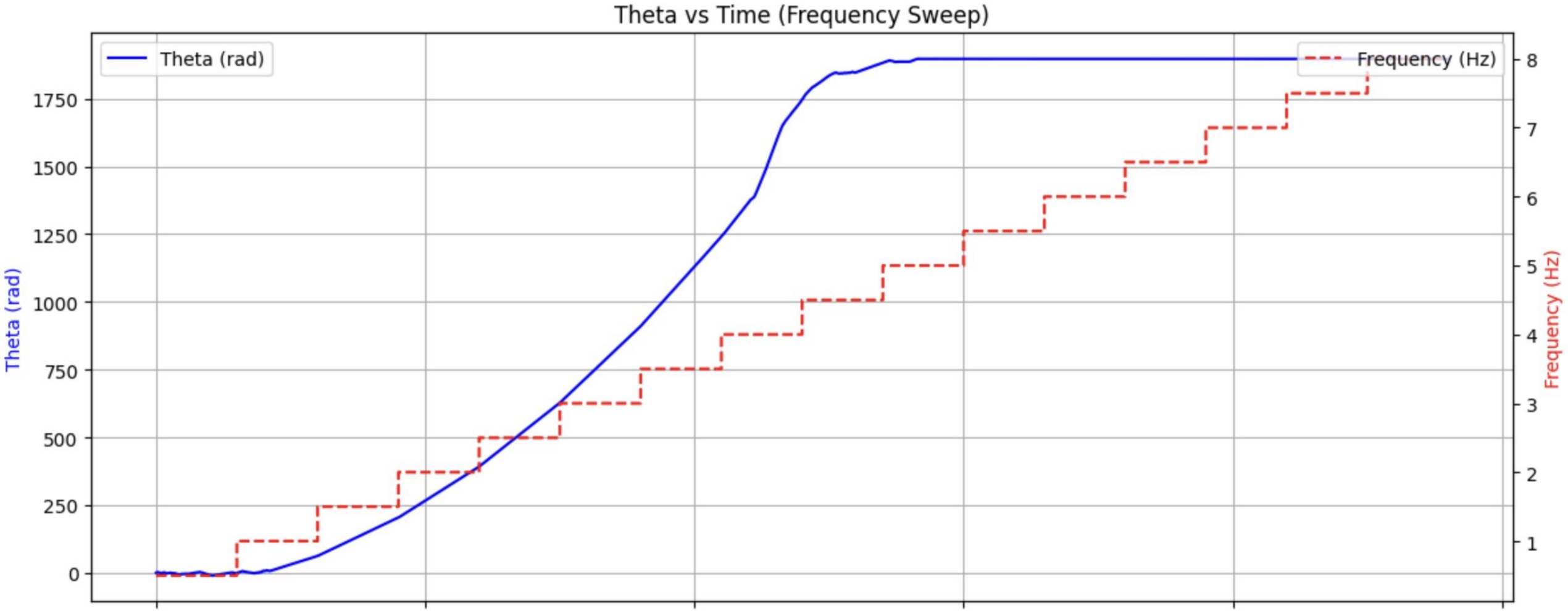
Modification 8: 0.2Hz step-up in frequencies AND 500 sampling points over 5s. Result -- Max angular velocity > 4.0Hz. Max theta above 1200 rad.

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.

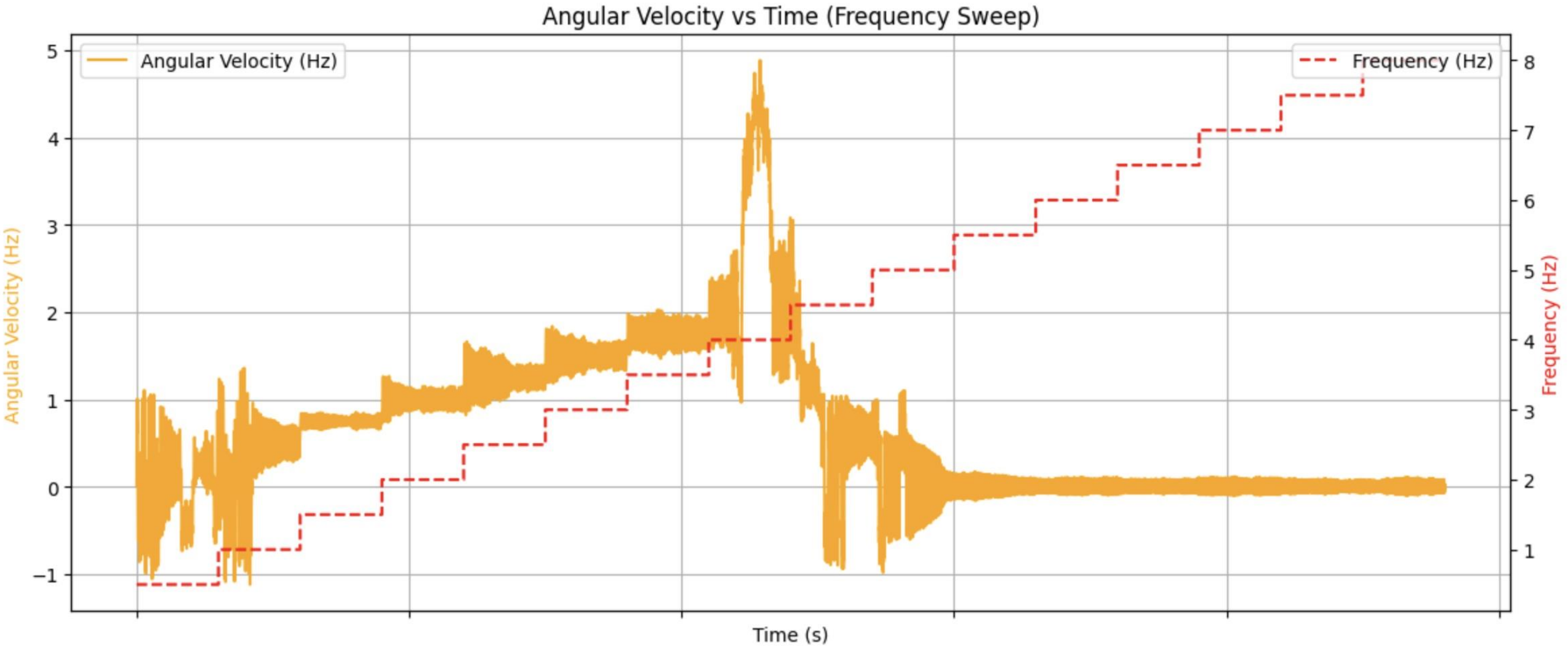
Experimenting with Parameters of Rotor System

Best Result: time_span = 30s, driving frequency step = 0.5Hz, number of sampling points = 73.33 per second



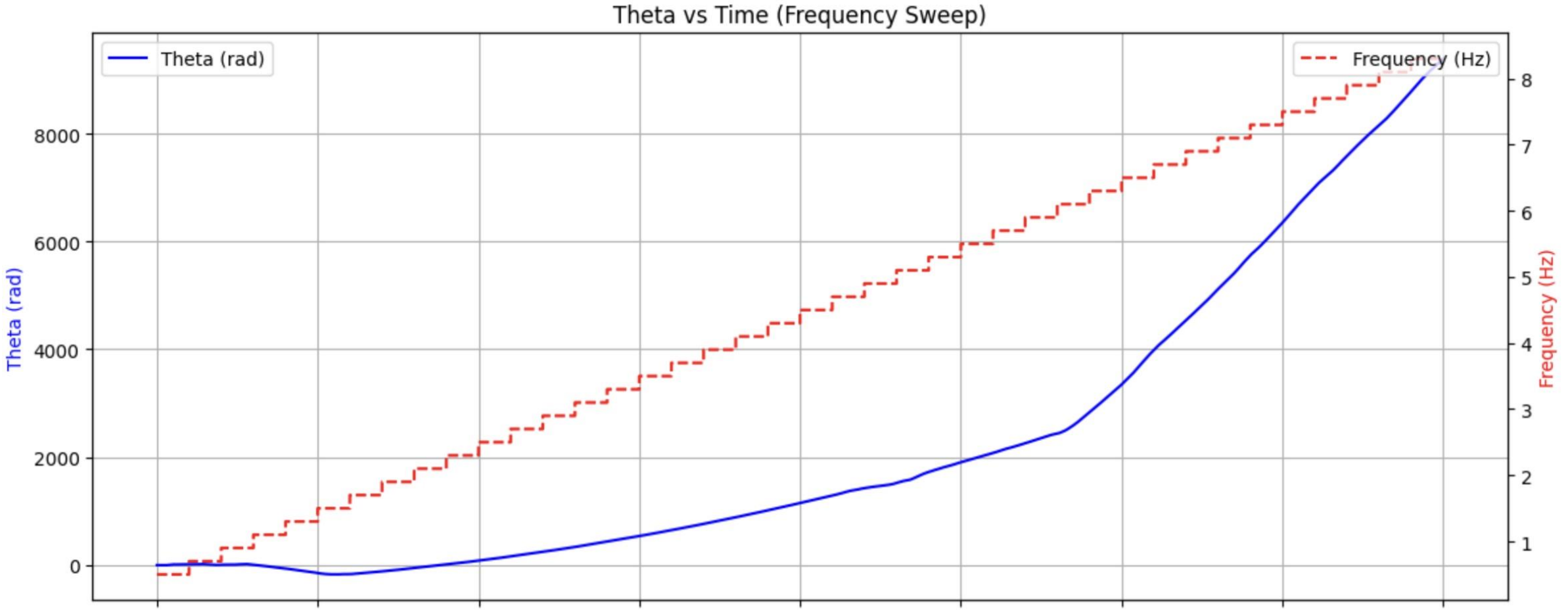
Experimenting with Parameters of Rotor System

Best Result: time_span = 30s, driving frequency step = 0.5Hz, number of sampling points = 73.33 per second



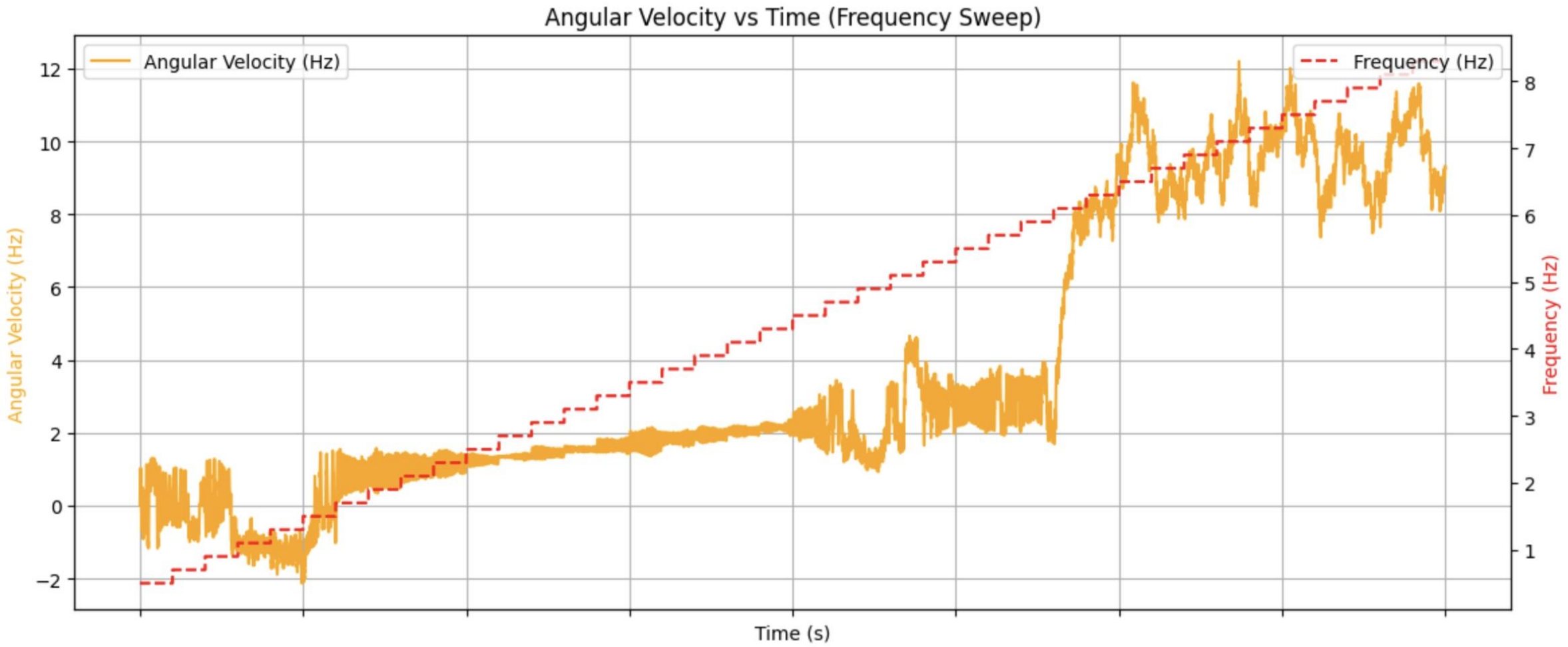
Experimenting with Parameters of Rotor System

Best Result: time_span = 10s, driving frequency step = 0.2Hz, number of sampling points = 50 per second



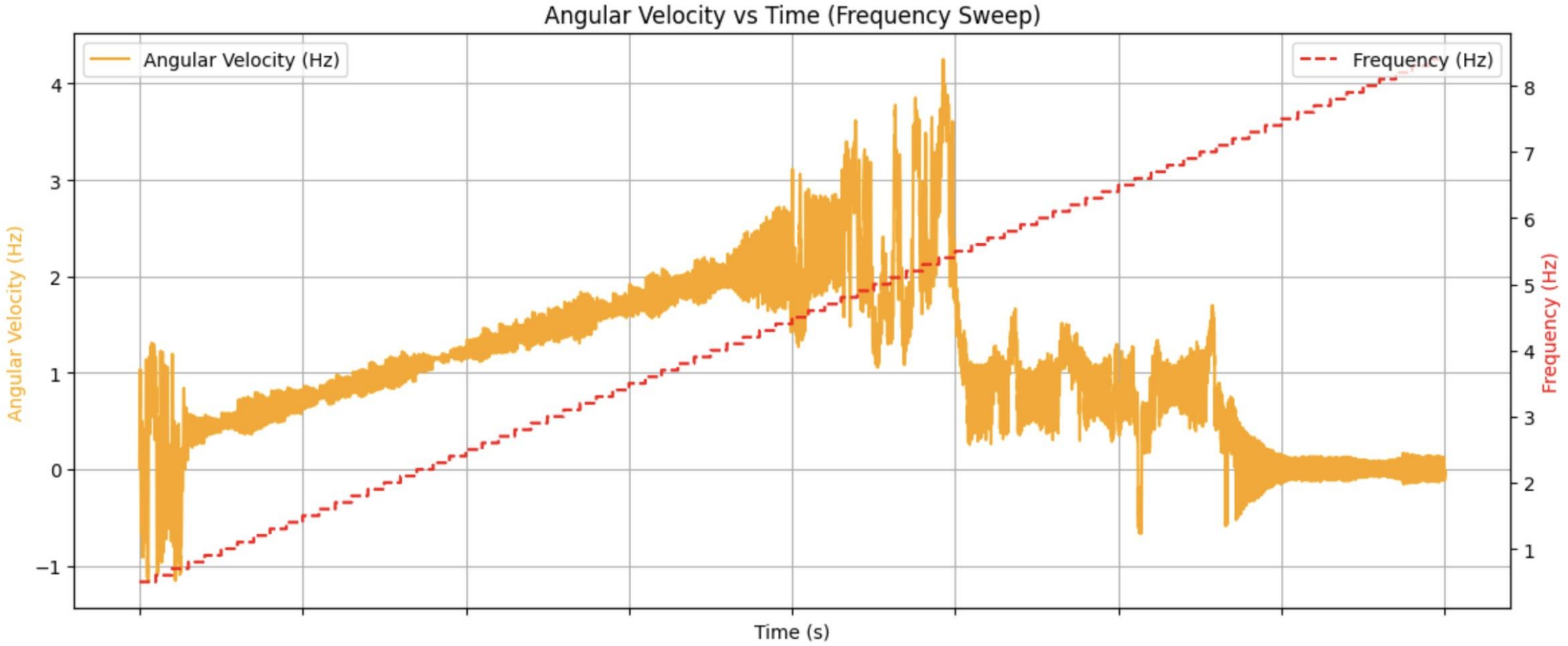
Experimenting with Parameters of Rotor System

Best Result: time_span = 10s, driving frequency step = 0.2Hz, number of sampling points = 50 per second



Experimenting with Parameters of Rotor System

time_span = 5s, driving frequency step = 0.1Hz, number of sampling points = 100 per second



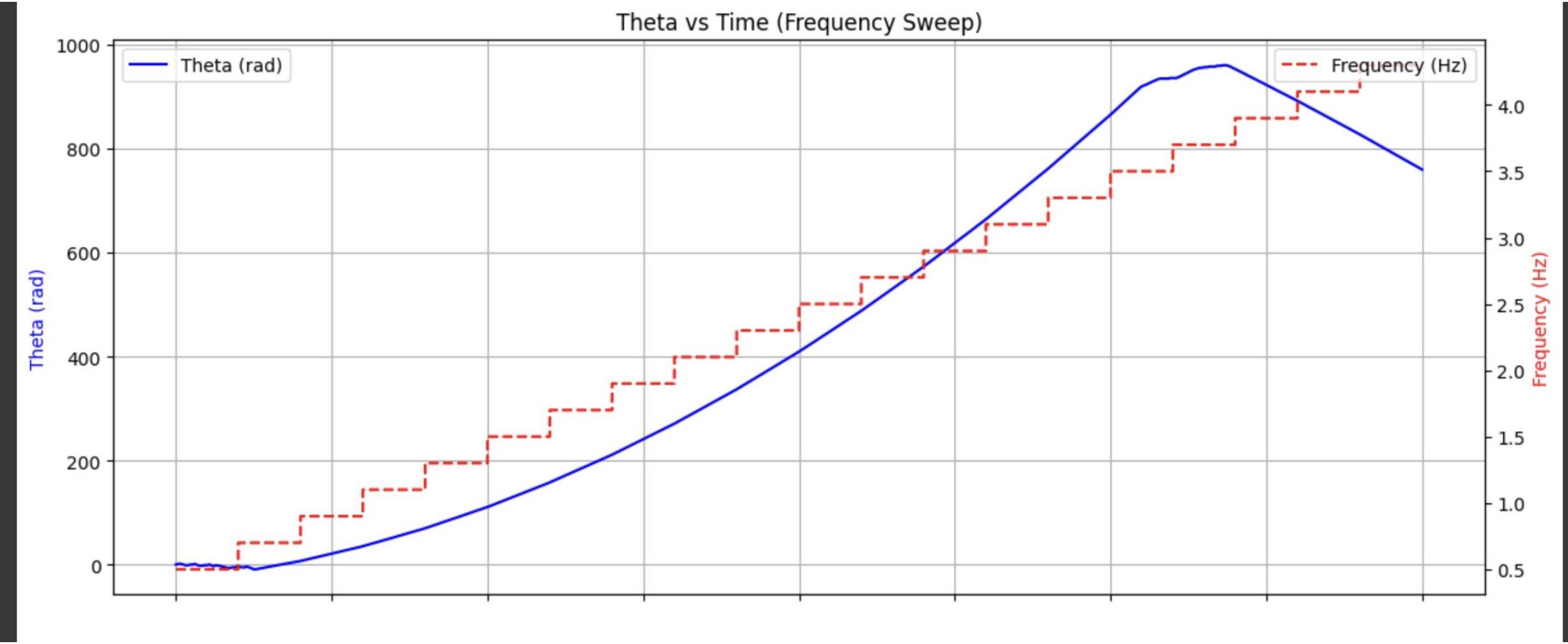
Experimenting with Parameters of Rotor System

2nd Round of Tests

Modification 1: 0.5Hz step-up in frequencies AND 2200 sampling points across 30s. (runtime 5 mins)
Result -- Max angular velocity near 5Hz. Max theta at 1750 rad. Max theta > 1750 rad.
Modification 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.
Modification 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.

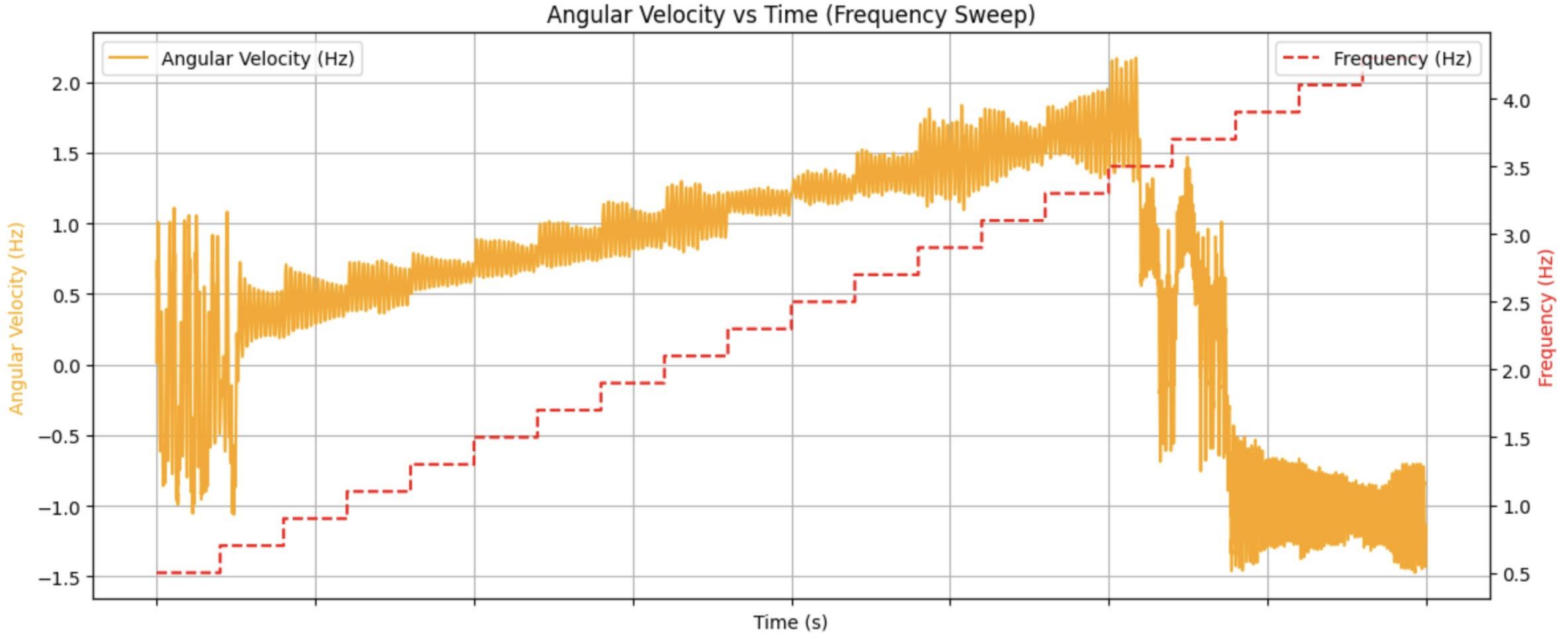
Experimenting with Parameters of Rotor System

time_span = 10s, driving frequency step = 0.2Hz, number of sampling points = 100 per second



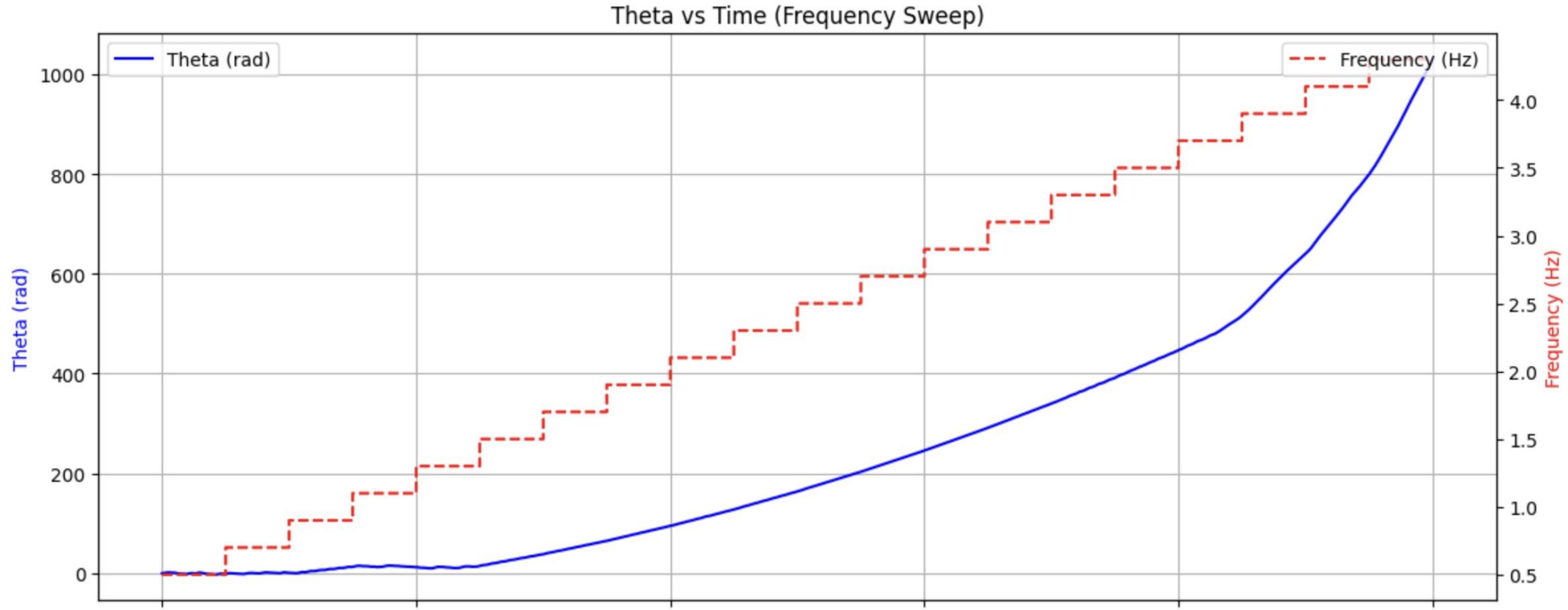
Experimenting with Parameters of Rotor System

time_span = 10s, driving frequency step = 0.2Hz, number of sampling points = 100 per second



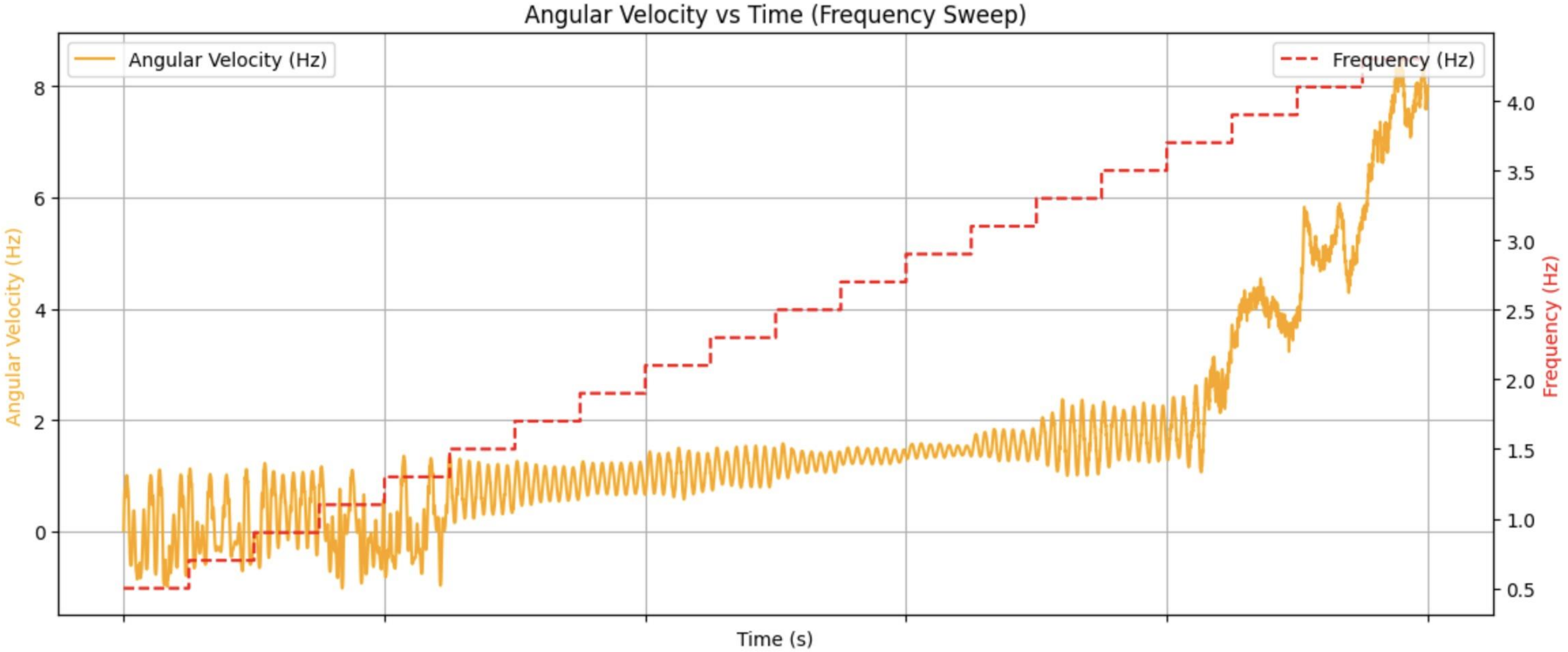
Experimenting with Parameters of Rotor System

time_span = 5s, driving frequency step = 0.2Hz, number of sampling points = 200 per second



Experimenting with Parameters of Rotor System

time_span = 5s, driving frequency step = 0.2Hz, number of sampling points = 200 per second



Experimenting with Parameters of Rotor System

Interpretations

- Increasing the number of sampling points per second does not change anything.
- Minimum step-up frequency for reasonable runtime is around 0.1Hz for a time span of 10 to 30s.
- Reducing number of sampling points (in this case halved) for a time span avoids expensive computation.
- Negative effect in reduction of sampling points for a time span mitigated by decrease in step-up frequency.
- Increase in sampling points together with decrease in frequency step sizes gives the best results (next slide). But too many sampling points does not give the best result.
- Too small step size also does not give best result (0.2Hz better than 0.1Hz for time_span = 10s).

Conclusion

**Expensiveness of computation decreases with decreasing sampling points, decreasing time span and increasing step size.

**Greater delay in decay of rotor frequency caused by sufficiently high number of sampling points per second and sufficiently small driving frequency step size (sufficient information and less effects of damping caused by abruptness of driving frequency changes).

Building a Simple NN

Random initialization of weights as rng

```
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

# Initialize random key
rng = jax.random.PRNGKey(0)
```

```
# Generate training data
theta_train = jnp.linspace(0, 2*jnp.pi, 1000)
torque_train = jax.vmap(partial(compute_torques_scalar, t=20.0, f=3.0))(theta_train)

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

Training Data with choice of t and f

Building a Simple NN

```
# Define model
class TorquePredictor(nn.Module):
    @nn.compact
    def __call__(self, x):
        x = jnp.concatenate([x, jnp.sin(x), jnp.cos(x)], axis=-1)
        x = nn.Dense(64)(x)
        x = nn.relu(x)
        x = nn.Dense(32)(x)
        x = nn.relu(x)
        return nn.Dense(1)(x)

# Initialize model
model = TorquePredictor()
params = model.init(rng, jnp.ones((1, 1)))
```

Note: Periodicity of torque function

Building a Simple NN

Training of neural network

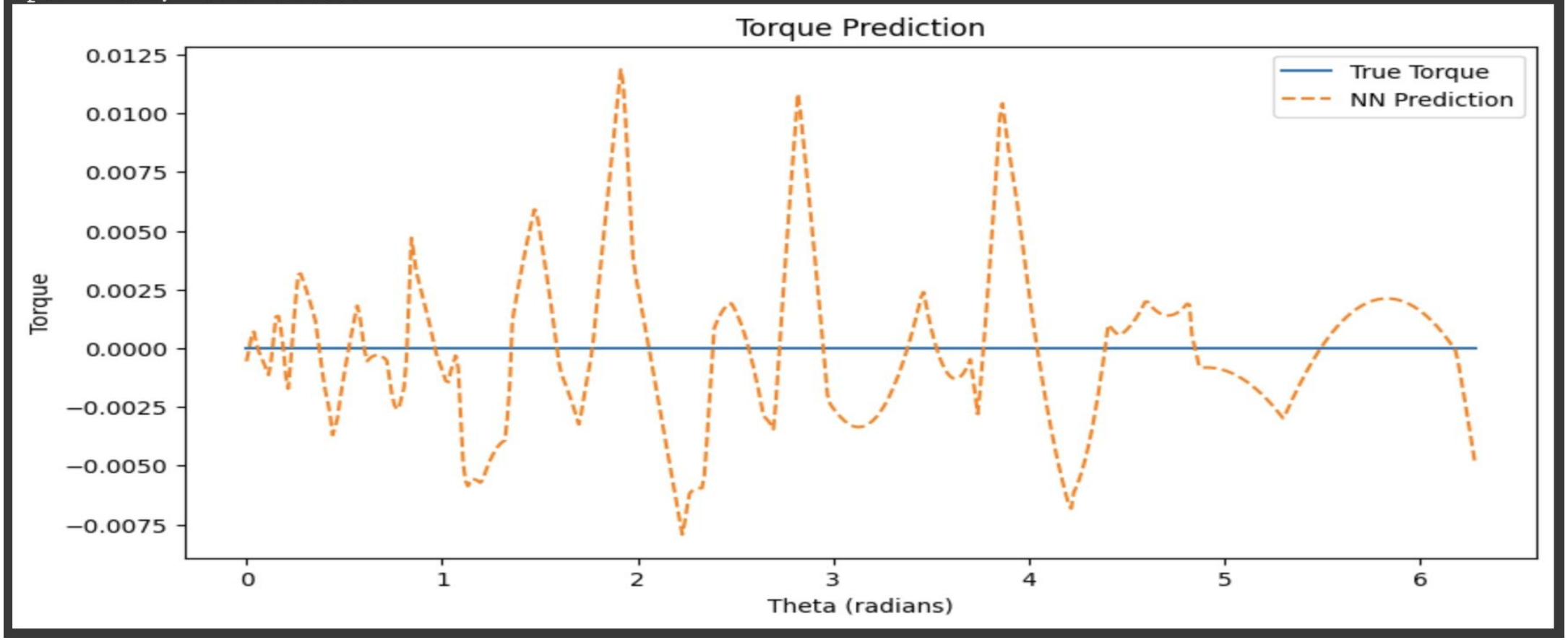
Graphical comparison

```python
# Create train state
optimizer = optax.adam(1e-3)
state = train_state.TrainState.create(
    apply_fn=model.apply,
    params=params,
    tx=optimizer
)

# Training loop
@jax.jit
def train_step(state, batch):
    theta_batch, torque_batch = batch
    def loss_fn(params):
        pred = state.apply_fn(params, theta_batch)
        return jnp.mean((pred - torque_batch)**2)
    loss, grads = jax.value_and_grad(loss_fn)(state.params)
    return state.apply_gradients(grads=grads), loss

for epoch in range(1001):
    state, loss = train_step(state, (theta_train, torque_train))
    if epoch % 100 == 0:
        print(f"Epoch {epoch}, Loss: {loss:.4f}")

theta_test = jnp.linspace(0, 2*jnp.pi, 500).reshape(-1, 1)
torque_true = jax.vmap(partial(compute_torques_scalar, t=20.0, f=3.0))(theta_test.squeeze())
torque_pred = model.apply(state.params, theta_test)
```

```python
# Plot results
import matplotlib.pyplot as plt
plt.figure(figsize=(10, 5))
plt.plot(theta_test, torque_true, label='True Torque')
plt.plot(theta_test, torque_pred, '--', label='NN Prediction')
plt.xlabel('Theta (radians)')
plt.ylabel('Torque')
plt.legend()
plt.title('Torque Prediction')
plt.show()
```

Building a Simple NN

```
Epoch 0, Loss: 8.2295
Epoch 100, Loss: 0.0009
Epoch 200, Loss: 0.0003
Epoch 300, Loss: 0.0001
Epoch 400, Loss: 0.0001
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
```



Building a Simple NN

Min-Max Normalization/Scaling

NORMALIZATION

Normalization Vs. Standardization (Feature Scaling in Machine Learning)

- Normalization is conducted to make feature values range from 0 to 1.

$$x' = \frac{x - \min(x)}{\max(x) - min(x)}$$

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
stock_df = scaler.fit_transform(stock_df)
```

Building a Simple NN

Min-Max Normalization/Scaling

```
# Symmetric Min-Max Scaling (to [-1,1] range)
def symmetric_min_max_scale(x, x_min=None, x_max=None):
    "Scale data to [-1, 1] range preserving zero."
    if x_min is None:
        x_min = jnp.min(x)
    if x_max is None:
        x_max = jnp.max(x)
    # Scale symmetrically around zero
    scale = jnp.maximum(jnp.abs(x_min), jnp.abs(x_max))
    return x / scale, scale  #preserves negative and positive values

# Scale training torque (preserving sign)
torque_train_scaled, torque_scale = symmetric_min_max_scale(torque_train)

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

Building a Simple NN

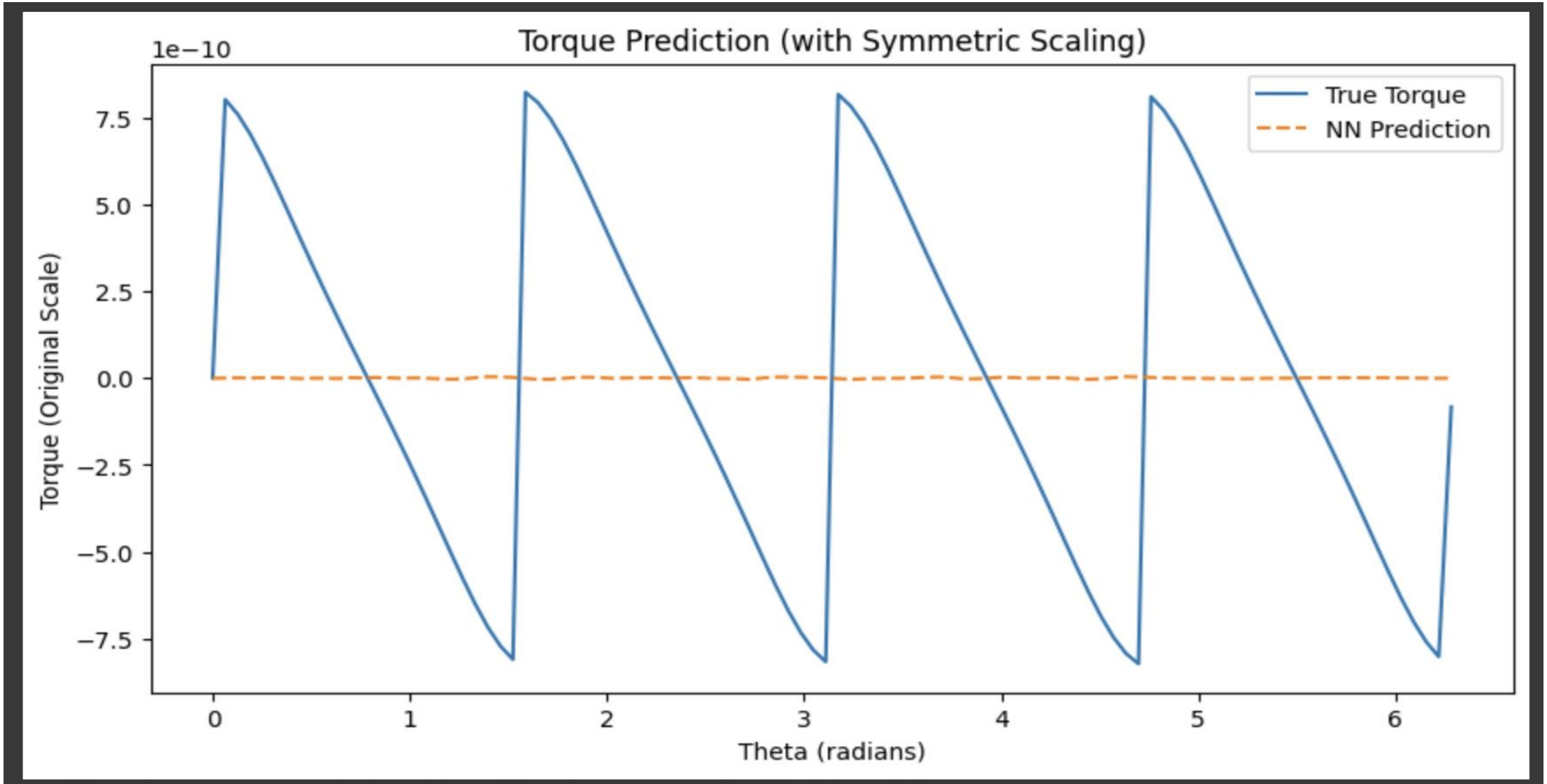
Min-Max Normalization/Scaling

```python
# Inverse Scaling for Predictions
def inverse_symmetric_min_max_scale(x_scaled, scale):
    "Convert scaled data back to original range."
    return x_scaled * scale
```

```python
# Generate test data
theta_test = jnp.linspace(0, 2*jnp.pi, 100).reshape(-1, 1)
torque_true = jax.vmap(partial(compute_torques_scalar, t=20.0, f=3.0))(theta_test.squeeze())

# Predict and inverse-scale
torque_pred_scaled = model.apply(state.params, theta_test)
torque_pred = inverse_symmetric_min_max_scale(torque_pred_scaled, torque_scale)
```

Building a Simple NN

Min-Max Normalization/Scaling



Building a Simple NN

Re-evaluation of Scaling

```python
def scale_torque(torque):
    """Scale torque to [-1, 1] range before normalization"""
    scale_factor = 1e10  # Adjust based on your torque magnitude
    scaled = torque * scale_factor
    return scaled, scale_factor

# Generate and scale training data
theta_train = jnp.linspace(0, 2*jnp.pi, 1000)
torque_train = jax.vmap(partial(compute_torques_scalar, t=20.0, f=3.0))(theta_train)
torque_train_scaled, torque_scale = scale_torque(torque_train)  # Now in ~[-1,1] range
# Reshape data
theta_train = theta_train.reshape(-1, 1)
torque_train_scaled = torque_train_scaled.reshape(-1, 1)
```

Building a Simple NN

Re-evaluation of Scaling

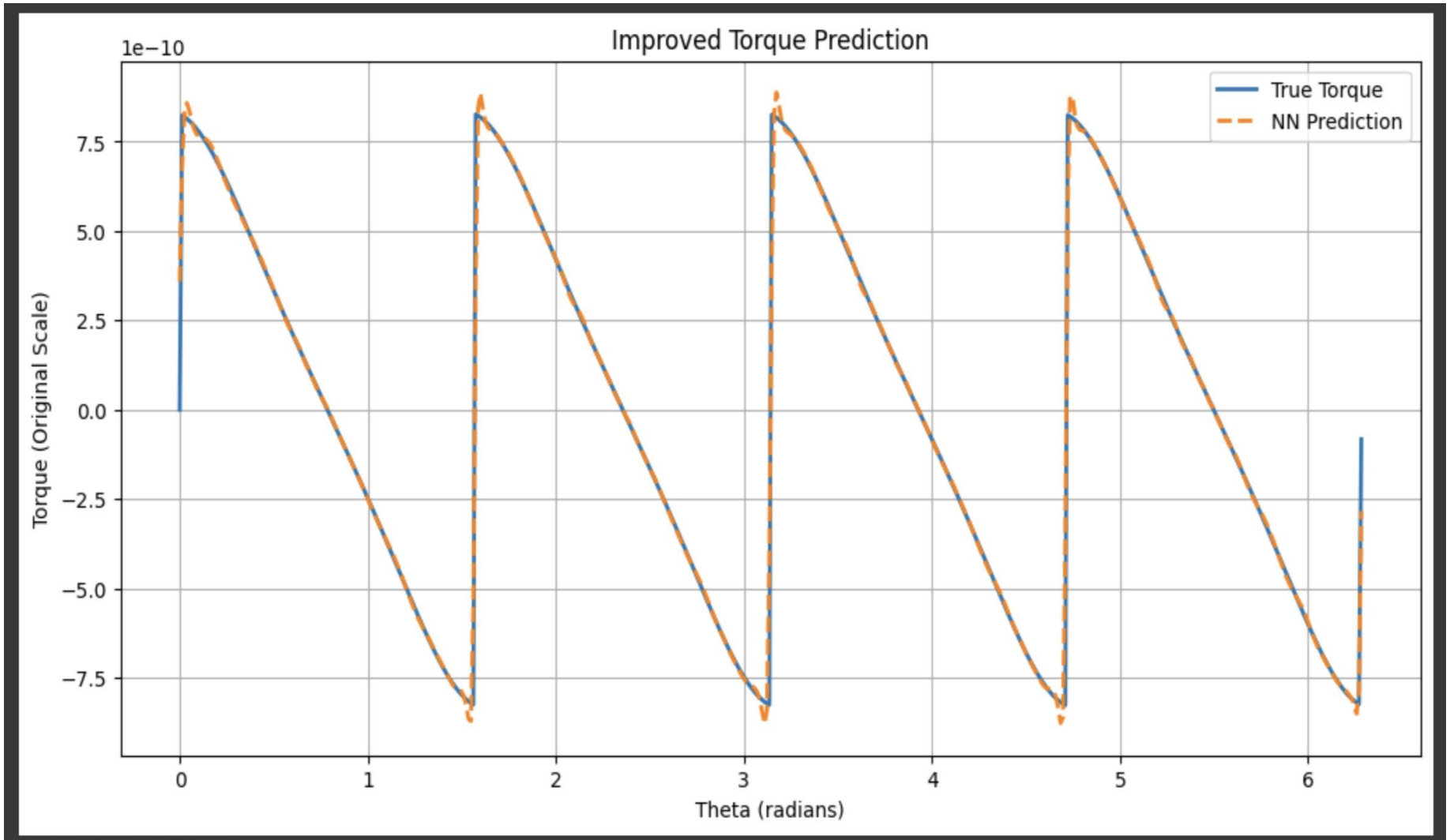
Swish is continuous and differentiable. Better gradient flow.

```
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
```

```
Epoch 0, Loss: 27.0896
Epoch 100, Loss: 12.5017
Epoch 200, Loss: 6.4408
Epoch 300, Loss: 4.1671
Epoch 400, Loss: 3.4420
Epoch 500, Loss: 2.9947
Epoch 600, Loss: 2.5821
Epoch 700, Loss: 2.2228
Epoch 800, Loss: 1.9127
Epoch 900, Loss: 1.6639
Epoch 1000, Loss: 1.4653
Epoch 1100, Loss: 1.2991
Epoch 1200, Loss: 1.1581
Epoch 1300, Loss: 1.0382
Epoch 1400, Loss: 0.9364
Epoch 1500, Loss: 0.8496
Epoch 1600, Loss: 0.7747
Epoch 1700, Loss: 0.7096
Epoch 1800, Loss: 0.6527
Epoch 1900, Loss: 0.6024
Epoch 2000, Loss: 0.5576
Epoch 2100, Loss: 0.5170
Epoch 2200, Loss: 0.4800
Epoch 2300, Loss: 0.4451
Epoch 2400, Loss: 0.4139
Epoch 2500, Loss: 0.3857
Epoch 2600, Loss: 0.3599
Epoch 2700, Loss: 0.3363
Epoch 2800, Loss: 0.3148
Epoch 2900, Loss: 0.2952
```

Building a Simple NN

Accurate Prediction of Torques!




```
Test MSE: 5.368e-19
Max Error: 1.714e-09
```

Next Step...

- Use of actual experimental data by collaborators to find the data-driven component function in the APHYNITY Model
- Begin with only predicting the torque first (same as what has been done here)?