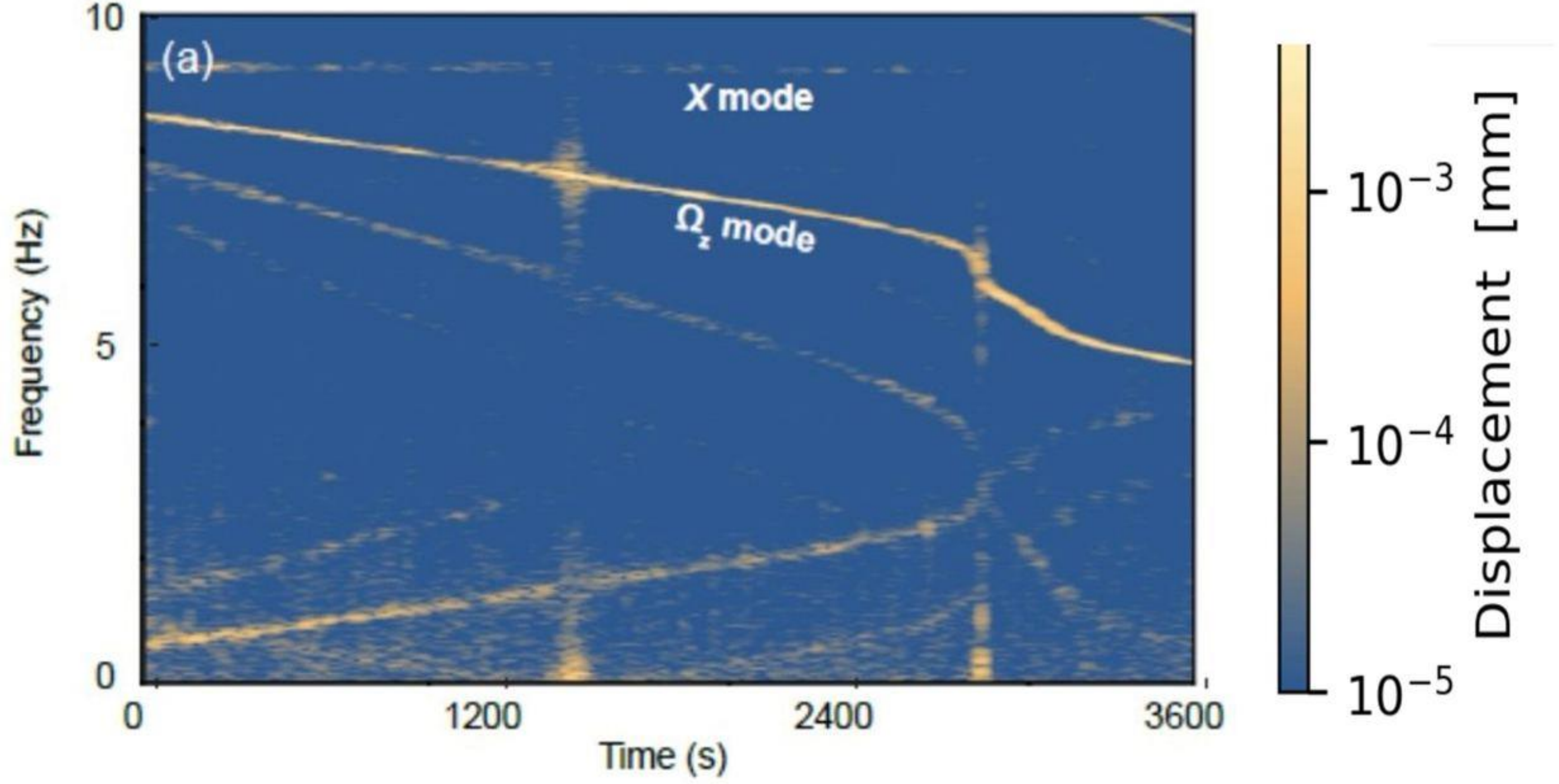
PINN for Levitating Rotor

Final Updates:

- **Downsizing of Experimental Data**
- **PINN Progress for driving_force = F(x, theta)**

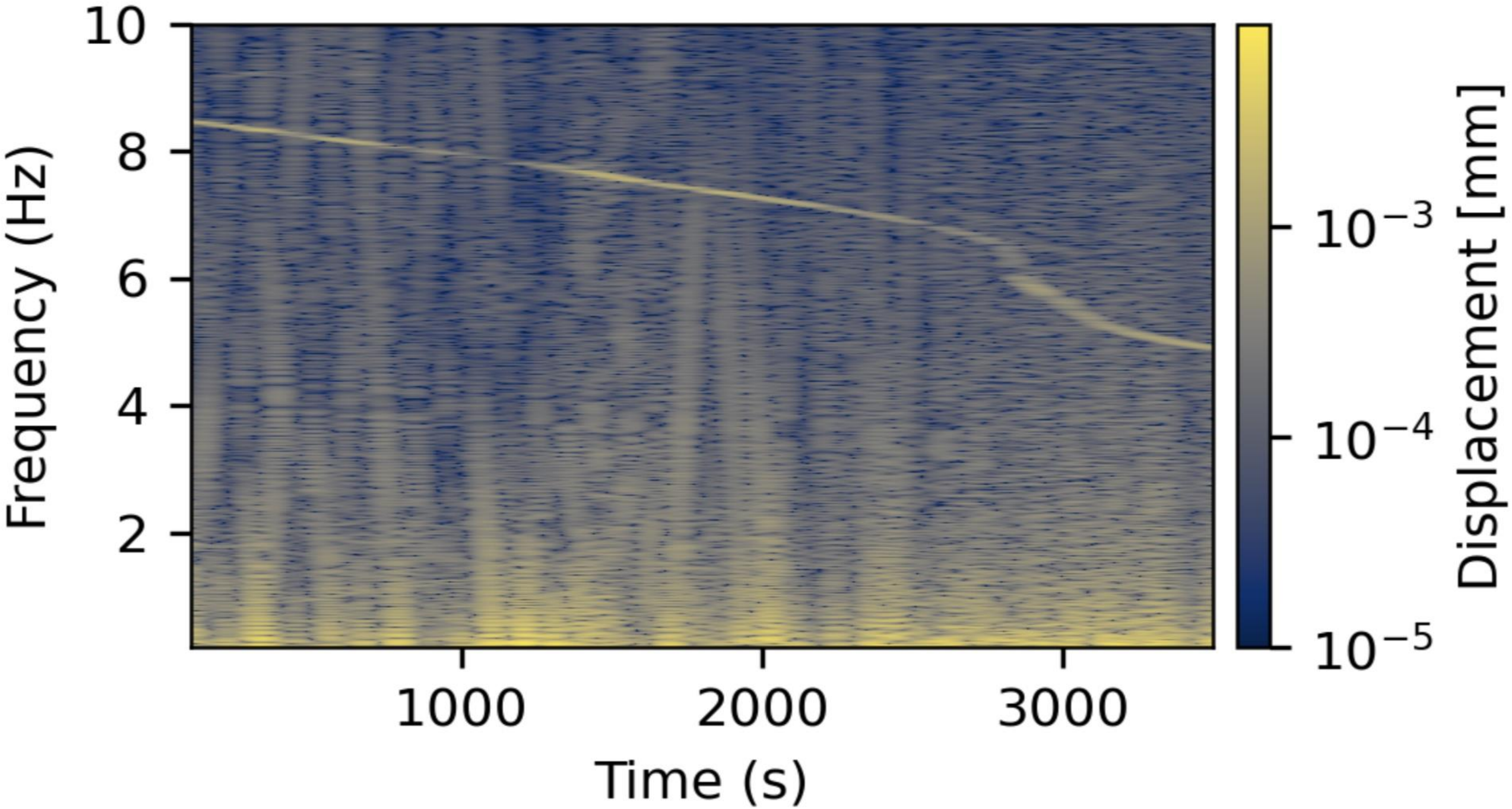
Downsizing Experimental Data

FFT window_size = 200. Original data points. Both theta mode and x mode are visible.



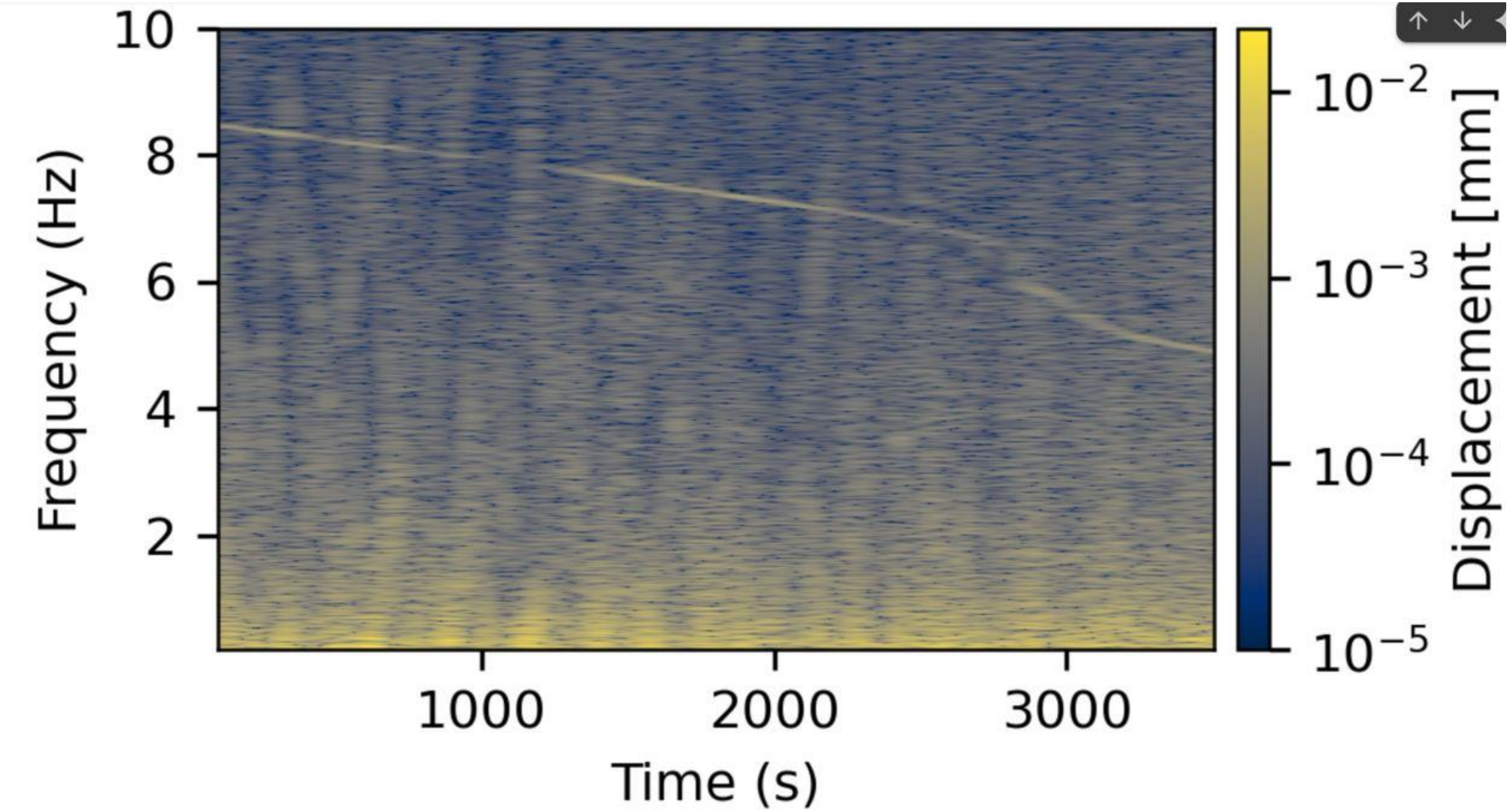
Downsizing Experimental Data

FFT window_size = 200. Downsize to 1 in 2 data points. Theta mode is visible and trace of X mode can be observed.



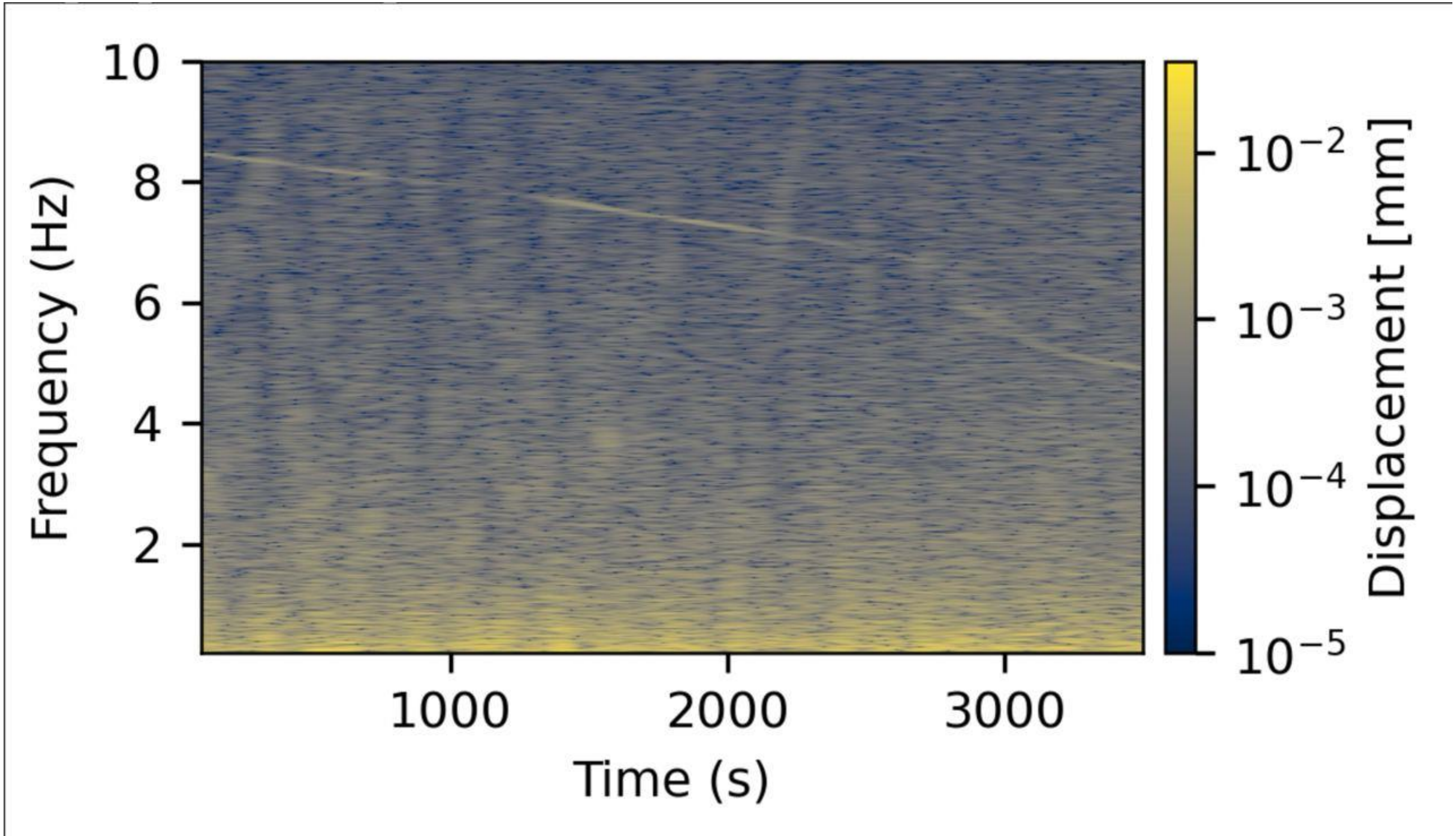
Downsizing Experimental Data

FFT window_size = 200. Downsize to 1 in 5 data points. Only theta mode is visible.



Downsizing Experimental Data

FFT window_size = 200. Downsize to 1 in 10 data points. Only theta mode is visible.

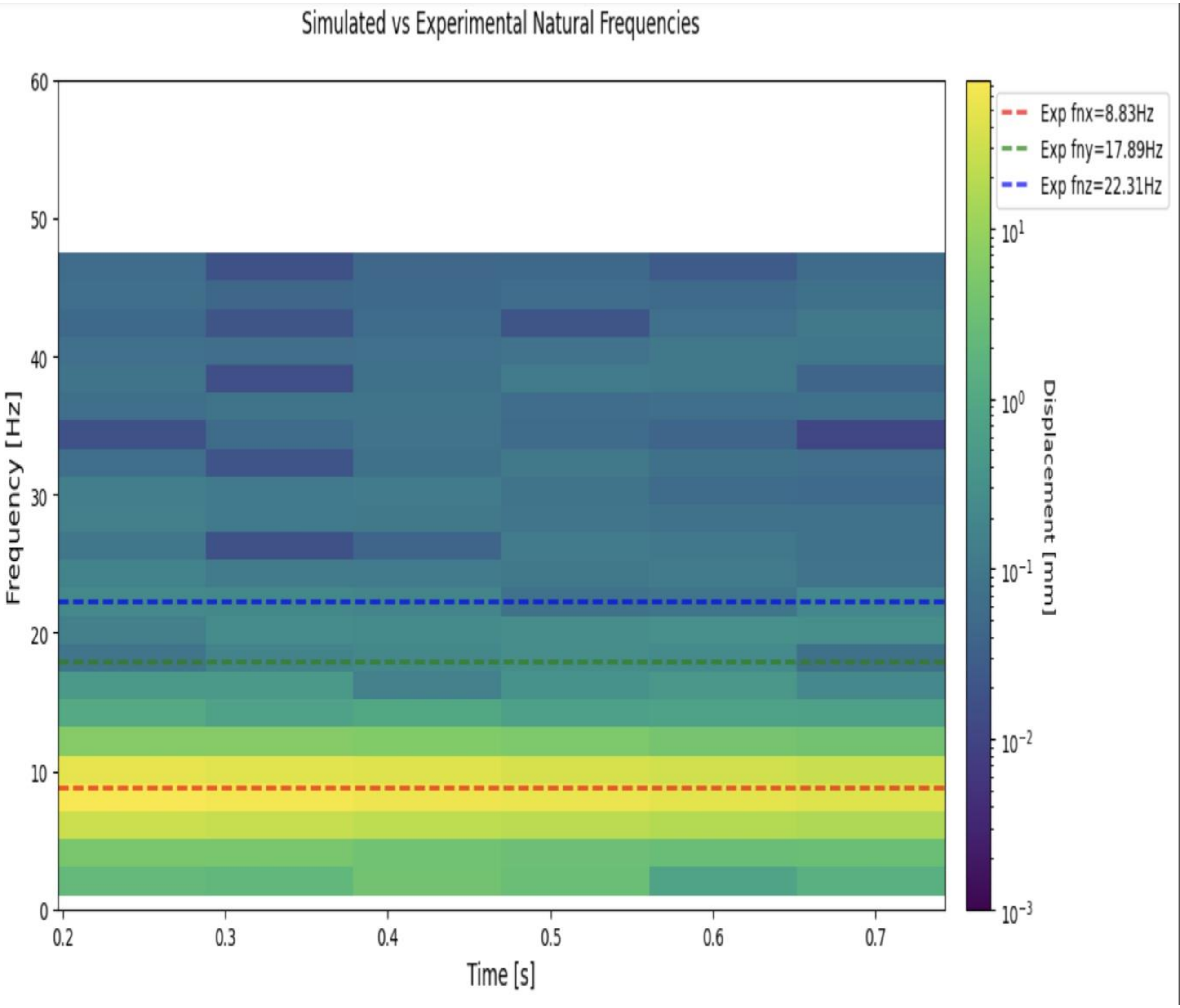


Defined FFT Function

```
def perform_fft_final(time_data, signal_data, window_size=0.5, overlap=0.8):
    # CRITICAL FIX: Proper sampling rate calculation
    fs = 1/(time_data[1] - time_data[0])  # Must use actual time differences
    print(f"Calculated sampling rate: {fs} Hz")  # Should be ~100Hz for your data

    win_N = int(window_size * fs)
    hop_N = int(win_N * (1 - overlap))

    # Window function with energy correction
    win = 0.5 * (1 - jnp.cos(2 * jnp.pi * jnp.arange(win_N) / (win_N - 1)))
    win_gain = jnp.mean(win)  # For amplitude correction

    # FFT processing
    nfft = win_N
    freq_bins = fftfreq(nfft, d=1/fs)[:nfft//2]  # Now with correct fs

    num_windows = (len(signal_data) - win_N) // hop_N + 1
    vel_spec = jnp.zeros((nfft//2, num_windows))
    t_frames = jnp.zeros(num_windows)
```

FFT Spectrum for Simulated Data

```python
# Define the system of ODEs
def equation_of_motion(y, t):
    theta, theta_dot, x, x_dot = y
    driving_force = jnp.sin(x) * jnp.sin(theta)
    total_tau = 0 - alpha*driving_force
    theta_ddot = (total_tau - mu * theta_dot) / I
    x_ddot = (driving_force - k * x - c * x_dot) / m
    return jnp.array([theta_dot, theta_ddot, x_dot, x_ddot])

# Generate full training data (over [0,1] seconds)
t_eval = jnp.linspace(0, 1, 100)  # Adjust this to [0,5] if needed #fs for ground truth
y0_train = jnp.array([0.1, 0.1, 0.1, 0.1])
solution_train = odeint(equation_of_motion, y0_train, t_eval, rtol=1e-3, atol=1e-3)
x_dot_train = solution_train[:, 3]
x_train = solution_train[:, 2]
theta_train = solution_train[:, 0]
theta_dot_train = solution_train[:, 1]
```



PINN

- Set driving_force = jnp.tanh(x)*jnp.tanh(theta) for simpler prediction.
- Applied Min-Max normalization for variables to range between –1 to 1.
- Decreased learning_rate
- Hidden layers contain tanh
- total_tau = 0 – alpha*driving_force

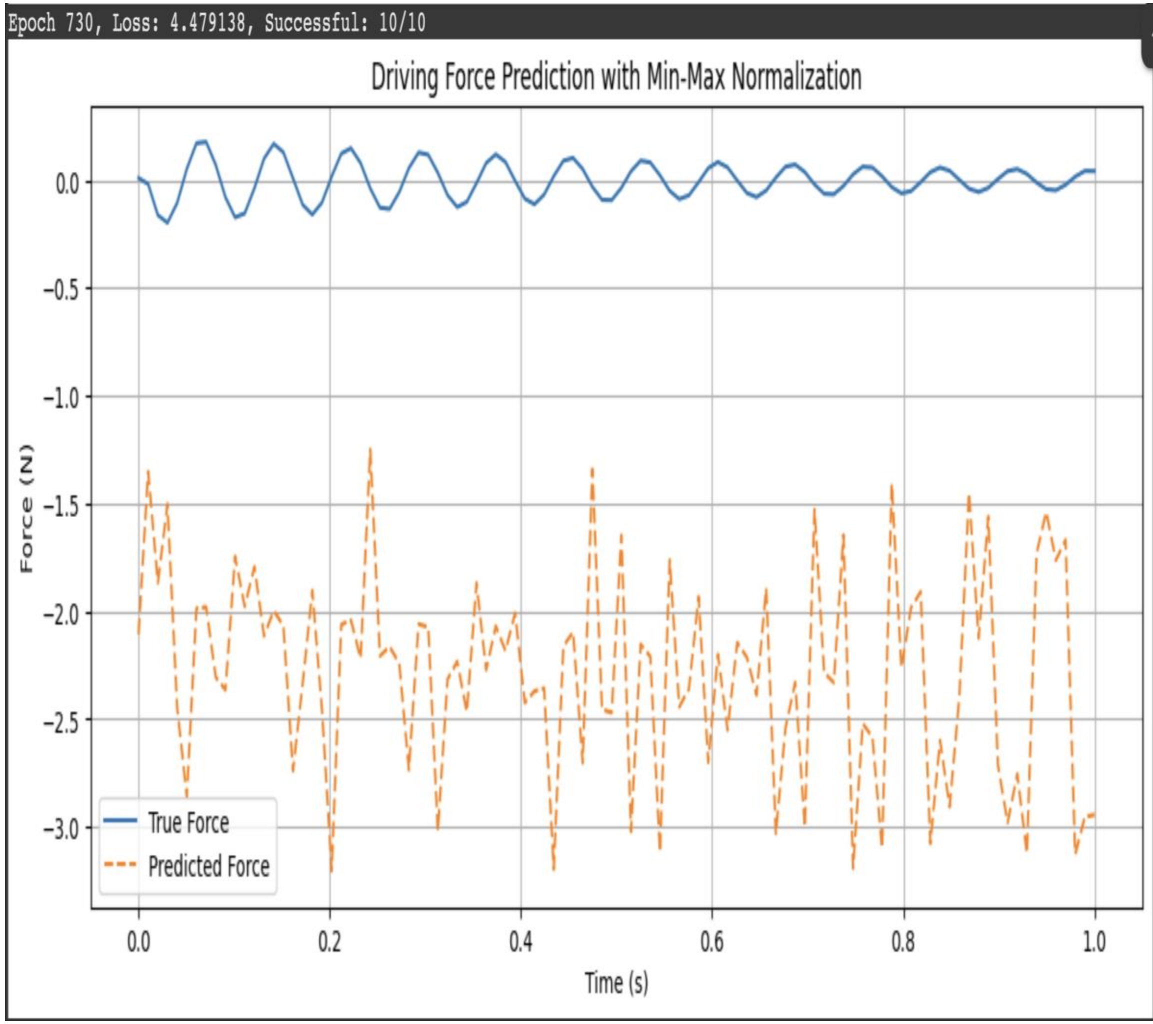
```python
# Initialize model
model = StableForcePredictor()
params = model.init(rng, jnp.ones((1, 4)))

# More conservative optimizer
optimizer = optax.chain(
    optax.clip_by_global_norm(0.1),
    optax.adam(learning_rate=1e-7),
    optax.add_decayed_weights(1e-4)  # L2 regularization
)

state = train_state.TrainState.create(
    apply_fn=model.apply,
    params=params,
    tx=optimizer
)
```

```python
    # Define the system of ODEs [unchanged]
    def equation_of_motion(y, t):
        theta, theta_dot, x, x_dot = y
        driving_force = jnp.tanh(x) * jnp.tanh(theta)
        total_tau = 0 - alpha*driving_force
        theta_ddot = (total_tau - mu * theta_dot) / I
        x_ddot = (driving_force - k * x - c * x_dot) / m
        return jnp.array([theta_dot, theta_ddot, x_dot, x_ddot])

    # Generate training data [unchanged]
    t_eval = jnp.linspace(0, 1, 100)
    y0_train = jnp.array([0.1, 0.1, 0.1, 0.1])
    solution_train = odeint(equation_of_motion, y0_train, t_eval, rtol=1e-6, atol=1e-6)

    # Calculate min-max normalization parameters
    def get_min_max(data):
        data_min = jnp.min(data, axis=0)
        data_max = jnp.max(data, axis=0)
        # Avoid division by zero for constant features
        data_range = jnp.where(data_max == data_min, 1.0, data_max - data_min)
        return data_min, data_range

    theta_min, theta_range = get_min_max(solution_train[:, 0])
    theta_dot_min, theta_dot_range = get_min_max(solution_train[:, 1])
    x_min, x_range = get_min_max(solution_train[:, 2])
    x_dot_min, x_dot_range = get_min_max(solution_train[:, 3])
```

```python
# Min-max normalization function (-1 to 1 range)
def min_max_normalize(x):
    theta = 2 * ((x[:, 0] - theta_min) / theta_range) - 1
    theta_dot = 2 * ((x[:, 1] - theta_dot_min) / theta_dot_range) - 1
    x_pos = 2 * ((x[:, 2] - x_min) / x_range) - 1
    x_dot = 2 * ((x[:, 3] - x_dot_min) / x_dot_range) - 1
    return jnp.column_stack((theta, theta_dot, x_pos, x_dot))

# Inverse normalization for predictions
def inverse_normalize(normalized, min_val, range_val):
    return (normalized + 1) * range_val / 2 + min_val

# Neural Network Definition with more stable architecture
class StableForcePredictor(nn.Module):
    @nn.compact
    def __call__(self, x):
        x_norm = min_max_normalize(x)
        x = nn.Dense(32)(x_norm)
        x = jax.nn.tanh(x)  # More stable than sin
        x = nn.Dense(16)(x)
        x = jax.nn.swish(x)
        x = nn.Dense(8)(x)
        x = jax.nn.tanh(x)
        return nn.Dense(1)(x)  # Output force prediction
```

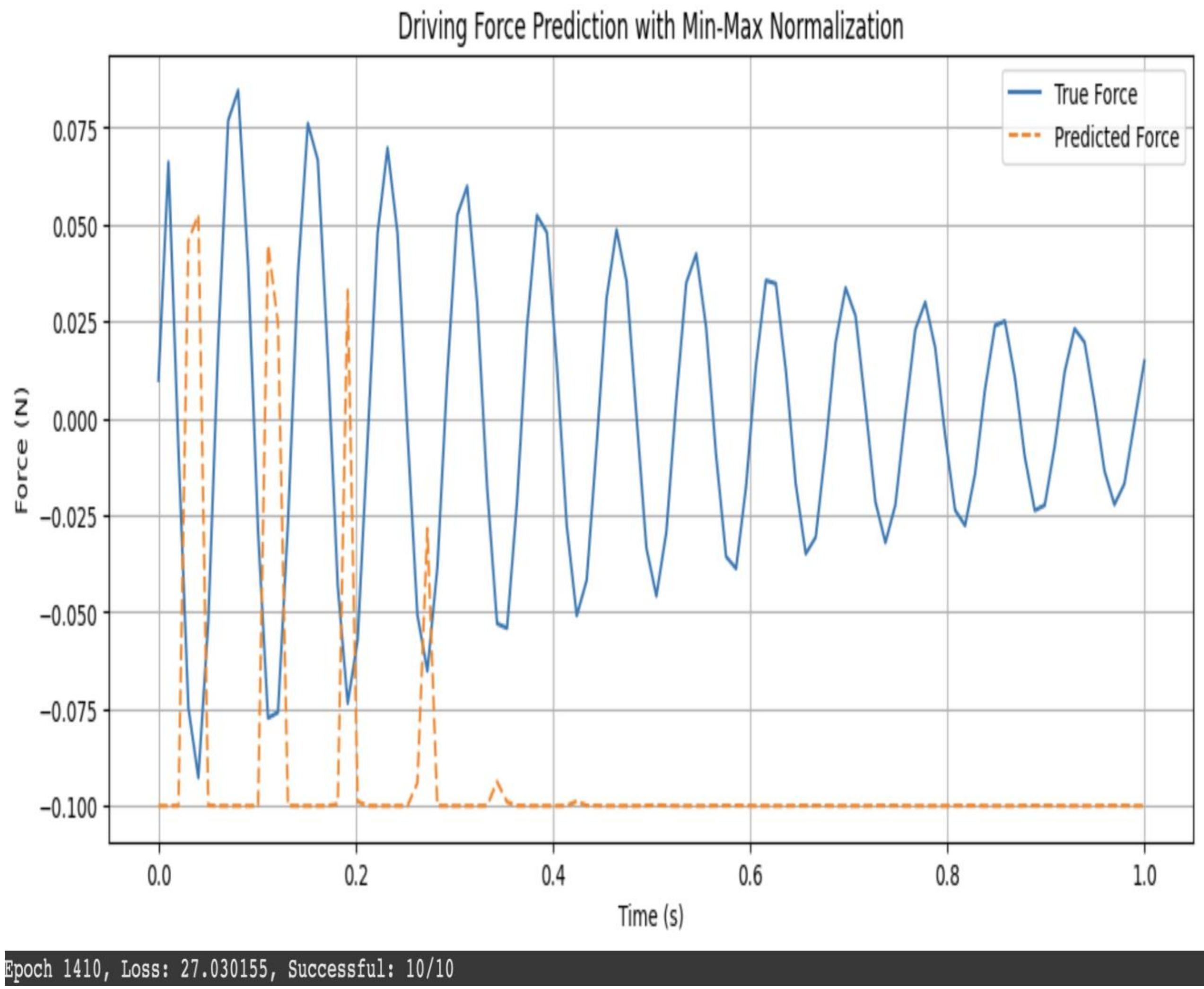
PINN

- Set driving_force = jnp.tanh(x)*jnp.tanh(theta) for simpler prediction.
- Applied Min-Max normalization for variables to range between –1 to 1.
- Decreased learning_rate
- Hidden layers contain tanh
- total_tau = 0 – alpha*driving_force
- Tolerance decreased to 10^-6. Original tolerance in odeint should follow JAX conversion from Numpy (check in GitHub).
- Prediction printed every 10 epochs
- Rest of training epochs after 730 fail: NaN



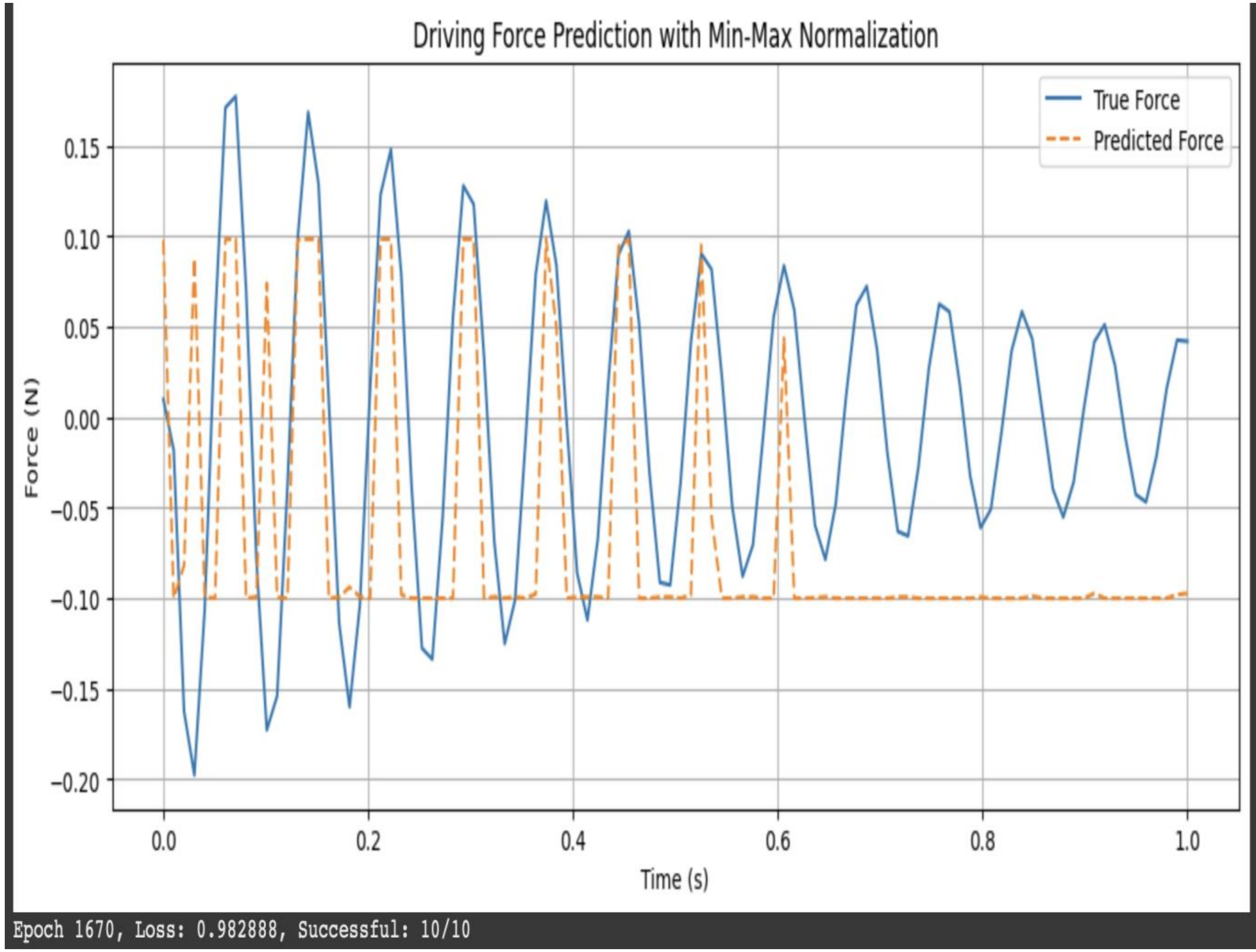
PINN

- Set driving_force = jnp.tanh(x)*jnp.tanh(theta) for simpler prediction.
- Applied Min-Max normalization for variables to range between –1 to 1.
- Decreased learning_rate
- Hidden layers contain tanh
- Output layer contains *0.1 scale and tanh following ground truth.
- total_tau = 0 – alpha*driving_force
- Tolerance decreased to 10^-6. Original tolerance in odeint should follow JAX conversion from Numpy (check in GitHub).
- Prediction printed every 10 epochs
- No NaNs.
- Losses increase



PINN

- Set driving_force = jnp.tanh(x)*jnp.tanh(theta) for simpler prediction.
- Applied Min-Max normalization for variables to range between –1 to 1.
- Decreased learning_rate
- Input to hidden layer only x and theta
- Hidden layers contain tanh
- Output layer contains *0.1 scale and tanh following ground truth.
- total_tau = 0 – alpha*driving_force
- Tolerance decreased to 10^-6. Original tolerance in odeint should follow JAX conversion from Numpy (check in GitHub).
- Prediction printed every 10 epochs
- No NaNs.
- Losses increase



FINAL COMMENTS ON PINN

- Issue with numerical instability due to odeint
- High difficulty in predicting driving_force after introducing total_tau = 0 – alpha*driving_force. Had no issue in prediction when total_tau = 0.
- Modifications show in these slides are only based on knowing the ground truth. However, not able to be done this way for actual experiment data.