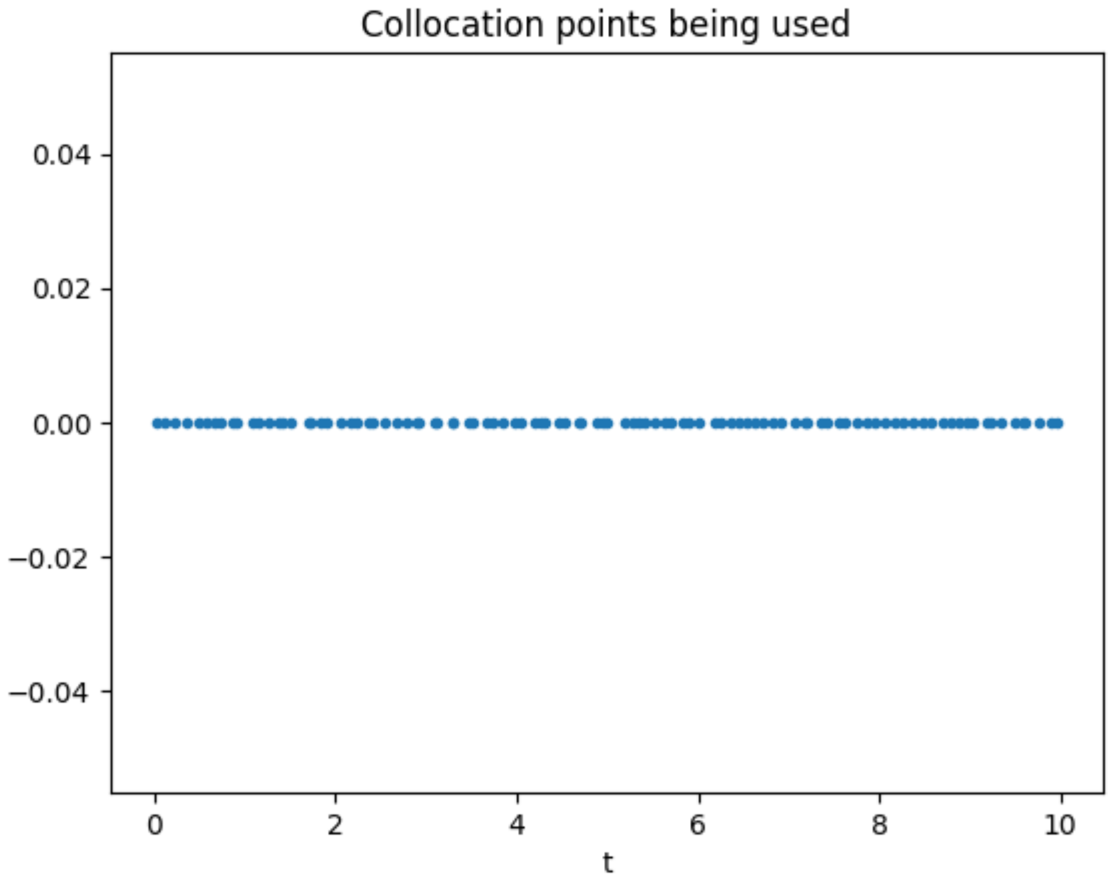
In [ ]:
```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import qmc

# Define constants
k = tf.constant(1.0, dtype=tf.float32)  # Spring constant
m = tf.constant(1.0, dtype=tf.float32)  # Mass
u0 = tf.constant(1.0, dtype=tf.float32)  # Initial displacement
u_prime0 = tf.constant(0.0, dtype=tf.float32)  # Initial velocity
t0 = 0.0
tfinal = 10.0

# Define Collocation Points
def defineCollocationPoints(t_bdry, N_de=100):
    ode_points = t_bdry[0] + (t_bdry[1] - t_bdry[0]) * qmc.LatinHypercube(d=1).random(n=N_de)
    return ode_points

de_points = defineCollocationPoints([t0, tfinal], 100)
plt.plot(de_points[:,0], 0*de_points[:,0], '.')
plt.xlabel('t')
plt.title('Collocation points being used')
plt.savefig('CollocationPoints1D.png')

# Build the model
def build_model():
    model = tf.keras.Sequential([
        tf.keras.layers.Dense(10, activation='tanh', input_shape=(1,)),
        tf.keras.layers.Dense(10, activation='tanh'),
        tf.keras.layers.Dense(1)
    ])
    return model

# Training Function for First-Order System with Hard Constraints
@tf.function
def train_network_first_order_hard(t, model, gamma=1):
    with tf.GradientTape(persistent=True) as tape:
        u = u0 + t * model(t)
        ut = tape.gradient(u, t)
        eqn = ut + (k/m) * u
        DEloss = tf.reduce_mean(eqn**2)
        loss = DEloss
    grads = tape.gradient(loss, model.trainable_variables)
    return loss, grads

def PINNtrain(de_points, model, train_function, epochs=1000):
    N_de = len(de_points)
    bs_de = N_de
    lr_model = 1e-3
    epoch_loss = np.zeros(epochs)
    nr_batches = 0
    ds = tf.data.Dataset.from_tensor_slices(de_points.astype(np.float32)).cache().shuffle(N_de).batch(bs_de)
    opt = tf.keras.optimizers.Adam(lr_model)

    for i in range(epochs):
        for des in ds:
            loss, grads = train_function(des, model)
            opt.apply_gradients(zip(grads, model.trainable_variables))
            epoch_loss[i] += loss
            nr_batches += 1
        epoch_loss[i] /= nr_batches
        nr_batches = 0
        if (np.mod(i, 100) == 0):
            print(f"Loss {i}th epoch: {epoch_loss[i]: 6.4f}")
    return epoch_loss

# Train the model
model = build_model()
epochs = 5000
loss = PINNtrain(de_points, model, train_network_first_order_hard, epochs)

# Grid where to evaluate the model
m = 100
t = np.linspace(t0, tfinal, m)

# Model prediction
u = model(np.expand_dims(t, axis=1))[:,0]

# Exact solution
uexact = u0 * np.cos(np.sqrt(k/m) * t) + (u_prime0/np.sqrt(k/m)) * np.sin(np.sqrt(k/m) * t)

# Plot the solution
fig = plt.figure(figsize=(21, 7))
plt.subplot(131)
plt.plot(t, u)
plt.plot(t, uexact, 'k--')
plt.grid()
plt.xlabel('t')
plt.ylabel('u')
plt.legend(['NN solution', 'Exact solution'])
```

```python
# Plot the error
plt.subplot(132)
plt.plot(t, u - uexact)
plt.grid()
plt.xlabel('t')
plt.ylabel('error')

# Plot the loss function
plt.subplot(133)
plt.semilogy(np.linspace(1, epochs, epochs), loss)
plt.grid()
plt.xlabel('epochs')
plt.ylabel('loss')

plt.savefig('HarmonicOscillatorPINN.png')
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
WARNING:tensorflow:Calling GradientTape.gradient on a persistent tape inside its context is significantly less efficient than calling it outside the context (it causes the gradient ops to be recorded on the tape, leading to increased CPU and memory usage). Only call GradientTape.gradient inside the context if you actually want to trace the gradient in order to compute higher order derivatives.
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-1-42f0bd541e39> in <cell line: 0>()
     68 model = build_model()
     69 epochs = 5000
---> 70 loss = PINNtrain(de_points, model, train_network_first_order_hard, epochs)
     71
     72 # Grid where to evaluate the model

<ipython-input-1-42f0bd541e39> in PINNtrain(de_points, model, train_function, epochs)
     55     for i in range(epochs):
     56         for des in ds:
---> 57             loss, grads = train_function(des, model)
     58             opt.apply_gradients(zip(grads, model.trainable_variables))
     59             epoch_loss[i] += loss

/usr/local/lib/python3.11/dist-packages/tensorflow/python/util/traceback_utils.py in error_handler(*args, **kwargs)
    151     except Exception as e:
    152       filtered_tb = _process_traceback_frames(e.__traceback__)
--> 153       raise e.with_traceback(filtered_tb) from None
    154     finally:
    155       del filtered_tb

/tmp/__autograph_generated_file436q5nmn.py in tf__train_network_first_order_hard(t, model, gamma)
     11                     u = ag__.ld(u0) + ag__.ld(t) * ag__.converted_call(ag__.ld(model), (ag__.ld(t),), None, fscope)
     12                     ut = ag__.converted_call(ag__.ld(tape).gradient, (ag__.ld(u), ag__.ld(t)), None, fscope)
---> 13                     eqn = ag__.ld(ut) + ag__.ld(k) / ag__.ld(m) * ag__.ld(u)
     14                     DEloss = ag__.converted_call(ag__.ld(tf).reduce_mean, (ag__.ld(eqn) ** 2,), None, fscope)
     15                     loss = ag__.ld(DEloss)

ValueError: in user code:

    File "<ipython-input-1-42f0bd541e39>", line 40, in train_network_first_order_hard  *
        eqn = ut + (k/m) * u

    ValueError: None values not supported.
```

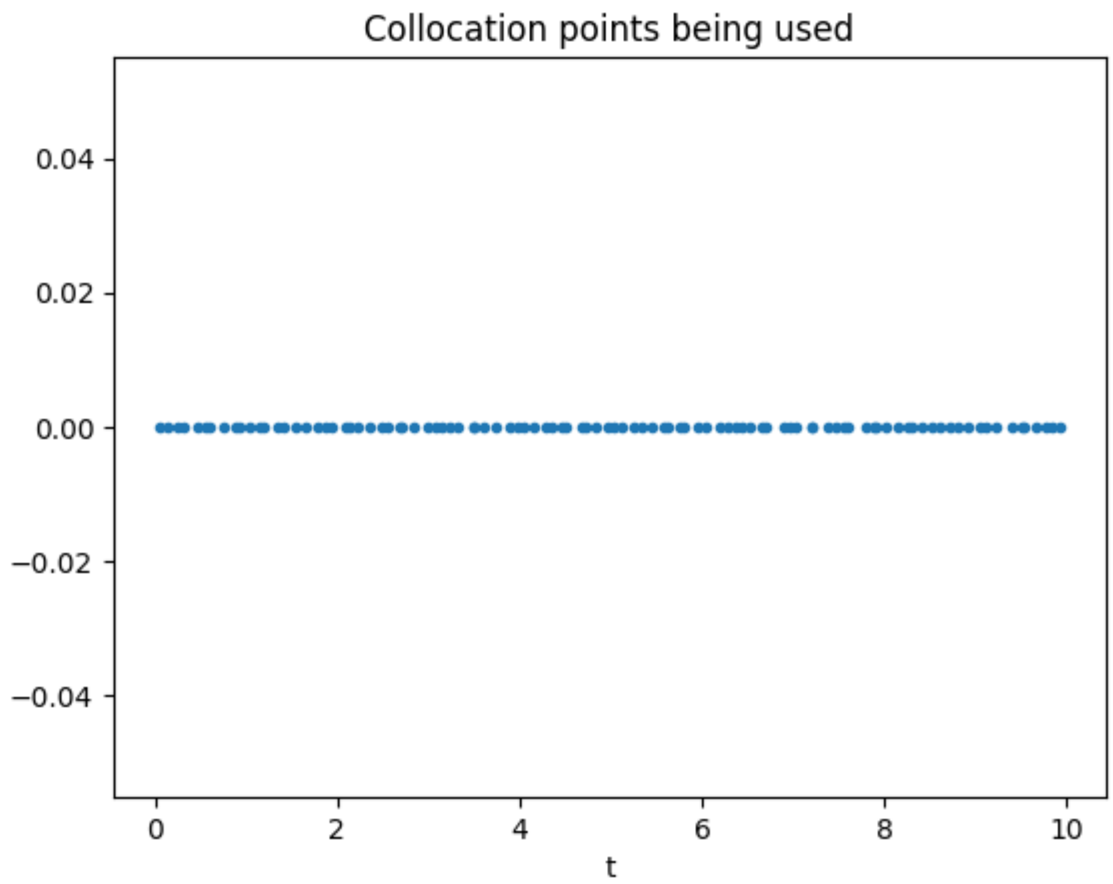

Collocation points being used
0.04
0.02
0.00
−0.02
−0.04
0
2
4
6
8
10
t

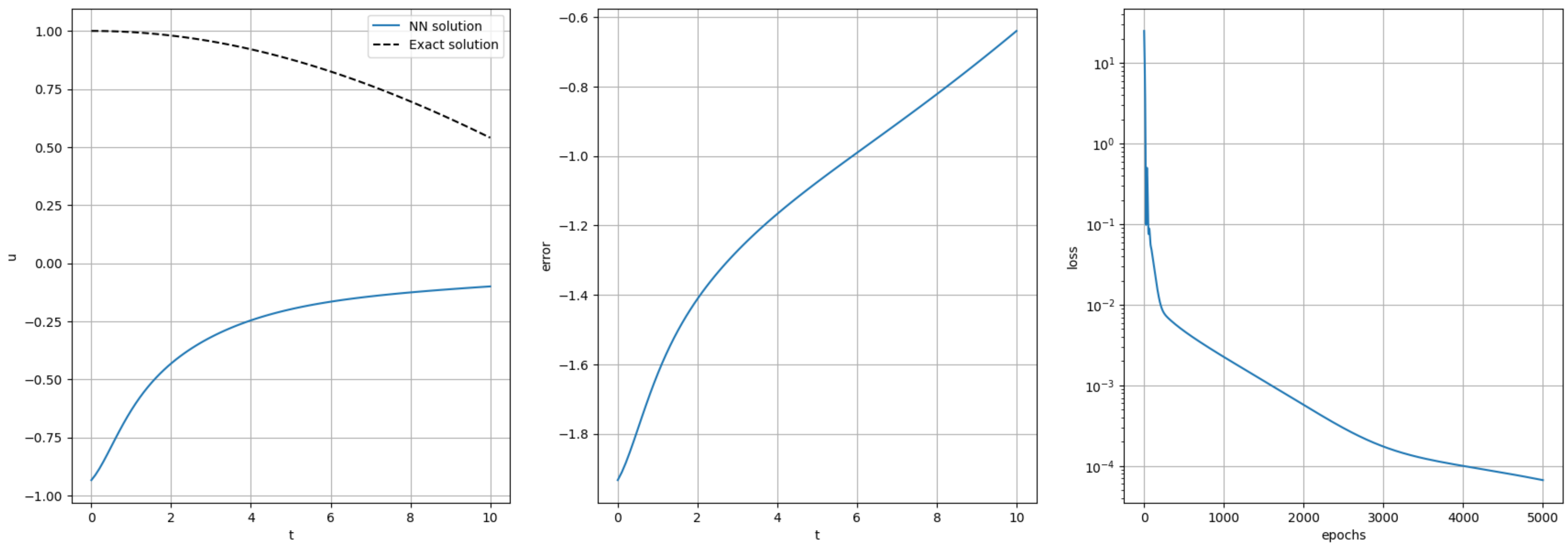
In [1]:

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import qmc

# Define constants
k = tf.constant(1.0, dtype=tf.float32)  # Spring constant
m = tf.constant(1.0, dtype=tf.float32)  # Mass
u0 = tf.constant(1.0, dtype=tf.float32)  # Initial displacement
u_prime0 = tf.constant(0.0, dtype=tf.float32)  # Initial velocity
t0 = 0.0
tfinal = 10.0

# Define Collocation Points
def defineCollocationPoints(t_bdry, N_de=100):
    ode_points = t_bdry[0] + (t_bdry[1] - t_bdry[0]) * qmc.LatinHypercube(d=1).random(n=N_de)
    return ode_points

de_points = defineCollocationPoints([t0, tfinal], 100)
plt.plot(de_points[:,0], 0*de_points[:,0], '.')
plt.xlabel('t')
plt.title('Collocation points being used')
plt.savefig('CollocationPoints1D.png')

# Build the model
def build_model():
    model = tf.keras.Sequential([
        tf.keras.layers.Dense(10, activation='tanh', input_shape=(1,)),
        tf.keras.layers.Dense(10, activation='tanh'),
        tf.keras.layers.Dense(1)
    ])
    return model

# Training Function for First-Order System with Hard Constraints
@tf.function
def train_network_first_order_hard(t, model, gamma=1):
    with tf.GradientTape(persistent=True) as tape:
        # Watch the input tensor `t`
        tape.watch(t)

        # Compute the predicted displacement `u`
        u = u0 + t * model(t)

        # Compute the derivative of `u` with respect to `t`
        ut = tape.gradient(u, t)

        # Compute the equation loss
        eqn = ut + (k/m) * u
        DEloss = tf.reduce_mean(eqn**2)
        loss = DEloss

    # Compute gradients of the loss with respect to the model's trainable variables
    grads = tape.gradient(loss, model.trainable_variables)

    # Clean up the persistent tape
    del tape

    return loss, grads

def PINNtrain(de_points, model, train_function, epochs=1000):
    N_de = len(de_points)
    bs_de = N_de
    lr_model = 1e-3
    epoch_loss = np.zeros(epochs)
    nr_batches = 0
    ds = tf.data.Dataset.from_tensor_slices(de_points.astype(np.float32)).cache().shuffle(N_de).batch(bs_de)
    opt = tf.keras.optimizers.Adam(lr_model)

    for i in range(epochs):
        for des in ds:
            loss, grads = train_function(des, model)
            opt.apply_gradients(zip(grads, model.trainable_variables))
            epoch_loss[i] += loss
            nr_batches += 1
        epoch_loss[i] /= nr_batches
        nr_batches = 0
        if (np.mod(i, 100) == 0):
            print(f"Loss {i}th epoch: {epoch_loss[i]: 6.4f}")
    return epoch_loss

# Train the model
model = build_model()
epochs = 5000
loss = PINNtrain(de_points, model, train_network_first_order_hard, epochs)

# Grid where to evaluate the model
m = 100
t = np.linspace(t0, tfinal, m)

# Model prediction
u = model(np.expand_dims(t, axis=1))[:,0]
```

```
# Exact solution
uexact = u0 * np.cos(np.sqrt(k/m) * t) + (u_prime0/np.sqrt(k/m)) * np.sin(np.sqrt(k/m) * t)

# Plot the solution
fig = plt.figure(figsize=(21, 7))
plt.subplot(131)
plt.plot(t, u)
plt.plot(t, uexact, 'k--')
plt.grid()
plt.xlabel('t')
plt.ylabel('u')
plt.legend(['NN solution', 'Exact solution'])

# Plot the error
plt.subplot(132)
plt.plot(t, u - uexact)
plt.grid()
plt.xlabel('t')
plt.ylabel('error')

# Plot the loss function
plt.subplot(133)
plt.semilogy(np.linspace(1, epochs, epochs), loss)
plt.grid()
plt.xlabel('epochs')
plt.ylabel('loss')

plt.savefig('HarmonicOscillatorPINN.png')
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
WARNING:tensorflow:Calling GradientTape.gradient on a persistent tape inside its context is significantly less efficient than calling it outside the context (it causes the gradient ops to be recorded on the tape, leading to increased CPU and memory usage). Only call GradientTape.gradient inside the context if you actually want to trace the gradient in order to compute higher order derivatives.
Loss 0th epoch:  25.0076
Loss 100th epoch:  0.0427
Loss 200th epoch:  0.0106
Loss 300th epoch:  0.0069
Loss 400th epoch:  0.0056
Loss 500th epoch:  0.0047
Loss 600th epoch:  0.0040
Loss 700th epoch:  0.0035
Loss 800th epoch:  0.0030
Loss 900th epoch:  0.0026
Loss 1000th epoch:  0.0023
Loss 1100th epoch:  0.0020
Loss 1200th epoch:  0.0017
Loss 1300th epoch:  0.0015
Loss 1400th epoch:  0.0013
Loss 1500th epoch:  0.0011
Loss 1600th epoch:  0.0010
Loss 1700th epoch:  0.0009
Loss 1800th epoch:  0.0008
Loss 1900th epoch:  0.0007
Loss 2000th epoch:  0.0006
Loss 2100th epoch:  0.0005
Loss 2200th epoch:  0.0004
Loss 2300th epoch:  0.0004
Loss 2400th epoch:  0.0003
Loss 2500th epoch:  0.0003
Loss 2600th epoch:  0.0003
Loss 2700th epoch:  0.0002
Loss 2800th epoch:  0.0002
Loss 2900th epoch:  0.0002
Loss 3000th epoch:  0.0002
Loss 3100th epoch:  0.0002
Loss 3200th epoch:  0.0001
Loss 3300th epoch:  0.0001
Loss 3400th epoch:  0.0001
Loss 3500th epoch:  0.0001
Loss 3600th epoch:  0.0001
Loss 3700th epoch:  0.0001
Loss 3800th epoch:  0.0001
Loss 3900th epoch:  0.0001
Loss 4000th epoch:  0.0001
Loss 4100th epoch:  0.0001
Loss 4200th epoch:  0.0001
Loss 4300th epoch:  0.0001
Loss 4400th epoch:  0.0001
Loss 4500th epoch:  0.0001
Loss 4600th epoch:  0.0001
Loss 4700th epoch:  0.0001
Loss 4800th epoch:  0.0001
Loss 4900th epoch:  0.0001
```





In [2]:
```
@tf.function
def train_first_order_system(t, model, k=2.0, m=1.0):
    with tf.GradientTape(persistent=True) as tape:
        # Predict u and v using the model
        outputs = model(t)
        u_pred = u0 + t * outputs[:, 0]  # Hard constraint for u
        v_pred = u_prime0 + t * outputs[:, 1]  # Hard constraint for v

        # Compute derivatives
        du_dt = tape.gradient(u_pred, t)
        dv_dt = tape.gradient(v_pred, t)

        # ODE residuals
        residual1 = du_dt - v_pred
        residual2 = dv_dt + (k/m) * u_pred

        # Loss
        loss = tf.reduce_mean(residual1**2 + residual2**2)

    grads = tape.gradient(loss, model.trainable_variables)
    return loss, grads
```

In [3]:
```
@tf.function
def train_second_order_ode(t, model, k=2.0, m=1.0):
    with tf.GradientTape(persistent=True) as tape:
        # Predict u with hard constraints
        u_pred = u0 + u_prime0 * t + t**2 * model(t)[:, 0]

        # Compute first and second derivatives
        du_dt = tape.gradient(u_pred, t)
        d2u_dt2 = tape.gradient(du_dt, t)

        # ODE residual
        residual = d2u_dt2 + (k/m) * u_pred

        # Loss
        loss = tf.reduce_mean(residual**2)

    grads = tape.gradient(loss, model.trainable_variables)
    return loss, grads
```

In [4]:
```
def build_model(output_dim=2):  # 2 outputs for the first-order system
    model = tf.keras.Sequential([
        tf.keras.layers.Dense(32, activation='tanh', input_shape=(1,)),
        tf.keras.layers.Dense(32, activation='tanh'),
        tf.keras.layers.Dense(output_dim)
    ])
    return model
```

In [5]:
```
def PINNtrain(de_points, model, train_function, epochs=5000):
    ds = tf.data.Dataset.from_tensor_slices(de_points.astype(np.float32)).batch(100)
    optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)
    loss_history = []

    for epoch in range(epochs):
        epoch_loss = 0
        for batch in ds:
            with tf.GradientTape() as tape:
                loss, grads = train_function(batch, model)
            optimizer.apply_gradients(zip(grads, model.trainable_variables))
            epoch_loss += loss.numpy()
        loss_history.append(epoch_loss / len(ds))
        if epoch % 100 == 0:
            print(f"Epoch {epoch}: Loss = {loss_history[-1]:.4f}")
    return loss_history
```

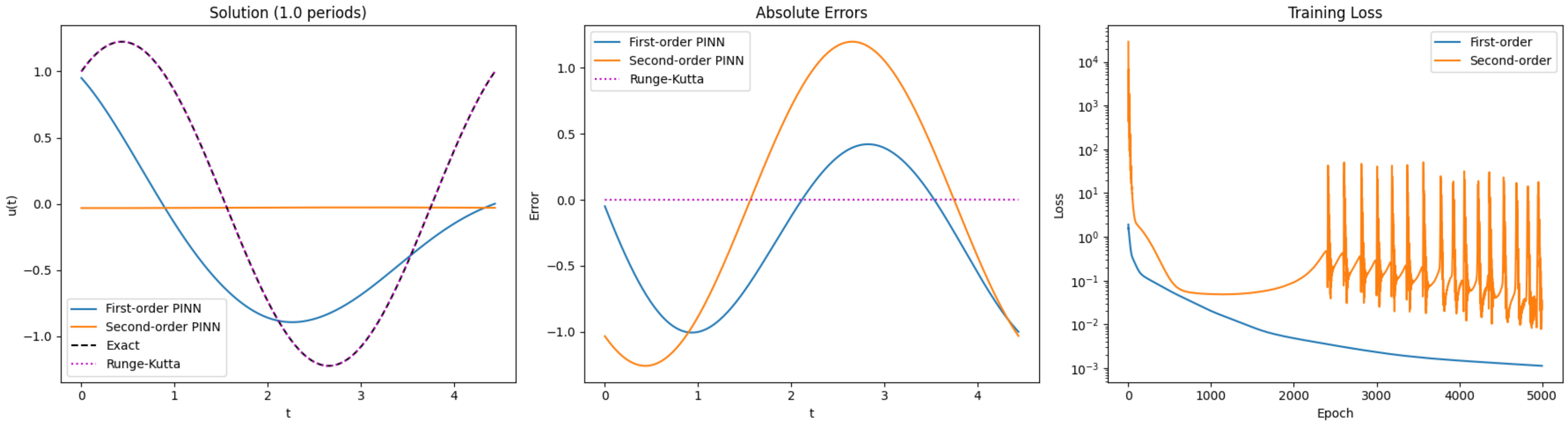
In [6]:
```
def defineCollocationPoints(t_bdry, N_de=100):
    sampler = qmc.LatinHypercube(d=1)
    samples = sampler.random(n=N_de)
    return t_bdry[0] + (t_bdry[1] - t_bdry[0]) * samples

de_points = defineCollocationPoints([0, 10], N_de=100)  # Example for t_final=10
```

In [7]:
```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import qmc
from scipy.integrate import solve_ivp

# Set seed for reproducibility
np.random.seed(42)
tf.random.set_seed(42)

# Problem parameters
m = 1.0
k = 2.0
u0 = tf.constant(1.0, dtype=tf.float32)  # Initial displacement (as TensorFlow constant)
u_prime0 = tf.constant(1.0, dtype=tf.float32)  # Initial velocity (as TensorFlow constant)
omega = np.sqrt(k/m)
period = 2*np.pi/omega
t_finals = [period, 2*period, 5*period]  # 1, 2, and 5 periods

# Neural network architecture
def build_model(output_dim=1):
    return tf.keras.Sequential([
        tf.keras.layers.Dense(32, activation='tanh', input_shape=(1,)),
        tf.keras.layers.Dense(32, activation='tanh'),
        tf.keras.layers.Dense(output_dim)
    ])

# Generate collocation points using LHS
def define_collocation_points(t_bdry, N_de=100):
    sampler = qmc.LatinHypercube(d=1)
    samples = sampler.random(n=N_de)
    return t_bdry[0] + (t_bdry[1] - t_bdry[0]) * samples

# First-order system training function
@tf.function
def train_first_order(t, model):
    with tf.GradientTape(persistent=True) as tape:
        t = tf.convert_to_tensor(t, dtype=tf.float32)
        tape.watch(t)  # Explicitly watch the input tensor t

        outputs = model(t)

        # Hard constraints
        u_pred = u0 + t * outputs[:, 0:1]
        v_pred = u_prime0 + t * outputs[:, 1:2]

        # Compute derivatives
        du_dt = tape.gradient(u_pred, t)
        dv_dt = tape.gradient(v_pred, t)

        # ODE residuals
        residual1 = du_dt - v_pred
        residual2 = dv_dt + (k/m) * u_pred

        # Total loss
        loss = tf.reduce_mean(residual1**2 + residual2**2)

    grads = tape.gradient(loss, model.trainable_variables)
    return loss, grads

# Second-order ODE training function
@tf.function
def train_second_order(t, model):
    with tf.GradientTape(persistent=True) as tape:
        t = tf.convert_to_tensor(t, dtype=tf.float32)
        tape.watch(t)  # Explicitly watch the input tensor t

        outputs = model(t)

        # Hard constraints (u(0) = u0, u'(0) = u_prime0)
        u_pred = u0 + u_prime0*t + t**2 * outputs[:, 0]

        # Compute derivatives
        du_dt = tape.gradient(u_pred, t)
        d2u_dt2 = tape.gradient(du_dt, t)

        # ODE residual
        residual = d2u_dt2 + (k/m) * u_pred

        # Total loss
        loss = tf.reduce_mean(residual**2)

    grads = tape.gradient(loss, model.trainable_variables)
    return loss, grads

# Training loop
def pinn_train(de_points, model, train_function, epochs=5000):
    ds = tf.data.Dataset.from_tensor_slices(de_points.astype(np.float32)).batch(100)
    optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)
    loss_history = []
```

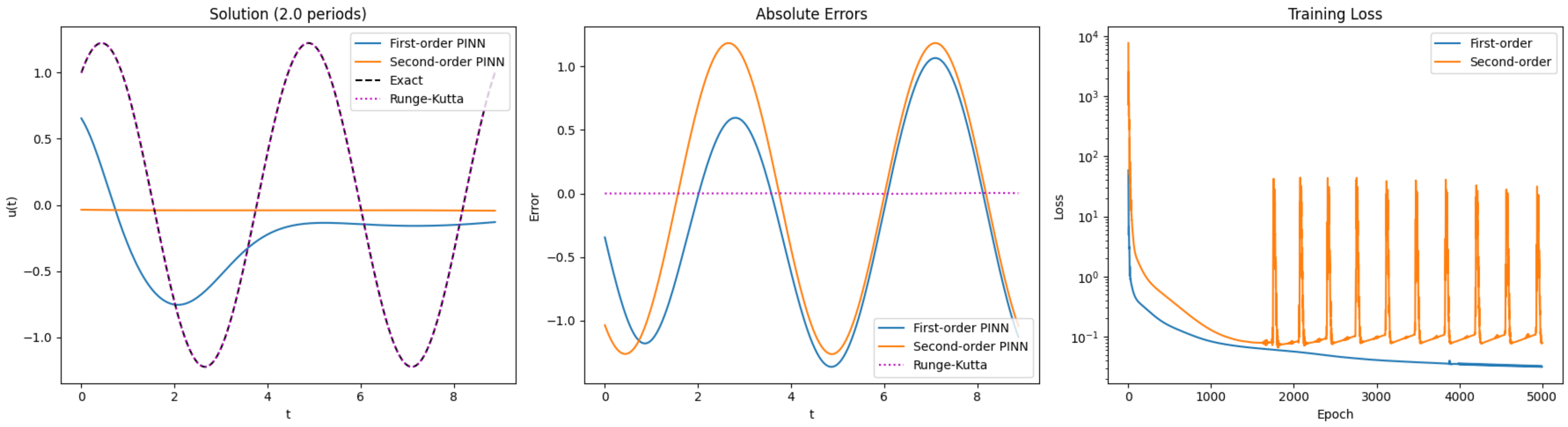
```
    for epoch in range(epochs):
        epoch_loss = 0.0
        for batch in ds:
            loss, grads = train_function(batch, model)
            optimizer.apply_gradients(zip(grads, model.trainable_variables))
            epoch_loss += loss.numpy()
        loss_history.append(epoch_loss/len(ds))
        if epoch % 500 == 0:
            print(f"Epoch {epoch}: Loss = {loss_history[-1]:.4f}")
    return loss_history

# Exact solution
def exact_solution(t):
    return u0.numpy() * np.cos(omega*t) + (u_prime0.numpy()/omega) * np.sin(omega*t)

# Solve with classical method
def solve_classical(t_span, t_eval):
    def rhs(t, y):
        return [y[1], -k/m*y[0]]

    sol = solve_ivp(rhs, t_span, [u0.numpy(), u_prime0.numpy()], t_eval=t_eval, method='RK45')
    return sol.y[0]

# Main execution
for t_final in t_finals:
    print(f"\nTraining for t_final = {t_final:.2f} ({t_final/period:.1f} periods)")

    # Generate collocation points
    de_points = define_collocation_points([0, t_final], 200)

    # Train first-order system
    model_first = build_model(output_dim=2)
    loss_first = pinn_train(de_points, model_first, train_first_order)

    # Train second-order system
    model_second = build_model(output_dim=1)
    loss_second = pinn_train(de_points, model_second, train_second_order)

    # Evaluation points
    t_test = np.linspace(0, t_final, 200).reshape(-1, 1)

    # Get predictions
    u_first = model_first(t_test)[:, 0].numpy()
    u_second = model_second(t_test).numpy().flatten()
    u_exact = exact_solution(t_test.flatten())
    u_classical = solve_classical([0, t_final], t_test.flatten())

    # Plot results
    plt.figure(figsize=(18, 5))

    # Solutions plot
    plt.subplot(1, 3, 1)
    plt.plot(t_test, u_first, label='First-order PINN')
    plt.plot(t_test, u_second, label='Second-order PINN')
    plt.plot(t_test, u_exact, 'k--', label='Exact')
    plt.plot(t_test, u_classical, 'm:', label='Runge-Kutta')
    plt.title(f'Solution ({t_final/period:.1f} periods)')
    plt.xlabel('t')
    plt.ylabel('u(t)')
    plt.legend()

    # Error plot
    plt.subplot(1, 3, 2)
    plt.plot(t_test, u_first - u_exact, label='First-order PINN')
    plt.plot(t_test, u_second - u_exact, label='Second-order PINN')
    plt.plot(t_test, u_classical - u_exact, 'm:', label='Runge-Kutta')
    plt.title('Absolute Errors')
    plt.xlabel('t')
    plt.ylabel('Error')
    plt.legend()

    # Loss plot
    plt.subplot(1, 3, 3)
    plt.semilogy(loss_first, label='First-order')
    plt.semilogy(loss_second, label='Second-order')
    plt.title('Training Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()

    plt.tight_layout()
    plt.savefig(f'results_{t_final/period:.1f}periods.png')
    plt.show()
```

```
Training for t_final = 4.44 (1.0 periods)
Epoch 0: Loss = 1.9162
Epoch 500: Loss = 0.0585
Epoch 1000: Loss = 0.0203
Epoch 1500: Loss = 0.0086
Epoch 2000: Loss = 0.0048
Epoch 2500: Loss = 0.0033
Epoch 3000: Loss = 0.0023
```

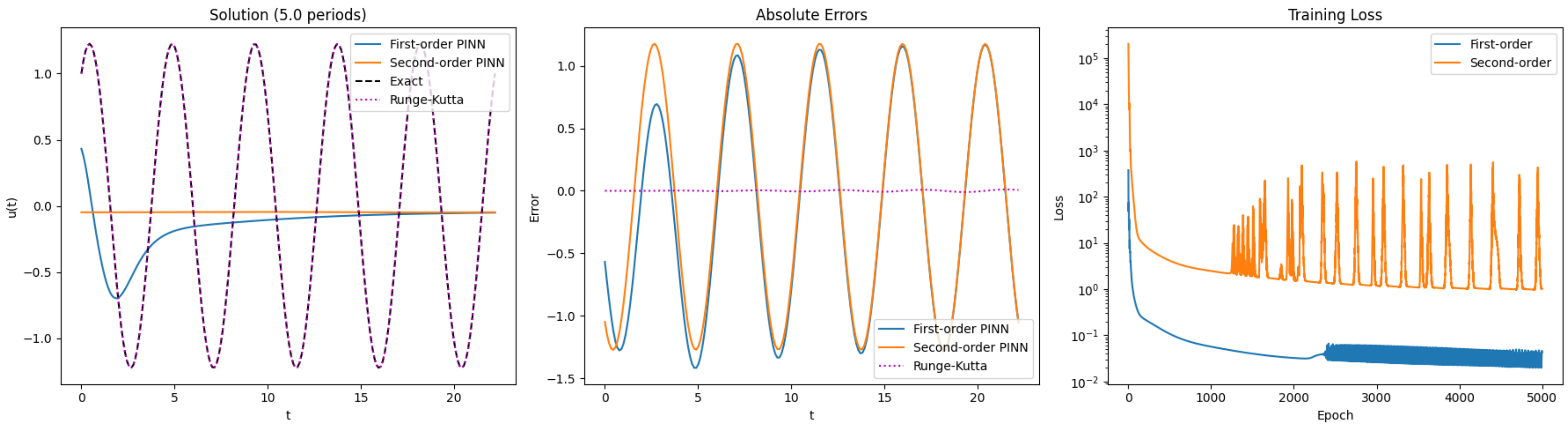
```
Epoch 3500: Loss = 0.0018
Epoch 4000: Loss = 0.0015
Epoch 4500: Loss = 0.0013
Epoch 0: Loss = 28825.1016
Epoch 500: Loss = 0.1099
Epoch 1000: Loss = 0.0492
Epoch 1500: Loss = 0.0531
Epoch 2000: Loss = 0.0919
Epoch 2500: Loss = 0.2321
Epoch 3000: Loss = 0.0459
Epoch 3500: Loss = 0.1273
Epoch 4000: Loss = 0.0733
Epoch 4500: Loss = 0.0598
```




```
Training for t_final = 8.89 (2.0 periods)
Epoch 0: Loss = 58.5968
Epoch 500: Loss = 0.1511
Epoch 1000: Loss = 0.0833
Epoch 1500: Loss = 0.0652
Epoch 2000: Loss = 0.0565
Epoch 2500: Loss = 0.0470
Epoch 3000: Loss = 0.0408
Epoch 3500: Loss = 0.0374
Epoch 4000: Loss = 0.0345
Epoch 4500: Loss = 0.0327
Epoch 0: Loss = 7684.4041
Epoch 500: Loss = 0.4165
Epoch 1000: Loss = 0.1289
Epoch 1500: Loss = 0.0802
Epoch 2000: Loss = 0.0805
Epoch 2500: Loss = 0.0800
Epoch 3000: Loss = 0.0947
Epoch 3500: Loss = 2.6970
Epoch 4000: Loss = 0.0872
Epoch 4500: Loss = 0.1019
```




```
Training for t_final = 22.21 (5.0 periods)
Epoch 0: Loss = 374.9545
Epoch 500: Loss = 0.1070
Epoch 1000: Loss = 0.0565
Epoch 1500: Loss = 0.0399
Epoch 2000: Loss = 0.0321
Epoch 2500: Loss = 0.0286
Epoch 3000: Loss = 0.0408
Epoch 3500: Loss = 0.0269
Epoch 4000: Loss = 0.0512
Epoch 4500: Loss = 0.0469
Epoch 0: Loss = 204181.5547
Epoch 500: Loss = 4.1159
Epoch 1000: Loss = 2.4595
Epoch 1500: Loss = 2.1663
Epoch 2000: Loss = 2.3502
Epoch 2500: Loss = 2.8992
Epoch 3000: Loss = 1.1939
Epoch 3500: Loss = 1.0841
Epoch 4000: Loss = 1.0395
Epoch 4500: Loss = 1.0616
```


Solution (5.0 periods)
First-order PINN
Second-order PINN
Exact
Runge-Kutta
u(t)
t
Absolute Errors
Error
First-order PINN
Second-order PINN
Runge-Kutta
t
Training Loss
First-order
Second-order
Loss
Epoch