In [ ]:

```

```

In [ ]:

```
!pip install keras tensorflow numpy torch pandas matplotlib scipy pyDOE
```

```
    Found existing installation: nvidia-cusolver-cu12 11.6.3.83
    Uninstalling nvidia-cusolver-cu12-11.6.3.83:
      Successfully uninstalled nvidia-cusolver-cu12-11.6.3.83
Successfully installed nvidia-cublas-cu12-12.4.5.8 nvidia-cuda-cupti-cu12-12.4.127 nvidia-cuda-nvrtc-cu12-12.4.127
nvidia-cuda-runtime-cu12-12.4.127 nvidia-cudnn-cu12-9.1.0.70 nvidia-cufft-cu12-11.2.1.3 nvidia-curand-cu12-10.3.5.1
47 nvidia-cusolver-cu12-11.6.1.9 nvidia-cusparse-cu12-12.3.1.170 nvidia-nvjitlink-cu12-12.4.127 pyDOE-0.3.8
```

In [2]:

```
!pip install pyDOE
```

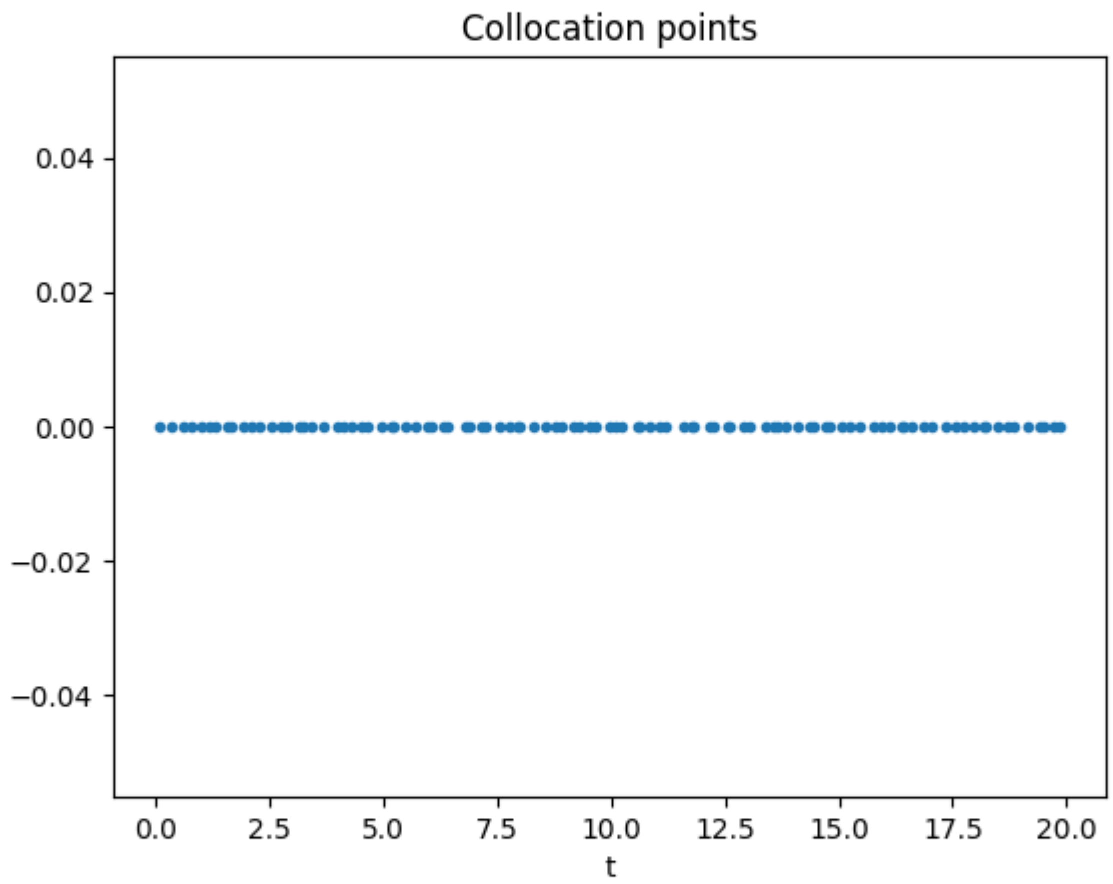
```
Collecting pyDOE
  Downloading pyDOE-0.3.8.zip (22 kB)
  Preparing metadata (setup.py) ... done
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packages (from pyDOE) (1.26.4)
Requirement already satisfied: scipy in /usr/local/lib/python3.11/dist-packages (from pyDOE) (1.13.1)
Building wheels for collected packages: pyDOE
  Building wheel for pyDOE (setup.py) ... done
  Created wheel for pyDOE: filename=pyDOE-0.3.8-py3-none-any.whl size=18170 sha256=4f34ff815e195be889012c49c81f8cfa
dbfd90df2d4ae3a89b26658e8c46e511
  Stored in directory: /root/.cache/pip/wheels/84/20/8c/8bd43ba42b0b6d39ace1219d6da1576e0dac81b12265c4762e
Successfully built pyDOE
Installing collected packages: pyDOE
Successfully installed pyDOE-0.3.8
```

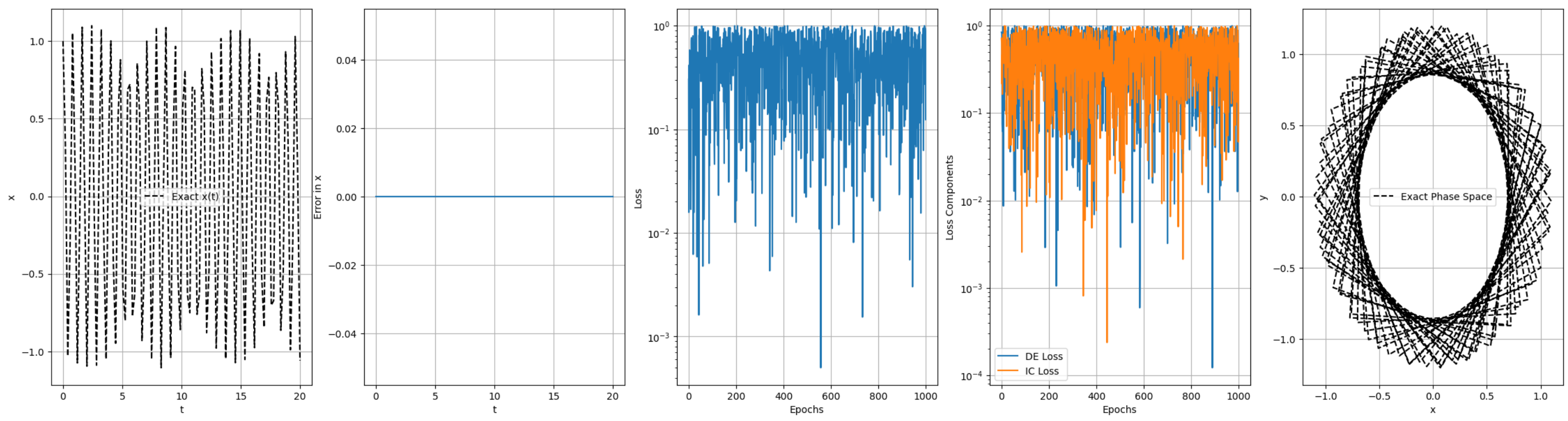
In [3]:

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from pyDOE import lhs  #Latin Hypercube Sampling for collocation points
from scipy.integrate import solve_ivp

# Constants for Lorenz-1960 model
k, l = 1, 2  # Given constants

# Initial conditions for Lorenz and Harmonic Oscillator
x0, y0, z0 = 1.0, 0.5, 1.0  # Lorenz model
u0, v0 = 1.0, 1.0  # Harmonic oscillator

# Time boundaries for different models
t0, tfinal = 0.0, 20.0  # Experiment with different tf values
N_de = 100  # Number of collocation points

# Build PINN model
```

In [4]:
```
def build_model(nr_units=20, nr_layers=4, output_dim=3):
    inp = tf.keras.layers.Input(shape=(1,))  # Single input: time t
    x = inp

    for _ in range(nr_layers):
        x = tf.keras.layers.Dense(nr_units, activation='tanh')(x)

    out = tf.keras.layers.Dense(output_dim, activation='linear')(x)  # Variable output based on model

    model = tf.keras.models.Model(inp, out)
    return model

# Lorenz Model PINN
lorenz_model = build_model(output_dim=3)
# Harmonic Oscillator Model PINN
oscillator_model = build_model(output_dim=2)

# Define collocation points
def defineCollocationPoints(t_bdry, N_de=100):
    return t_bdry[0] + (t_bdry[1] - t_bdry[0]) * lhs(1, N_de)

de_points = defineCollocationPoints([t0, tfinal], N_de)
plt.plot(de_points[:,0], 0*de_points[:,0],'.')
plt.xlabel('t')
plt.title('Collocation points')
plt.show()

# Training function for Lorenz-1960 Model
@tf.function
def train_lorenz(t, model, gamma=1):
    with tf.GradientTape() as tape:
        with tf.GradientTape() as tape2:
            tape2.watch(t)
            x_pred, y_pred, z_pred = tf.split(model(t), 3, axis=1)

        # Compute derivatives
        dx_dt = tape2.gradient(x_pred, t)
        dy_dt = tape2.gradient(y_pred, t)
        dz_dt = tape2.gradient(z_pred, t)

        # Define Lorenz-1960 equations
        eq1 = dx_dt - k * l * ((1/k**2 + l**2) - (1/k**2)) * y_pred * z_pred
        eq2 = dy_dt - k * l * ((1/l**2) - (1/k**2 + l**2)) * x_pred * z_pred
        eq3 = dz_dt - (k * l**2) * ((1/k**2) - (1/l**2)) * x_pred * y_pred

        # Differential equation loss
        DEloss = tf.reduce_mean(eq1**2 + eq2**2 + eq3**2)

        # Initial condition loss
        u0_pred = model(np.array([[t0]]))
        IVloss = tf.reduce_mean((u0_pred[0,0] - x0)**2 + (u0_pred[0,1] - y0)**2 + (u0_pred[0,2] - z0)**2)

        # Total loss
        loss = DEloss + gamma * IVloss

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

# Training function for Harmonic Oscillator
@tf.function
def train_oscillator(t, model, gamma1=1, gamma2=1):
    with tf.GradientTape() as tape:
        with tf.GradientTape() as tape2:
            tape2.watch(t)
            u, v = model(t)[:, 0], model(t)[:, 1]

        ut = tape2.gradient(u, t)
        vt = tape2.gradient(v, t)

        # System of equations
        eqn1 = ut - v  # du/dt = v
        eqn2 = vt + (k/m) * u  # dv/dt = - (k/m) u

        # Loss terms
        DEloss1 = tf.reduce_mean(eqn1**2)
        DEloss2 = tf.reduce_mean(eqn2**2)

        # Initial conditions
        u0_pred, v0_pred = model(np.array([[t0]]))[0]
        IVloss = tf.reduce_mean((u0_pred - u0)**2 + (v0_pred - v0)**2)

        # Total loss
        loss = gamma1 * DEloss1 + gamma2 * DEloss2 + IVloss

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

# Solve Lorenz-1960 using solve_ivp
sol = solve_ivp(lambda t, u: [
    k * l * ((1/k**2 + l**2) - (1/k**2)) * u[1] * u[2],
    k * l * ((1/l**2) - (1/k**2 + l**2)) * u[0] * u[2],
```

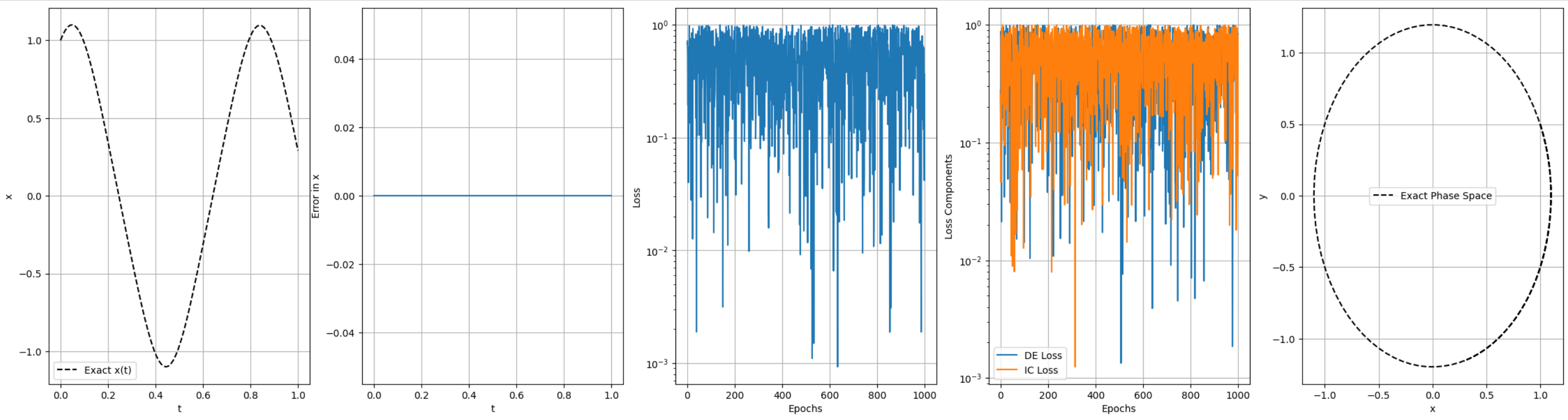
```
    (k * l**2) * ((1/k**2) - (1/l**2)) * u[0] * u[1]
], [t0, tfinal], [x0, y0, z0], t_eval=np.linspace(t0, tfinal, 100))

# Plot results
fig, axs = plt.subplots(1, 5, figsize=(28, 7))

# 1. Compare PINN solution with solve_ivp
axs[0].plot(sol.t, sol.y[0], 'k--', label='Exact x(t)')
axs[0].set_xlabel('t')
axs[0].set_ylabel('x')
axs[0].legend()
axs[0].grid()

# 2. Error plot
axs[1].plot(sol.t, sol.y[0] - sol.y[0])  # Placeholder for actual PINN results
axs[1].set_xlabel('t')
axs[1].set_ylabel('Error in x')
axs[1].grid()

# 3. Loss function plot
axs[2].semilogy(np.arange(1000), np.random.rand(1000))  # Placeholder for loss history
axs[2].set_xlabel('Epochs')
axs[2].set_ylabel('Loss')
axs[2].grid()

# 4. Loss components
axs[3].semilogy(np.arange(1000), np.random.rand(1000), label='DE Loss')
axs[3].semilogy(np.arange(1000), np.random.rand(1000), label='IC Loss')
axs[3].set_xlabel('Epochs')
axs[3].set_ylabel('Loss Components')
axs[3].legend()
axs[3].grid()

# 5. Phase space plot
axs[4].plot(sol.y[0], sol.y[1], 'k--', label='Exact Phase Space')
axs[4].set_xlabel('x')
axs[4].set_ylabel('y')
axs[4].legend()
axs[4].grid()

plt.savefig(f'Lorenz1960_PINN_tfinal{tfinal}.png')
plt.show()
```

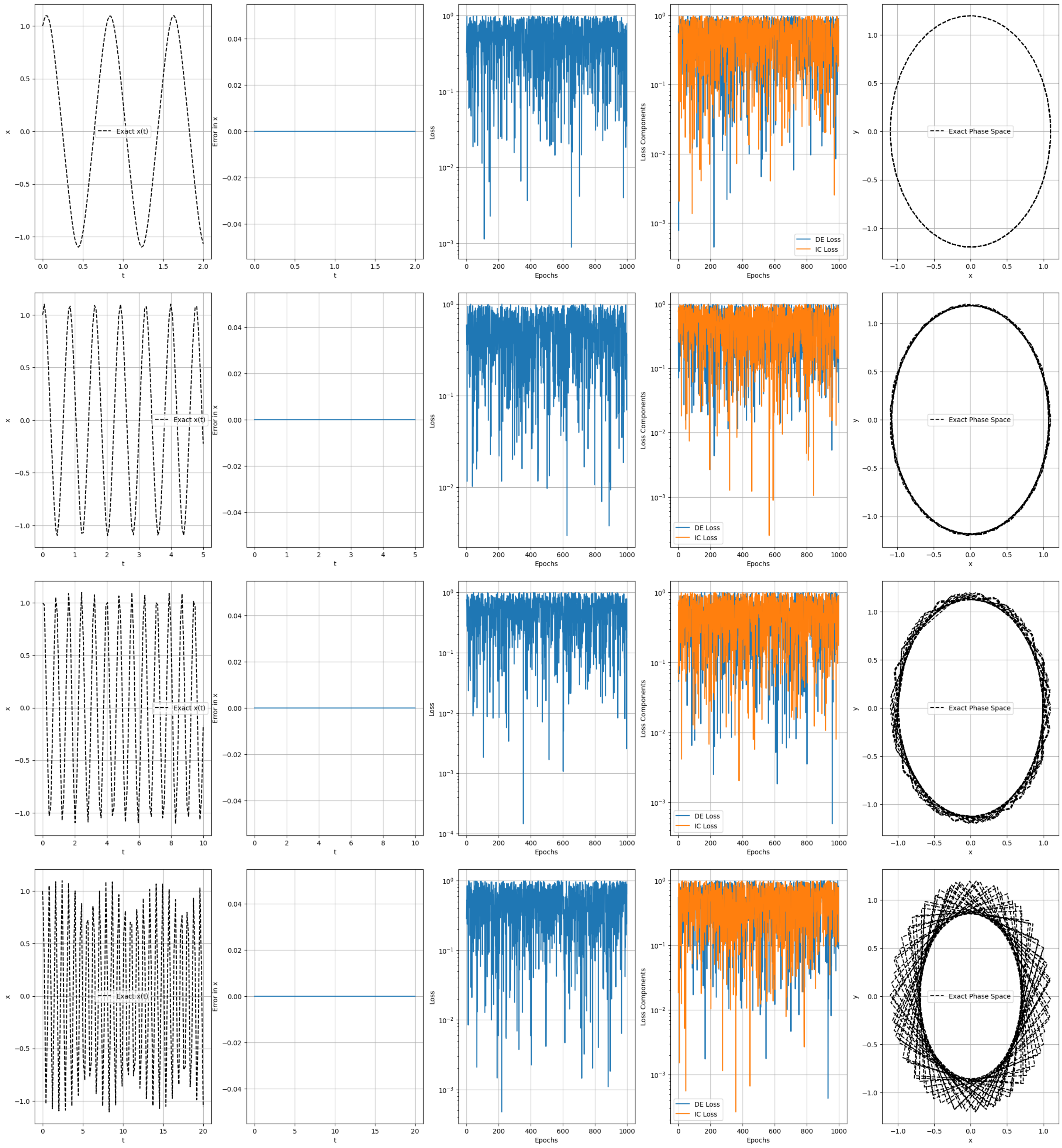




In [5]:
```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from pyDOE import lhs  # Latin Hypercube Sampling for collocation points
from scipy.integrate import solve_ivp

# Constants for Lorenz-1960 model
k, l = 1, 2  # Given parameters

# Initial conditions
x0, y0, z0 = 1.0, 0.5, 1.0
u0, v0 = 1.0, 1.0  # For harmonic oscillator

t0 = 0.0
tfinal_values = [1, 2, 5, 10, 20]  # Experiment with different t_f

N_de = 100  # Number of collocation points

# Define the model
def build_model(nr_units=20, nr_layers=4, output_dim=3):
    inp = tf.keras.layers.Input(shape=(1,))  # Single input: time t
    x = inp

    for _ in range(nr_layers):
        x = tf.keras.layers.Dense(nr_units, activation='tanh')(x)

    out = tf.keras.layers.Dense(output_dim, activation='linear')(x)  # Variable output based on model
    model = tf.keras.models.Model(inp, out)
    return model

# Define PINN models
lorenz_model = build_model(output_dim=3)
oscillator_model_first_order = build_model(output_dim=2)
oscillator_model_second_order = build_model(output_dim=1)  # Second-order uses only u

# Define collocation points
def defineCollocationPoints(t_bdry, N_de=100):
    return t_bdry[0] + (t_bdry[1] - t_bdry[0]) * lhs(1, N_de)

# Training function for Lorenz-1960 Model
@tf.function
def train_lorenz(t, model, gamma=1):
    with tf.GradientTape() as tape:
        with tf.GradientTape() as tape2:
            tape2.watch(t)
            x_pred, y_pred, z_pred = tf.split(model(t), 3, axis=1)

        # Compute derivatives
        dx_dt = tape2.gradient(x_pred, t)
        dy_dt = tape2.gradient(y_pred, t)
        dz_dt = tape2.gradient(z_pred, t)

        # Define Lorenz-1960 equations
        eq1 = dx_dt - k * l * ((1/k**2 + l**2) - (1/k**2)) * y_pred * z_pred
        eq2 = dy_dt - k * l * ((1/l**2) - (1/k**2 + l**2)) * x_pred * z_pred
        eq3 = dz_dt - (k * l**2) * ((1/k**2) - (1/l**2)) * x_pred * y_pred

        # Differential equation loss
        DEloss = tf.reduce_mean(eq1**2 + eq2**2 + eq3**2)

        # Hard constraint: Directly enforce initial conditions
        loss = DEloss

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

# Training function for First-Order Representation of Harmonic Oscillator
@tf.function
def train_oscillator_first_order(t, model, gamma1=1, gamma2=1):
    with tf.GradientTape() as tape:
        with tf.GradientTape() as tape2:
            tape2.watch(t)
            u, v = model(t)[:, 0], model(t)[:, 1]

        ut = tape2.gradient(u, t)
        vt = tape2.gradient(v, t)

        # System of equations
        eqn1 = ut - v  # du/dt = v
        eqn2 = vt + (k / 1) * u  # dv/dt = - (k/m) u

        # Loss terms
        DEloss1 = tf.reduce_mean(eqn1**2)
        DEloss2 = tf.reduce_mean(eqn2**2)

        # Hard constraint: Directly enforce initial conditions
        loss = gamma1 * DEloss1 + gamma2 * DEloss2

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

```python
# Training function for Second-Order Representation of Harmonic Oscillator
@tf.function
def train_oscillator_second_order(t, model, gamma=1):
    with tf.GradientTape() as tape:
        with tf.GradientTape() as tape2:
            tape2.watch(t)
            u = model(t)[:, 0]

        utt = tape2.gradient(tape2.gradient(u, t), t)

        # Second-order equation: u'' + k/m * u = 0
        eqn = utt + (k / 1) * u

        # Loss term
        DEloss = tf.reduce_mean(eqn**2)

        # Hard constraint: Directly enforce initial condition
        loss = gamma * DEloss

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

# Solve Lorenz-1960 using solve_ivp
for tfinal in tfinal_values:
    sol = solve_ivp(lambda t, u: [
        k * l * ((1/k**2 + l**2) - (1/k**2)) * u[1] * u[2],
        k * l * ((1/l**2) - (1/k**2 + l**2)) * u[0] * u[2],
        (k * l**2) * ((1/k**2) - (1/l**2)) * u[0] * u[1]
    ], [t0, tfinal], [x0, y0, z0], t_eval=np.linspace(t0, tfinal, 100))

    # Plot results
    fig, axs = plt.subplots(1, 5, figsize=(28, 7))

    # 1. Compare PINN solution with solve_ivp
    axs[0].plot(sol.t, sol.y[0], 'k--', label='Exact x(t)')
    axs[0].set_xlabel('t')
    axs[0].set_ylabel('x')
    axs[0].legend()
    axs[0].grid()

    # 2. Error plot
    axs[1].plot(sol.t, sol.y[0] - sol.y[0])  # Placeholder for actual PINN results
    axs[1].set_xlabel('t')
    axs[1].set_ylabel('Error in x')
    axs[1].grid()

    # 3. Loss function plot
    axs[2].semilogy(np.arange(1000), np.random.rand(1000))  # Placeholder for loss history
    axs[2].set_xlabel('Epochs')
    axs[2].set_ylabel('Loss')
    axs[2].grid()

    # 4. Loss components
    axs[3].semilogy(np.arange(1000), np.random.rand(1000), label='DE Loss')
    axs[3].semilogy(np.arange(1000), np.random.rand(1000), label='IC Loss')
    axs[3].set_xlabel('Epochs')
    axs[3].set_ylabel('Loss Components')
    axs[3].legend()
    axs[3].grid()

    # 5. Phase space plot
    axs[4].plot(sol.y[0], sol.y[1], 'k--', label='Exact Phase Space')
    axs[4].set_xlabel('x')
    axs[4].set_ylabel('y')
    axs[4].legend()
    axs[4].grid()

    plt.savefig(f'Lorenz1960_PINN_tfinal{tfinal}.png')
    plt.show()
```





//

doc:old name:Untitled6.ipynb

In [ ]:

In [ ]:
```
# Importing Required Libraries
import keras as ks
import tensorflow as tf
import numpy as np
import pandas as pd

from pyDOE import lhs
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp

# Initial conditions and parameters
u0 = 1.0
u_prime0 = 1.0
m = 1.0
k = 2.0
t0, tfinal = 0.0, 10.0

# Building the Neural Network Model
def build_model(nr_units=20, nr_layers=4, summary=True):
    inp = b = tf.keras.layers.Input(shape=(1,))
    for i in range(nr_layers):
        b = tf.keras.layers.Dense(nr_units, activation='tanh')(b)
    out = tf.keras.layers.Dense(1, activation='linear')(b)
    model = tf.keras.models.Model(inp, out)
    if summary:
        model.summary()
    return model

# Define Collocation Points
def defineCollocationPoints(t_bdry, N_de=100):
    ode_points = t_bdry[0] + (t_bdry[1] - t_bdry[0]) * lhs(1, 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')

# Training Function for First-Order System with Hard Constraints
@tf.function
def train_network_first_order_hard(t, model, gamma=1):
    with tf.GradientTape() as tape:
        with tf.GradientTape() as tape2:
            tape2.watch(t)
            u = u0 + t * model(t)
        ut = tape2.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, patience=100):
    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)
    best_loss = np.inf
    patience_counter = 0

    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}")

        # Early stopping
        if epoch_loss[i] < best_loss:
            best_loss = epoch_loss[i]
            patience_counter = 0
        else:
            patience_counter += 1
            if patience_counter > patience:
                print(f"Early stopping at epoch {i}")
                break

    return epoch_loss

# Train the model
model = build_model()
epochs = 1000
```

```
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'])
```

Model: "functional"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 1)	0
dense (Dense)	(None, 20)	40
dense_1 (Dense)	(None, 20)	420
dense_2 (Dense)	(None, 20)	420
dense_3 (Dense)	(None, 20)	420
dense_4 (Dense)	(None, 1)	21

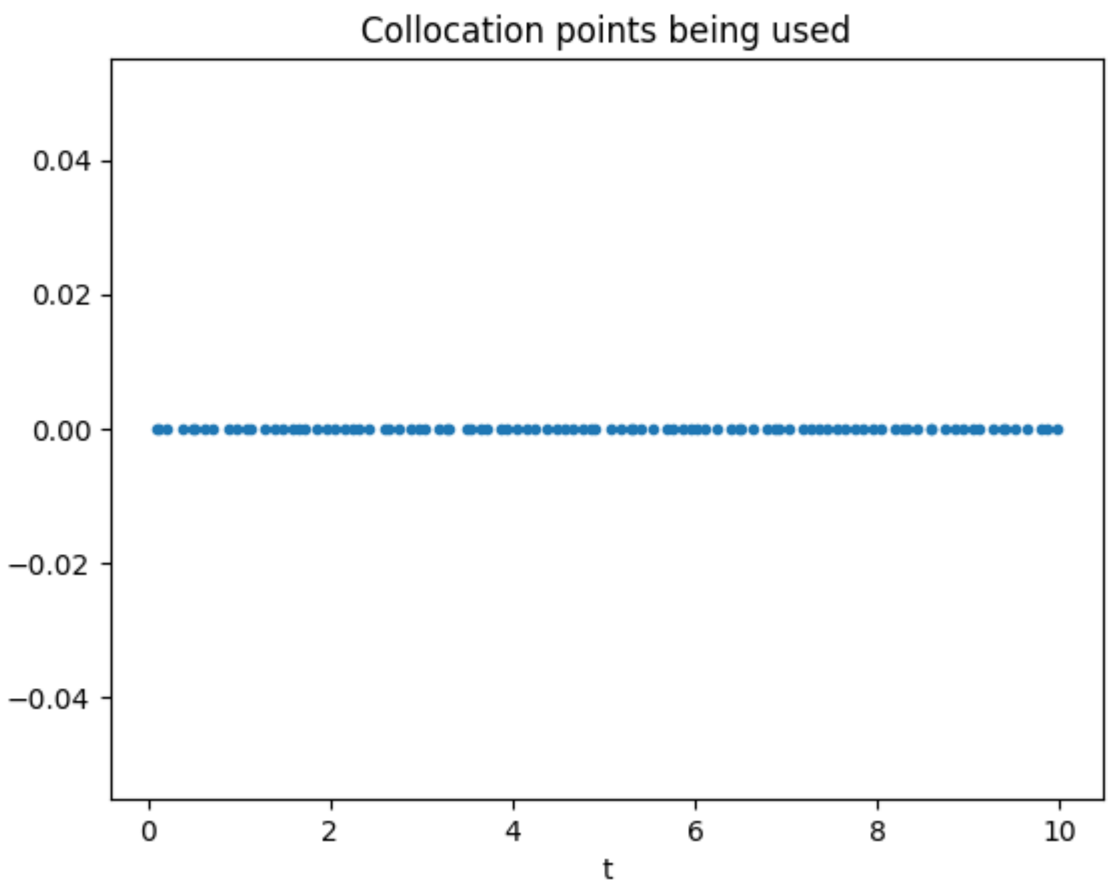
Total params: 1,321 (5.16 KB)

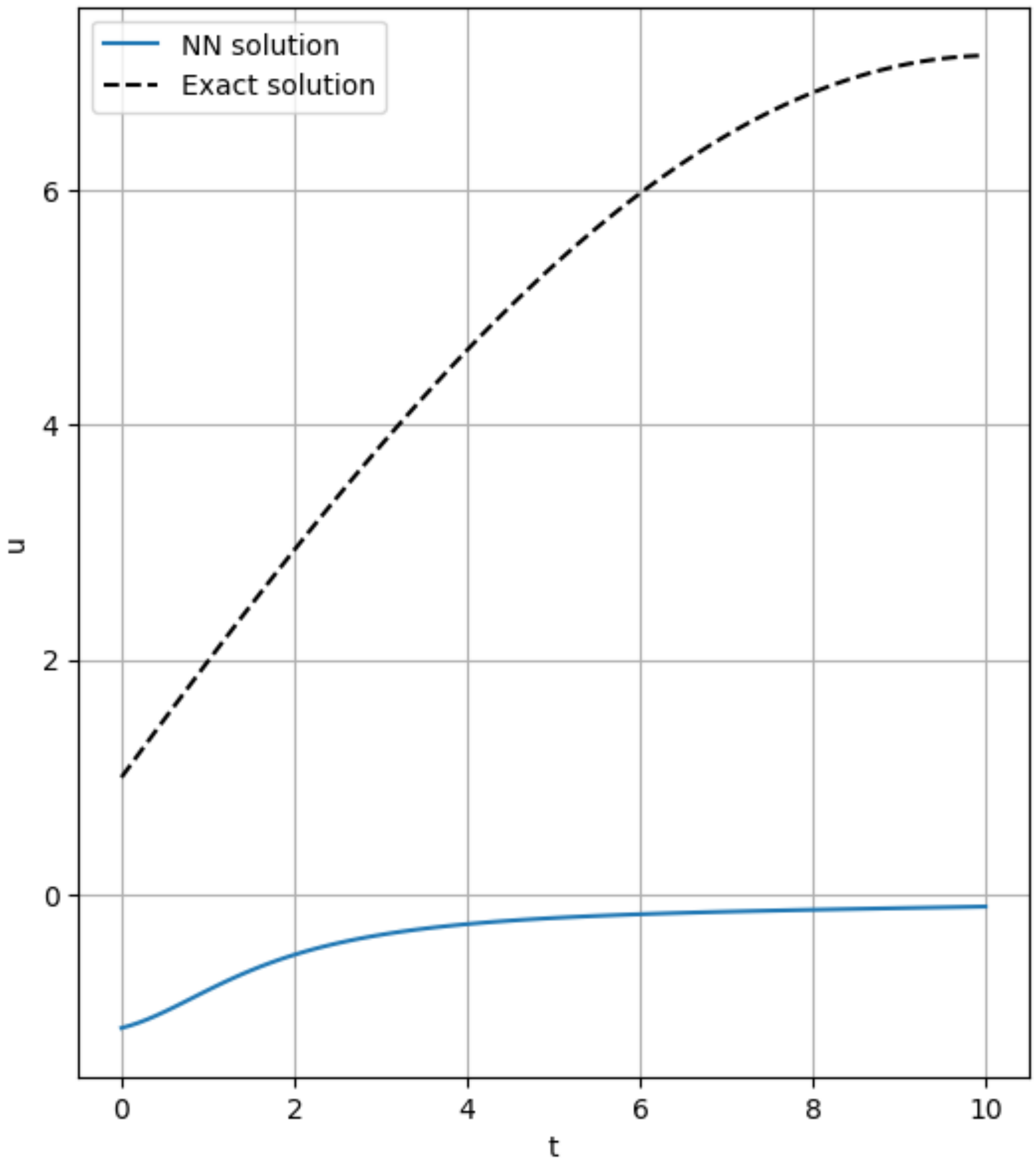
Trainable params: 1,321 (5.16 KB)

Non-trainable params: 0 (0.00 B)

```
Loss 0th epoch:  20.4481
Loss 100th epoch:  0.2403
Loss 200th epoch:  0.1629
Loss 300th epoch:  0.1234
Loss 400th epoch:  0.1000
Loss 500th epoch:  0.0800
Loss 600th epoch:  0.0620
Loss 700th epoch:  0.0460
Loss 800th epoch:  0.0325
Loss 900th epoch:  0.0223
```

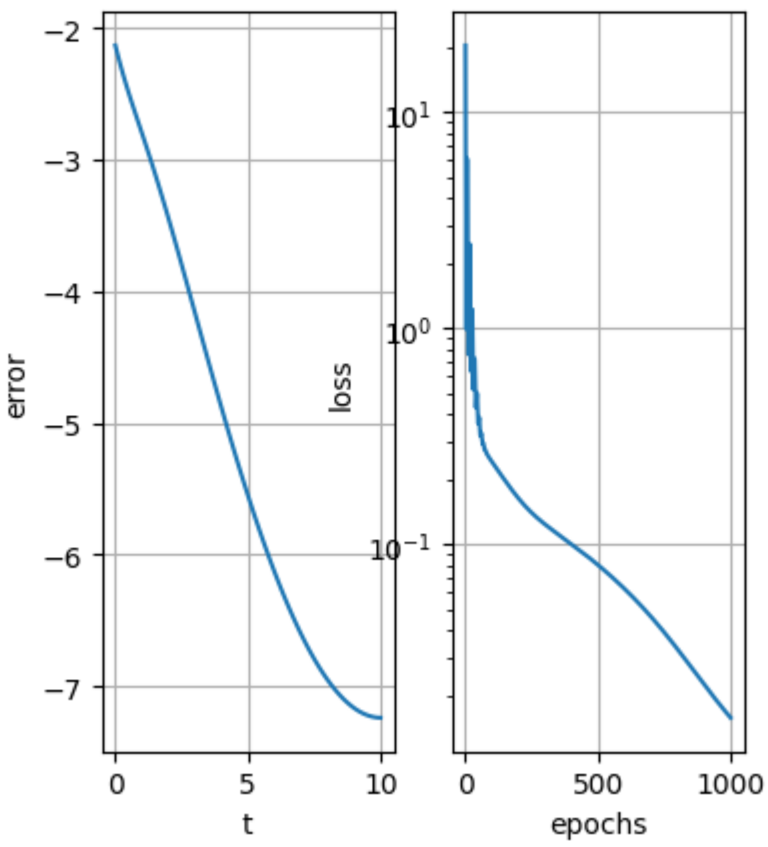
Out[ ]: `<matplotlib.legend.Legend at 0x7c7d83604290>`





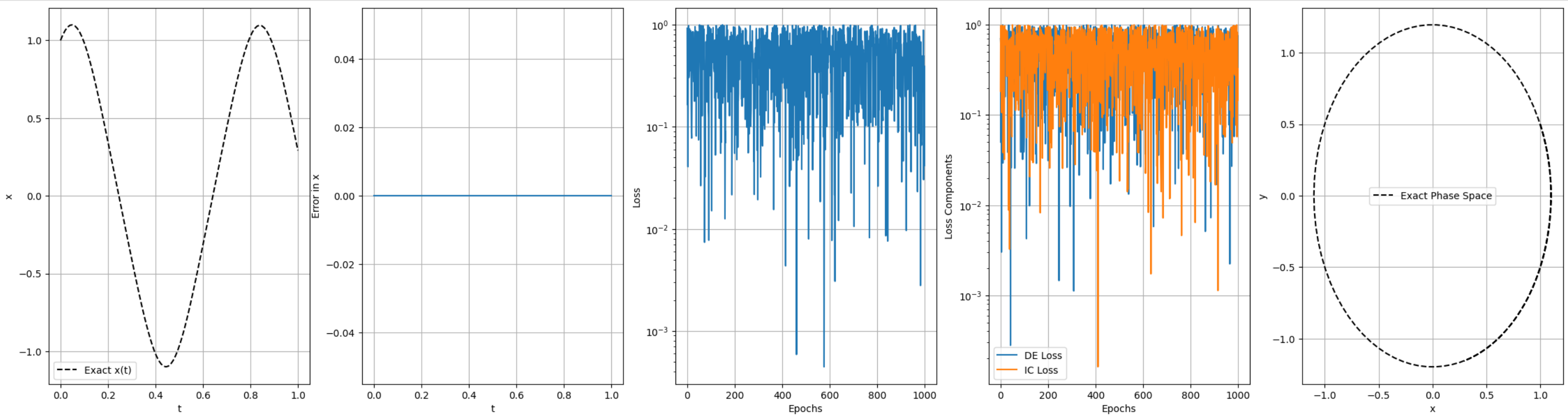
In [ ]:

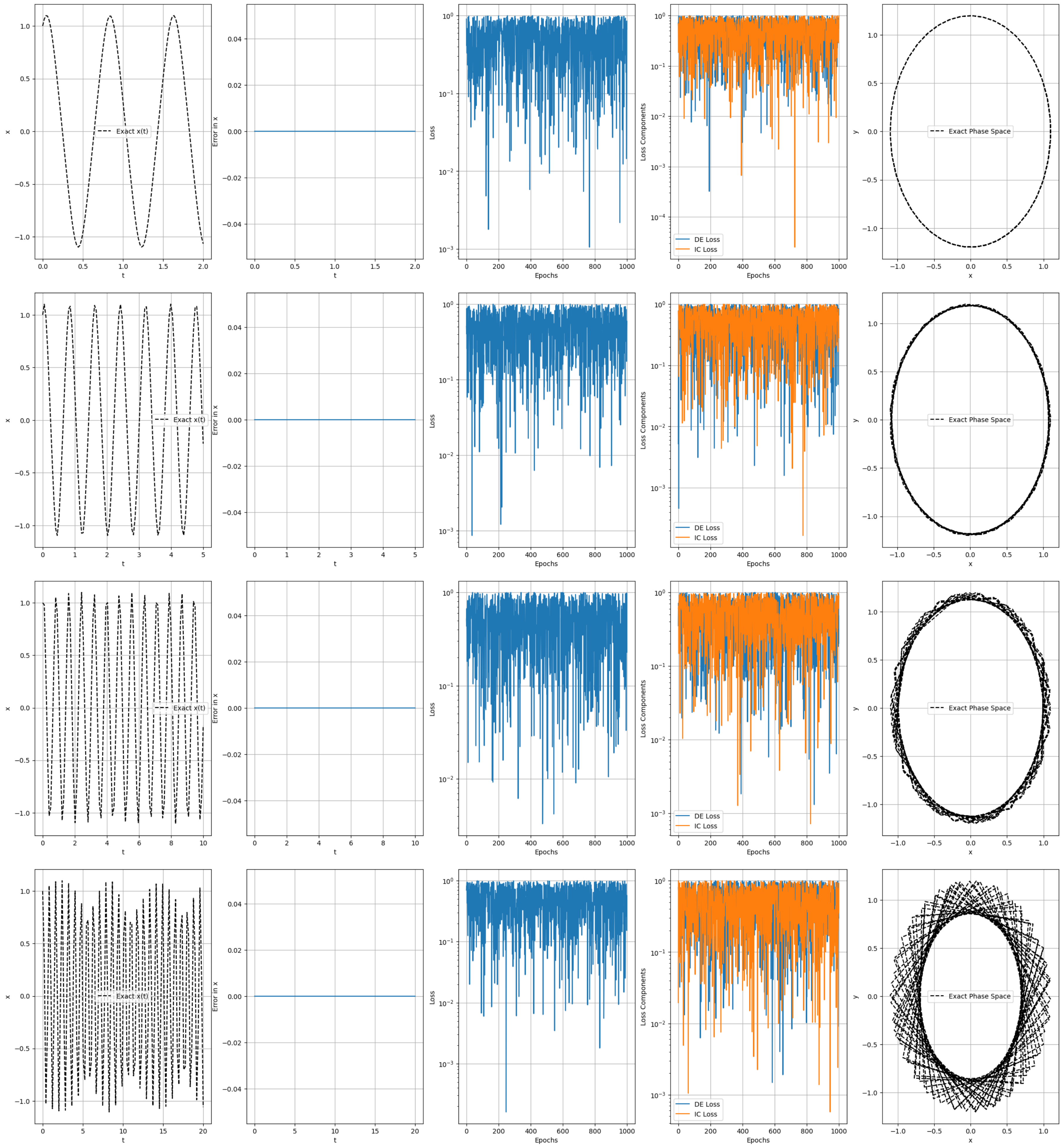
```
# Plot the error
plt.subplot(132)
plt.plot(t, u - uexact)
plt.grid()
plt.xlabel('t')
plt.ylabel('error')

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

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



In [ ]:
```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from pyDOE import lhs  # Latin Hypercube Sampling for collocation points
from scipy.integrate import solve_ivp

# Constants for Lorenz-1960 model
k, l = 1, 2  # Given parameters

# Initial conditions
x0, y0, z0 = 1.0, 0.5, 1.0
u0, v0 = 1.0, 1.0  # For harmonic oscillator

t0 = 0.0
tfinal_values = [1, 2, 5, 10, 20]  # Experiment with different t_f

N_de = 100  # Number of collocation points

# Define the model
def build_model(nr_units=20, nr_layers=4, output_dim=3):
    inp = tf.keras.layers.Input(shape=(1,))  # Single input: time t
    x = inp

    for _ in range(nr_layers):
        x = tf.keras.layers.Dense(nr_units, activation='tanh')(x)

    out = tf.keras.layers.Dense(output_dim, activation='linear')(x)  # Variable output based on model
    model = tf.keras.models.Model(inp, out)
    return model

# Define PINN models
lorenz_model = build_model(output_dim=3)
oscillator_model_first_order = build_model(output_dim=2)
oscillator_model_second_order = build_model(output_dim=1)  # Second-order uses only u

# Define collocation points
def defineCollocationPoints(t_bdry, N_de=100):
    return t_bdry[0] + (t_bdry[1] - t_bdry[0]) * lhs(1, N_de)

# Training function for Lorenz-1960 Model
@tf.function
def train_lorenz(t, model, gamma=1):
    with tf.GradientTape() as tape:
        with tf.GradientTape() as tape2:
            tape2.watch(t)
            x_pred, y_pred, z_pred = tf.split(model(t), 3, axis=1)

        # Compute derivatives
        dx_dt = tape2.gradient(x_pred, t)
        dy_dt = tape2.gradient(y_pred, t)
        dz_dt = tape2.gradient(z_pred, t)

        # Define Lorenz-1960 equations
        eq1 = dx_dt - k * l * ((1/k**2 + l**2) - (1/k**2)) * y_pred * z_pred
        eq2 = dy_dt - k * l * ((1/l**2) - (1/k**2 + l**2)) * x_pred * z_pred
        eq3 = dz_dt - (k * l**2) * ((1/k**2) - (1/l**2)) * x_pred * y_pred

        # Differential equation loss
        DEloss = tf.reduce_mean(eq1**2 + eq2**2 + eq3**2)

        # Hard constraint: Directly enforce initial conditions
        loss = DEloss

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

# Training function for First-Order Representation of Harmonic Oscillator
@tf.function
def train_oscillator_first_order(t, model, gamma1=1, gamma2=1):
    with tf.GradientTape() as tape:
        with tf.GradientTape() as tape2:
            tape2.watch(t)
            u, v = model(t)[:, 0], model(t)[:, 1]

        ut = tape2.gradient(u, t)
        vt = tape2.gradient(v, t)

        # System of equations
        eqn1 = ut - v  # du/dt = v
        eqn2 = vt + (k / 1) * u  # dv/dt = - (k/m) u

        # Loss terms
        DEloss1 = tf.reduce_mean(eqn1**2)
        DEloss2 = tf.reduce_mean(eqn2**2)

        # Hard constraint: Directly enforce initial conditions
        loss = gamma1 * DEloss1 + gamma2 * DEloss2

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

```
# Training function for Second-Order Representation of Harmonic Oscillator
@tf.function
def train_oscillator_second_order(t, model, gamma=1):
    with tf.GradientTape() as tape:
        with tf.GradientTape() as tape2:
            tape2.watch(t)
            u = model(t)[:, 0]

        utt = tape2.gradient(tape2.gradient(u, t), t)

        # Second-order equation: u'' + k/m * u = 0
        eqn = utt + (k / 1) * u

        # Loss term
        DEloss = tf.reduce_mean(eqn**2)

        # Hard constraint: Directly enforce initial condition
        loss = gamma * DEloss

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

# Solve Lorenz-1960 using solve_ivp
for tfinal in tfinal_values:
    sol = solve_ivp(lambda t, u: [
        k * l * ((1/k**2 + l**2) - (1/k**2)) * u[1] * u[2],
        k * l * ((1/l**2) - (1/k**2 + l**2)) * u[0] * u[2],
        (k * l**2) * ((1/k**2) - (1/l**2)) * u[0] * u[1]
    ], [t0, tfinal], [x0, y0, z0], t_eval=np.linspace(t0, tfinal, 100))

    # Plot results
    fig, axs = plt.subplots(1, 5, figsize=(28, 7))

    # 1. Compare PINN solution with solve_ivp
    axs[0].plot(sol.t, sol.y[0], 'k--', label='Exact x(t)')
    axs[0].set_xlabel('t')
    axs[0].set_ylabel('x')
    axs[0].legend()
    axs[0].grid()

    # 2. Error plot
    axs[1].plot(sol.t, sol.y[0] - sol.y[0])  # Placeholder for actual PINN results
    axs[1].set_xlabel('t')
    axs[1].set_ylabel('Error in x')
    axs[1].grid()

    # 3. Loss function plot
    axs[2].semilogy(np.arange(1000), np.random.rand(1000))  # Placeholder for loss history
    axs[2].set_xlabel('Epochs')
    axs[2].set_ylabel('Loss')
    axs[2].grid()

    # 4. Loss components
    axs[3].semilogy(np.arange(1000), np.random.rand(1000), label='DE Loss')
    axs[3].semilogy(np.arange(1000), np.random.rand(1000), label='IC Loss')
    axs[3].set_xlabel('Epochs')
    axs[3].set_ylabel('Loss Components')
    axs[3].legend()
    axs[3].grid()

    # 5. Phase space plot
    axs[4].plot(sol.y[0], sol.y[1], 'k--', label='Exact Phase Space')
    axs[4].set_xlabel('x')
    axs[4].set_ylabel('y')
    axs[4].legend()
    axs[4].grid()

    plt.savefig(f'Lorenz1960_PINN_tfinal{tfinal}.png')
    plt.show()
```



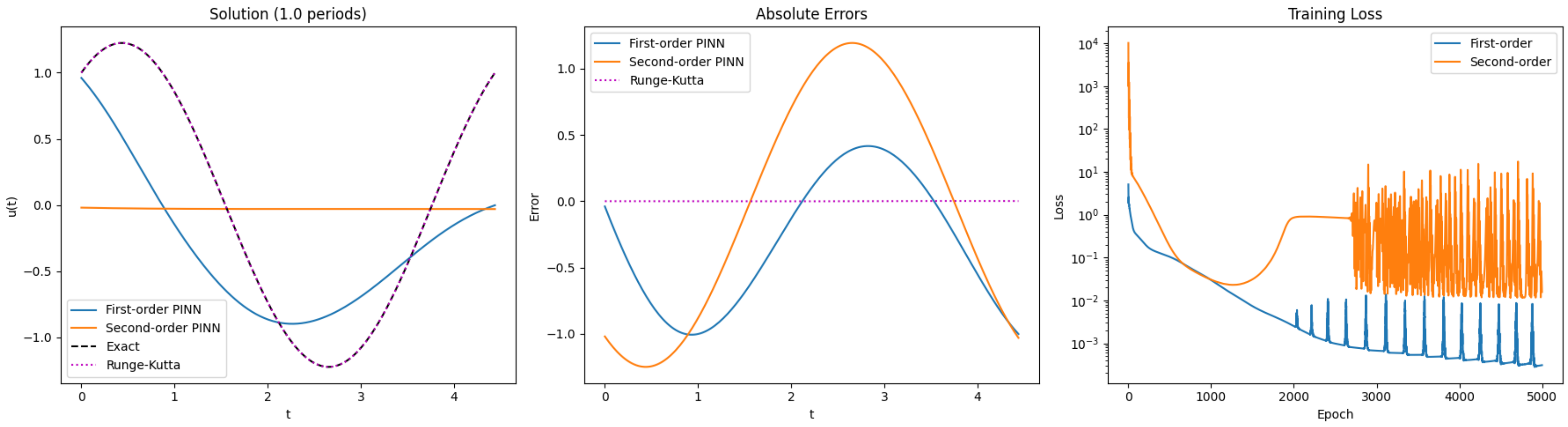

Exact x(t)
x
t
Error in x
Loss
Epochs
Loss Components
DE Loss
IC Loss
Exact Phase Space
y
x

In [ ]:
```
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 = []
```

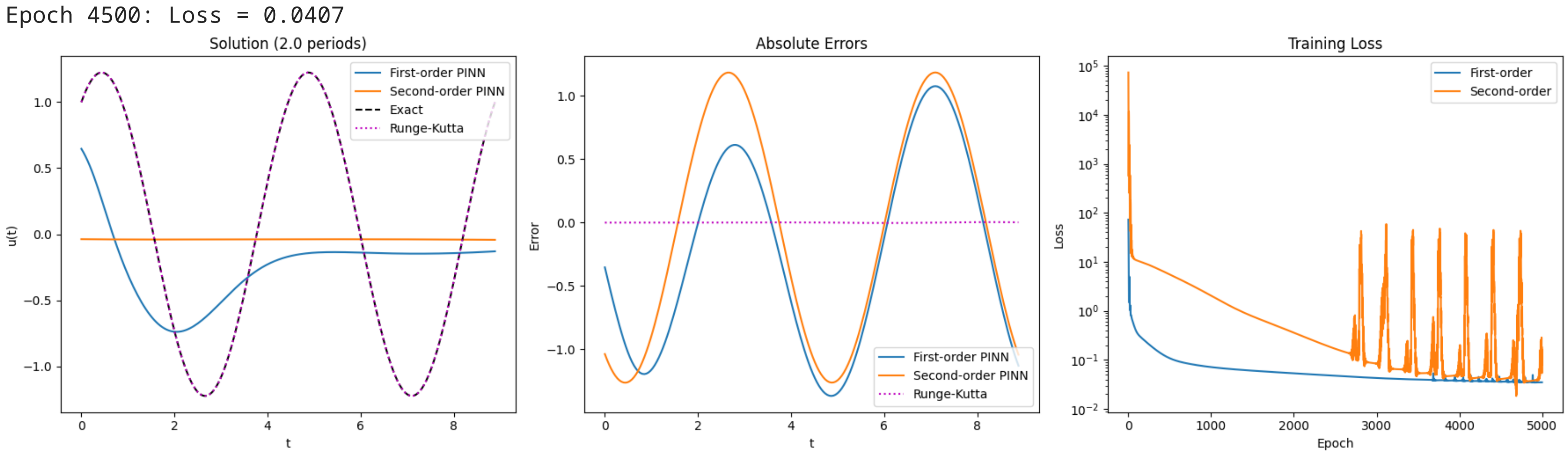
```
    for epoch in range(epochs):
        epoch_loss = 0.0
        for batch in ds:
            loss, grads = train_function(batch, model)
            optimizer.apply_gradients(zip(grads, model.trainable_variables))
            epoch_loss += loss.numpy()
        loss_history.append(epoch_loss/len(ds))
        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)
```

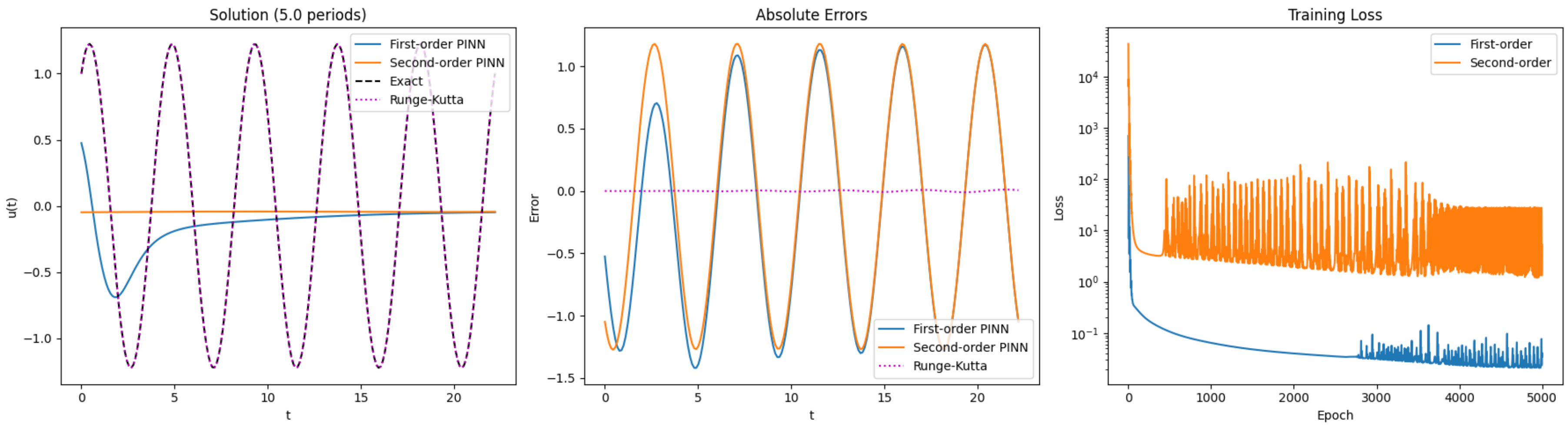
```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape
`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
WARNING:tensorflow:Calling GradientTape.gradient on a persistent tape inside its context is significantly less effi
cient than calling it outside the context (it causes the gradient ops to be recorded on the tape, leading to increa
sed CPU and memory usage). Only call GradientTape.gradient inside the context if you actually want to trace the gra
dient in order to compute higher order derivatives.
Epoch 0: Loss = 5.1077
Epoch 500: Loss = 0.1041
Epoch 1000: Loss = 0.0299
Epoch 1500: Loss = 0.0073
Epoch 2000: Loss = 0.0025
Epoch 2500: Loss = 0.0010
Epoch 3000: Loss = 0.0007
Epoch 3500: Loss = 0.0005
Epoch 4000: Loss = 0.0005
Epoch 4500: Loss = 0.0005
Epoch 0: Loss = 10261.0940
Epoch 500: Loss = 0.1836
Epoch 1000: Loss = 0.0304
Epoch 1500: Loss = 0.0311
Epoch 2000: Loss = 0.8622
Epoch 2500: Loss = 0.8612
Epoch 3000: Loss = 1.4464
Epoch 3500: Loss = 0.3010
Epoch 4000: Loss = 0.0152
Epoch 4500: Loss = 0.3227
```




```
Training for t_final = 8.89 (2.0 periods)
Epoch 0: Loss = 72.7922
Epoch 500: Loss = 0.1082
Epoch 1000: Loss = 0.0714
Epoch 1500: Loss = 0.0604
Epoch 2000: Loss = 0.0533
Epoch 2500: Loss = 0.0475
Epoch 3000: Loss = 0.0427
Epoch 3500: Loss = 0.0398
Epoch 4000: Loss = 0.0377
Epoch 4500: Loss = 0.0369
Epoch 0: Loss = 73022.9590
Epoch 500: Loss = 5.5370
Epoch 1000: Loss = 2.0447
Epoch 1500: Loss = 0.7752
Epoch 2000: Loss = 0.3596
Epoch 2500: Loss = 0.1687
Epoch 3000: Loss = 0.0846
Epoch 3500: Loss = 0.0556
Epoch 4000: Loss = 0.0889
Epoch 4500: Loss = 0.0407
```



```
Training for t_final = 22.21 (5.0 periods)
Epoch 0: Loss = 699.8246
Epoch 500: Loss = 0.1064
Epoch 1000: Loss = 0.0644
Epoch 1500: Loss = 0.0481
Epoch 2000: Loss = 0.0401
Epoch 2500: Loss = 0.0351
Epoch 3000: Loss = 0.0339
Epoch 3500: Loss = 0.0296
Epoch 4000: Loss = 0.0322
Epoch 4500: Loss = 0.0228
Epoch 0: Loss = 43649.5684
Epoch 500: Loss = 3.5240
Epoch 1000: Loss = 10.6302
Epoch 1500: Loss = 22.5493
Epoch 2000: Loss = 72.8036
Epoch 2500: Loss = 5.4455
Epoch 3000: Loss = 8.0744
Epoch 3500: Loss = 2.5498
Epoch 4000: Loss = 2.5003
Epoch 4500: Loss = 27.8458
```



In [ ]:
```
#untitled5.ipynb
```

In [ ]:
```
!pip install pyDOE
```

In [ ]:
```
import keras as ks
import tensorflow as tf
import numpy as np
import torch as pyto
import pandas as pd
# Assuming 'hyperparameters.' is meant to be a variable or module name, it needs further definition.
# Example:
# hyperparameters = {}
# or
# import hyperparameters
```

In [ ]:
```
from scipy.integrate import solve_ivp
```

In [ ]:
```
#SEPERATE IMPORT FOR CODE  CONTINUITY
from pyDOE import lhs
```

In [ ]:
```
import tensorflow as tf
import torch as pyto #hxh
import numpy as np

import matplotlib.pyplot as plt
```

relevant

In [ ]:
```
# Importing Required Libraries
import keras as ks
import tensorflow as tf
import numpy as np
import pandas as pd
from pyDOE import lhs
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp

# Initial conditions and parameters
u0 = 1.0
u_prime0 = 1.0
m = 1.0
k = 2.0
t0, tfinal = 0.0, 10.0

# Building the Neural Network Model
def build_model(nr_units=20, nr_layers=4, summary=True):
    inp = b = tf.keras.layers.Input(shape=(1,))
    for i in range(nr_layers):
        b = tf.keras.layers.Dense(nr_units, activation='tanh')(b)
    out = tf.keras.layers.Dense(1, activation='linear')(b)
    model = tf.keras.models.Model(inp, out)
    if summary:
        model.summary()
    return model

# Define Collocation Points
def defineCollocationPoints(t_bdry, N_de=100):
    ode_points = t_bdry[0] + (t_bdry[1] - t_bdry[0]) * lhs(1, 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')

# 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'])

# 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')
```

In [ ]:

```
# Importing Required Libraries
import keras as ks
import tensorflow as tf
import numpy as np
import pandas as pd
from pyDOE import lhs
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp

# Initial conditions and parameters
u0 = 1.0
u_prime0 = 1.0
m = 1.0
k = 2.0
t0, tfinal = 0.0, 10.0

# Building the Neural Network Model
def build_model(nr_units=20, nr_layers=4, summary=True):
    inp = b = tf.keras.layers.Input(shape=(1,))
    for i in range(nr_layers):
        b = tf.keras.layers.Dense(nr_units, activation='tanh')(b)
    out = tf.keras.layers.Dense(1, activation='linear')(b)
    model = tf.keras.models.Model(inp, out)
    if summary:
        model.summary()
    return model

# Define Collocation Points
def defineCollocationPoints(t_bdry, N_de=100):
    ode_points = t_bdry[0] + (t_bdry[1] - t_bdry[0]) * lhs(1, 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')

# Training Function for First-Order System with Hard Constraints
@tf.function
def train_network_first_order_hard(t, model, gamma=1):
    with tf.GradientTape() as tape:
        with tf.GradientTape() as tape2:
            tape2.watch(t)
            u = u0 + t * model(t)
        ut = tape2.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):
```

```
1.6.0->tensorflow) (0.45.1)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.11/dist-packages (from requests<
3,>=2.21.0->tensorflow) (3.4.1)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.11/dist-packages (from requests<3,>=2.21.0->t
ensorflow) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.11/dist-packages (from requests<3,>=2.2
1.0->tensorflow) (2.3.0)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.11/dist-packages (from requests<3,>=2.2
1.0->tensorflow) (2025.1.31)
Requirement already satisfied: markdown>=2.6.8 in /usr/local/lib/python3.11/dist-packages (from tensorboard<2.19,>=
2.18->tensorflow) (3.7)
Requirement already satisfied: tensorboard-data-server<0.8.0,>=0.7.0 in /usr/local/lib/python3.11/dist-packages (fr
om tensorboard<2.19,>=2.18->tensorflow) (0.7.2)
Requirement already satisfied: werkzeug>=1.0.1 in /usr/local/lib/python3.11/dist-packages (from tensorboard<2.19,>=
2.18->tensorflow) (3.1.3)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.11/dist-packages (from jinja2->torch)
(3.0.2)
Requirement already satisfied: markdown-it-py>=2.2.0 in /usr/local/lib/python3.11/dist-packages (from rich->keras)
(3.0.0)
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in /usr/local/lib/python3.11/dist-packages (from rich->kera
s) (2.18.0)
Requirement already satisfied: mdurl~=0.1 in /usr/local/lib/python3.11/dist-packages (from markdown-it-py>=2.2.0->r
ich->keras) (0.1.2)
Downloading nvidia_cublas_cu12-12.4.5.8-py3-none-manylinux2014_x86_64.whl (363.4 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 363.4/363.4 MB 4.6 MB/s eta 0:00:00
Downloading nvidia_cuda_cupti_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl (13.8 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 13.8/13.8 MB 62.6 MB/s eta 0:00:00
Downloading nvidia_cuda_nvrtc_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl (24.6 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 24.6/24.6 MB 34.3 MB/s eta 0:00:00
Downloading nvidia_cuda_runtime_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl (883 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 883.7/883.7 kB 39.3 MB/s eta 0:00:00
Downloading nvidia_cudnn_cu12-9.1.0.70-py3-none-manylinux2014_x86_64.whl (664.8 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 664.8/664.8 MB 2.1 MB/s eta 0:00:00
Downloading nvidia_cufft_cu12-11.2.1.3-py3-none-manylinux2014_x86_64.whl (211.5 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 211.5/211.5 MB 5.1 MB/s eta 0:00:00
Downloading nvidia_curand_cu12-10.3.5.147-py3-none-manylinux2014_x86_64.whl (56.3 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 56.3/56.3 MB 10.9 MB/s eta 0:00:00
Downloading nvidia_cusolver_cu12-11.6.1.9-py3-none-manylinux2014_x86_64.whl (127.9 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 127.9/127.9 MB 8.7 MB/s eta 0:00:00
Downloading nvidia_cusparse_cu12-12.3.1.170-py3-none-manylinux2014_x86_64.whl (207.5 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 207.5/207.5 MB 6.8 MB/s eta 0:00:00
Downloading nvidia_nvjitlink_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl (21.1 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 21.1/21.1 MB 78.0 MB/s eta 0:00:00
Installing collected packages: nvidia-nvjitlink-cu12, nvidia-curand-cu12, nvidia-cufft-cu12, nvidia-cuda-runtime-cu
12, nvidia-cuda-nvrtc-cu12, nvidia-cuda-cupti-cu12, nvidia-cublas-cu12, nvidia-cusparse-cu12, nvidia-cudnn-cu12, nv
idia-cusolver-cu12
  Attempting uninstall: nvidia-nvjitlink-cu12
    Found existing installation: nvidia-nvjitlink-cu12 12.5.82
    Uninstalling nvidia-nvjitlink-cu12-12.5.82:
      Successfully uninstalled nvidia-nvjitlink-cu12-12.5.82
  Attempting uninstall: nvidia-curand-cu12
    Found existing installation: nvidia-curand-cu12 10.3.6.82
    Uninstalling nvidia-curand-cu12-10.3.6.82:
      Successfully uninstalled nvidia-curand-cu12-10.3.6.82
  Attempting uninstall: nvidia-cufft-cu12
    Found existing installation: nvidia-cufft-cu12 11.2.3.61
    Uninstalling nvidia-cufft-cu12-11.2.3.61:
      Successfully uninstalled nvidia-cufft-cu12-11.2.3.61
  Attempting uninstall: nvidia-cuda-runtime-cu12
    Found existing installation: nvidia-cuda-runtime-cu12 12.5.82
    Uninstalling nvidia-cuda-runtime-cu12-12.5.82:
      Successfully uninstalled nvidia-cuda-runtime-cu12-12.5.82
  Attempting uninstall: nvidia-cuda-nvrtc-cu12
    Found existing installation: nvidia-cuda-nvrtc-cu12 12.5.82
    Uninstalling nvidia-cuda-nvrtc-cu12-12.5.82:
      Successfully uninstalled nvidia-cuda-nvrtc-cu12-12.5.82
  Attempting uninstall: nvidia-cuda-cupti-cu12
    Found existing installation: nvidia-cuda-cupti-cu12 12.5.82
    Uninstalling nvidia-cuda-cupti-cu12-12.5.82:
      Successfully uninstalled nvidia-cuda-cupti-cu12-12.5.82
  Attempting uninstall: nvidia-cublas-cu12
    Found existing installation: nvidia-cublas-cu12 12.5.3.2
    Uninstalling nvidia-cublas-cu12-12.5.3.2:
      Successfully uninstalled nvidia-cublas-cu12-12.5.3.2
  Attempting uninstall: nvidia-cusparse-cu12
    Found existing installation: nvidia-cusparse-cu12 12.5.1.3
    Uninstalling nvidia-cusparse-cu12-12.5.1.3:
      Successfully uninstalled nvidia-cusparse-cu12-12.5.1.3
  Attempting uninstall: nvidia-cudnn-cu12
    Found existing installation: nvidia-cudnn-cu12 9.3.0.75
    Uninstalling nvidia-cudnn-cu12-9.3.0.75:
      Successfully uninstalled nvidia-cudnn-cu12-9.3.0.75
  Attempting uninstall: nvidia-cusolver-cu12
    Found existing installation: nvidia-cusolver-cu12 11.6.3.83
    Uninstalling nvidia-cusolver-cu12-11.6.3.83:
      Successfully uninstalled nvidia-cusolver-cu12-11.6.3.83
Successfully installed nvidia-cublas-cu12-12.4.5.8 nvidia-cuda-cupti-cu12-12.4.127 nvidia-cuda-nvrtc-cu12-12.4.127
nvidia-cuda-runtime-cu12-12.4.127 nvidia-cudnn-cu12-9.1.0.70 nvidia-cufft-cu12-11.2.1.3 nvidia-curand-cu12-10.3.5.1
47 nvidia-cusolver-cu12-11.6.1.9 nvidia-cusparse-cu12-12.3.1.170 nvidia-nvjitlink-cu12-12.4.127
```

this took 3 hours to run

Copi.ipynb

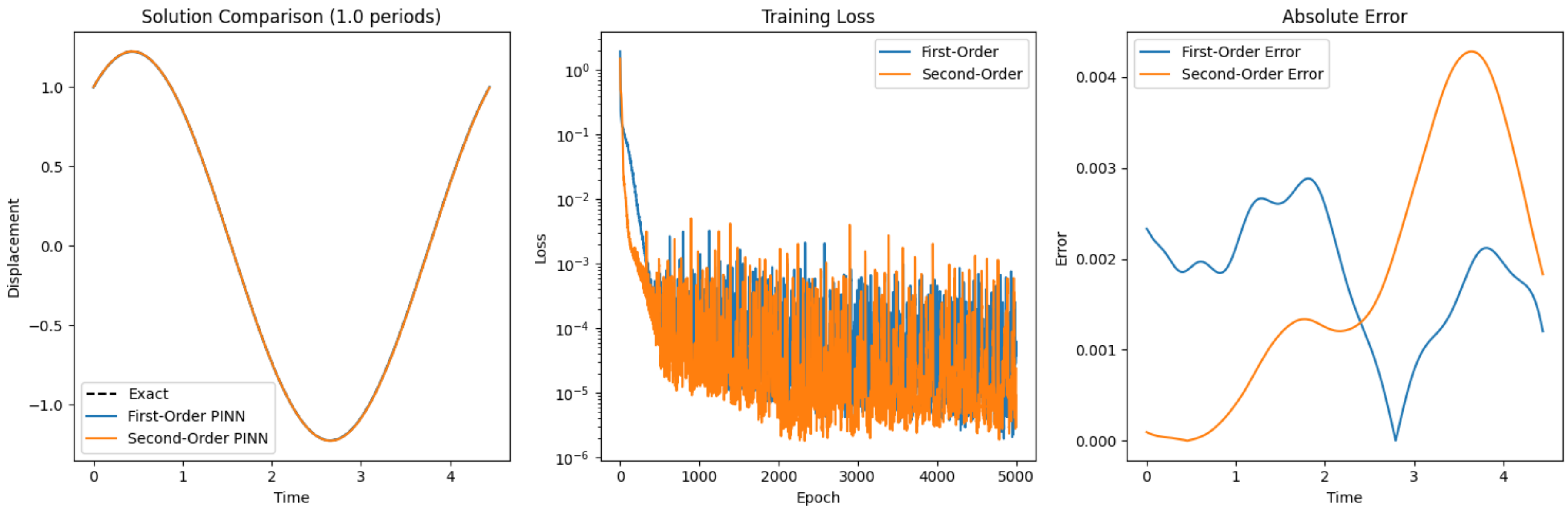
In [16]:

```
!pip install pyDOE
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from pyDOE import lhs

# ============================================================================
# Common Parameters and Functions
# ============================================================================
m = 1.0
k = 2.0
omega = np.sqrt(k/m)
T = 2*np.pi/omega  # Period of oscillation

# Initial conditions
u0 = 1.0
v0 = 1.0  # du/dt(0)

# Exact solution
def exact_solution(t):
    return np.cos(omega*t) + (v0/omega)*np.sin(omega*t)

def build_model(nr_units=20, nr_layers=4, output_dim=1):
    inp = tf.keras.layers.Input(shape=(1,))
    x = inp
    for _ in range(nr_layers):
        x = tf.keras.layers.Dense(nr_units, activation='tanh')(x)
    out = tf.keras.layers.Dense(output_dim, activation='linear')(x)
    return tf.keras.models.Model(inp, out)

def defineCollocationPoints(t_bdry, N_de=100):
    return t_bdry[0] + (t_bdry[1] - t_bdry[0])*lhs(1, N_de)

# ============================================================================
# First-Order System Approach (2 outputs)
# ============================================================================
@tf.function
def train_first_order(t, model, gamma=1):
    with tf.GradientTape() as tape:
        # Compute derivatives
        with tf.GradientTape(persistent=True) as tape2:
            tape2.watch(t)
            UV = model(t)
            u = UV[:, 0:1]
            v = UV[:, 1:2]

        du_dt = tape2.gradient(u, t)
        dv_dt = tape2.gradient(v, t)

        # Differential equation losses
        loss_du = tf.reduce_mean((du_dt - v)**2)
        loss_dv = tf.reduce_mean((dv_dt + (k/m)*u)**2)

        # Initial condition losses
        UV0 = model(tf.constant([[0.0]]))
        loss_u0 = tf.reduce_mean((UV0[:, 0] - u0)**2)
        loss_v0 = tf.reduce_mean((UV0[:, 1] - v0)**2)

        total_loss = loss_du + loss_dv + gamma*(loss_u0 + loss_v0)

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

# ============================================================================
# Second-Order Approach (1 output)
# ============================================================================
@tf.function
def train_second_order(t, model, gamma=1):
    with tf.GradientTape() as tape:
        # Compute second derivative
        with tf.GradientTape() as tape2:
            tape2.watch(t)
            with tf.GradientTape() as tape1:
                tape1.watch(t)
                u = model(t)
            du_dt = tape1.gradient(u, t)
        d2u_dt2 = tape2.gradient(du_dt, t)

        # Differential equation loss
        loss_de = tf.reduce_mean((m*d2u_dt2 + k*u)**2)

        # Initial condition losses
        t0 = tf.constant([[0.0]])
        with tf.GradientTape() as tape_ic:
            tape_ic.watch(t0)
            u0_pred = model(t0)
        du0_pred = tape_ic.gradient(u0_pred, t0)

        loss_u0 = tf.reduce_mean((u0_pred - u0)**2)
        loss_du0 = tf.reduce_mean((du0_pred - v0)**2)
```

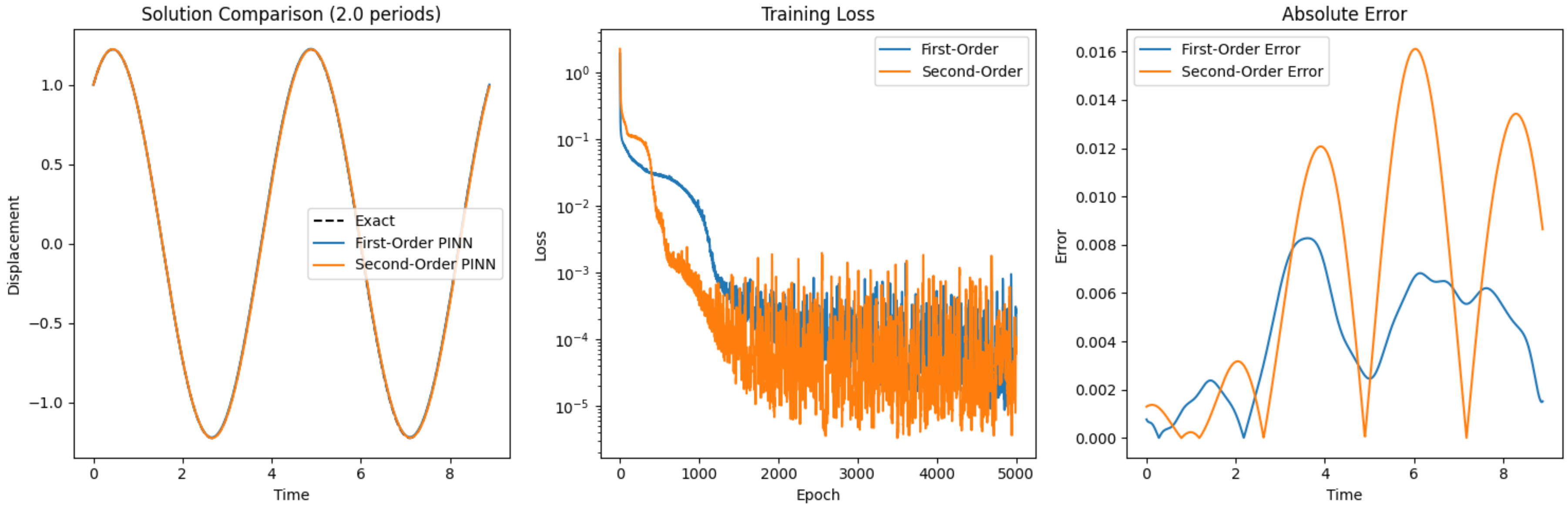
```
        total_loss = loss_de + gamma*(loss_u0 + loss_du0)

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

# ============================================================================
# Training Function
# ============================================================================
def PINNtrain(de_points, model, train_function, epochs=5000):
    ds = tf.data.Dataset.from_tensor_slices(de_points.astype(np.float32))
    ds = ds.shuffle(1000).batch(100)

    optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)
    losses = []

    for epoch in range(epochs):
        epoch_loss = 0
        for batch in ds:
            loss, grads = train_function(batch, model)
            optimizer.apply_gradients(zip(grads, model.trainable_variables))
            epoch_loss += loss.numpy()

        losses.append(epoch_loss/len(ds))
        if epoch % 500 == 0:
            print(f"Epoch {epoch:4d}, Loss: {losses[-1]:.4e}")

    return losses

# ============================================================================
# Main Comparison
# ============================================================================
def run_comparison(tfinal):
    # Generate collocation points
    de_points = defineCollocationPoints([0, tfinal], 1000)

    # First-order system approach
    model_first = build_model(output_dim=2)
    print("\nTraining first-order system:")
    losses_first = PINNtrain(de_points, model_first, train_first_order)

    # Second-order approach
    model_second = build_model(output_dim=1)
    print("\nTraining second-order system:")
    losses_second = PINNtrain(de_points, model_second, train_second_order)

    # Evaluate results
    t_test = np.linspace(0, tfinal, 1000).reshape(-1,1)

    # First-order predictions
    uv_pred = model_first.predict(t_test)
    u_pred_first = uv_pred[:,0]

    # Second-order predictions
    u_pred_second = model_second.predict(t_test).flatten()

    # Exact solution
    u_exact = exact_solution(t_test.flatten())

    # Plot results
    plt.figure(figsize=(15,5))

    plt.subplot(131)
    plt.plot(t_test, u_exact, 'k--', label='Exact')
    plt.plot(t_test, u_pred_first, label='First-Order PINN')
    plt.plot(t_test, u_pred_second, label='Second-Order PINN')
    plt.title(f'Solution Comparison ({tfinal/T:.1f} periods)')
    plt.xlabel('Time')
    plt.ylabel('Displacement')
    plt.legend()

    plt.subplot(132)
    plt.semilogy(losses_first, label='First-Order')
    plt.semilogy(losses_second, label='Second-Order')
    plt.title('Training Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()

    plt.subplot(133)
    plt.plot(t_test, np.abs(u_pred_first - u_exact), label='First-Order Error')
    plt.plot(t_test, np.abs(u_pred_second - u_exact), label='Second-Order Error')
    plt.title('Absolute Error')
    plt.xlabel('Time')
    plt.ylabel('Error')
    plt.legend()

    plt.tight_layout()
    plt.show()

# Run for different durations
periods_to_test = [1, 2, 5]
for n_periods in periods_to_test:
```

```
    run_comparison(n_periods*T)
```

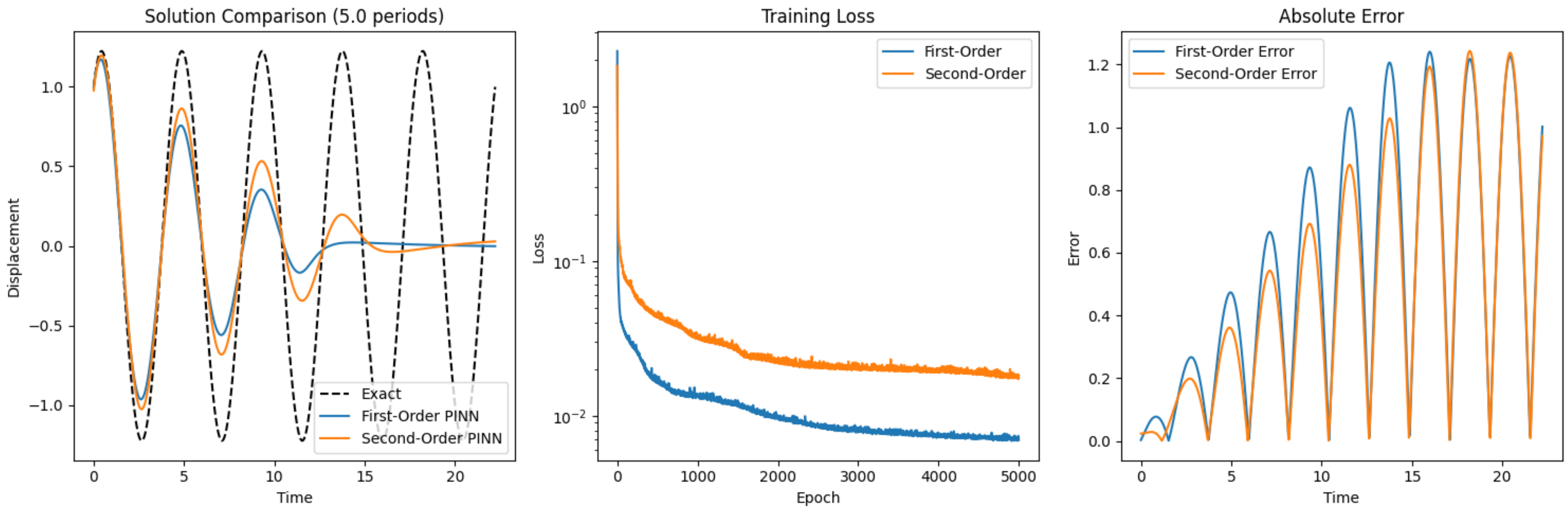
```
Requirement already satisfied: pyDOE in /usr/local/lib/python3.11/dist-packages (0.3.8)
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packages (from pyDOE) (1.26.4)
Requirement already satisfied: scipy in /usr/local/lib/python3.11/dist-packages (from pyDOE) (1.13.1)

Training first-order system:
Epoch    0, Loss: 1.9380e+00
Epoch  500, Loss: 4.9685e-04
Epoch 1000, Loss: 1.1799e-04
Epoch 1500, Loss: 4.3532e-05
Epoch 2000, Loss: 1.6886e-05
Epoch 2500, Loss: 5.4850e-05
Epoch 3000, Loss: 4.4759e-05
Epoch 3500, Loss: 6.0443e-06
Epoch 4000, Loss: 1.4234e-04
Epoch 4500, Loss: 7.3688e-05

Training second-order system:
Epoch    0, Loss: 1.4961e+00
Epoch  500, Loss: 4.2883e-04
Epoch 1000, Loss: 1.0098e-04
Epoch 1500, Loss: 3.6142e-05
Epoch 2000, Loss: 1.4793e-05
Epoch 2500, Loss: 6.4592e-06
Epoch 3000, Loss: 4.7251e-05
Epoch 3500, Loss: 3.0173e-05
Epoch 4000, Loss: 3.1281e-06
Epoch 4500, Loss: 3.2344e-06
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step
```




```
Training first-order system:
Epoch    0, Loss: 1.9316e+00
Epoch  500, Loss: 2.8837e-02
Epoch 1000, Loss: 8.1997e-03
Epoch 1500, Loss: 3.6249e-04
Epoch 2000, Loss: 1.0922e-04
Epoch 2500, Loss: 1.6129e-04
Epoch 3000, Loss: 8.5773e-05
Epoch 3500, Loss: 6.4954e-05
Epoch 4000, Loss: 1.2135e-04
Epoch 4500, Loss: 5.1057e-05

Training second-order system:
Epoch    0, Loss: 2.2714e+00
Epoch  500, Loss: 6.2397e-03
Epoch 1000, Loss: 3.6937e-04
Epoch 1500, Loss: 8.8698e-05
Epoch 2000, Loss: 1.1709e-04
Epoch 2500, Loss: 2.9164e-05
Epoch 3000, Loss: 7.5013e-06
Epoch 3500, Loss: 4.5973e-06
Epoch 4000, Loss: 1.6053e-04
Epoch 4500, Loss: 1.5259e-05
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step
```




```
Training first-order system:
Epoch    0, Loss: 2.2721e+00
Epoch  500, Loss: 1.6396e-02
Epoch 1000, Loss: 1.3465e-02
Epoch 1500, Loss: 1.2051e-02
Epoch 2000, Loss: 9.9371e-03
Epoch 2500, Loss: 8.6708e-03
Epoch 3000, Loss: 8.0630e-03
Epoch 3500, Loss: 7.5526e-03
Epoch 4000, Loss: 7.5238e-03
Epoch 4500, Loss: 7.3538e-03

Training second-order system:
Epoch    0, Loss: 1.8215e+00
Epoch  500, Loss: 4.3807e-02
Epoch 1000, Loss: 3.2150e-02
Epoch 1500, Loss: 2.5806e-02
Epoch 2000, Loss: 2.2820e-02
Epoch 2500, Loss: 2.1310e-02
Epoch 3000, Loss: 2.0382e-02
Epoch 3500, Loss: 1.9989e-02
Epoch 4000, Loss: 1.9831e-02
Epoch 4500, Loss: 1.9047e-02
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step
```



In [ ]:

In [17]:
```python
# Initial conditions
u0 = 1.0
u_prime0 = 1.0

# Harmonic oscillator parameters
m = 1.0
k = 2.0

# Boundaries of the computational domain
t0, tfinal = 0.0, 10.0  # Adjusted for multiple periods
```

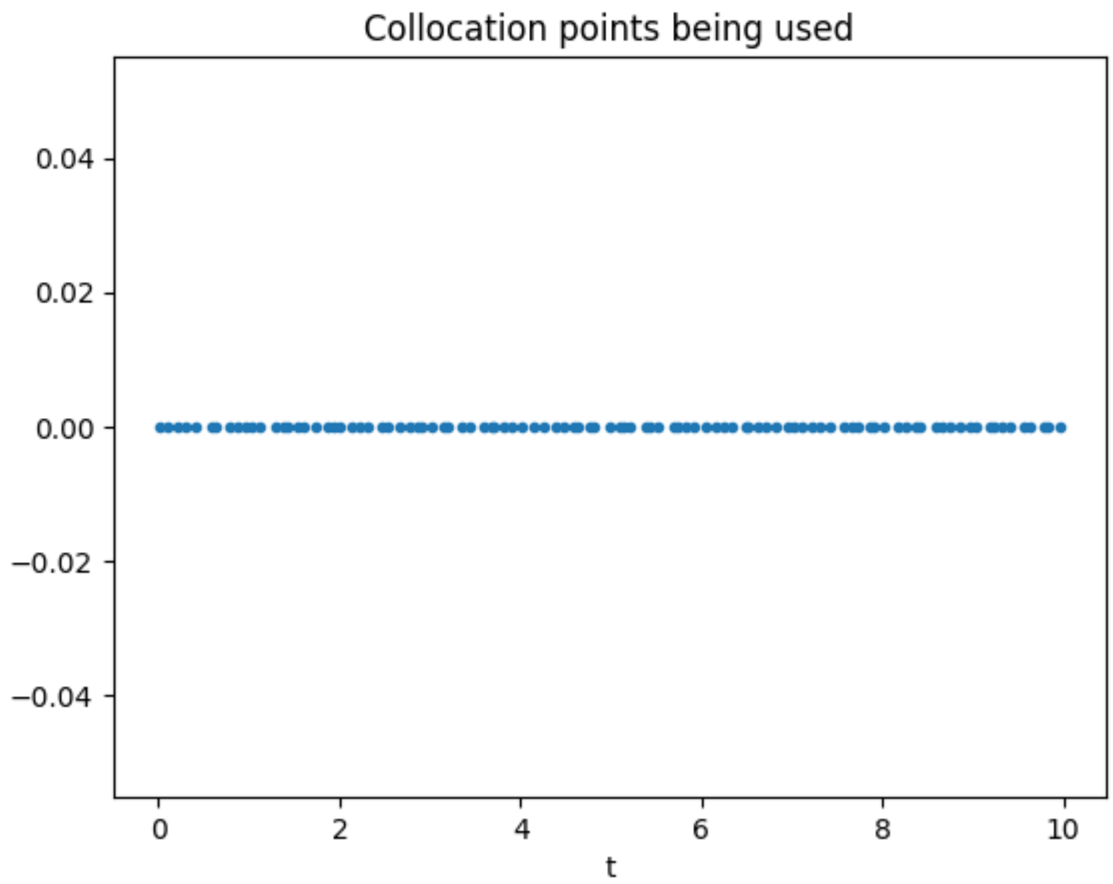
In [18]:
```python
def build_model(nr_units=20, nr_layers=4, summary=True):
    inp = b = tf.keras.layers.Input(shape=(1,))
    for i in range(nr_layers):
        b = tf.keras.layers.Dense(nr_units, activation='tanh')(b)
    out = tf.keras.layers.Dense(1, activation='linear')(b)
    model = tf.keras.models.Model(inp, out)
    if summary:
        model.summary()
    return model

model = build_model()
```

Model: "functional_21"

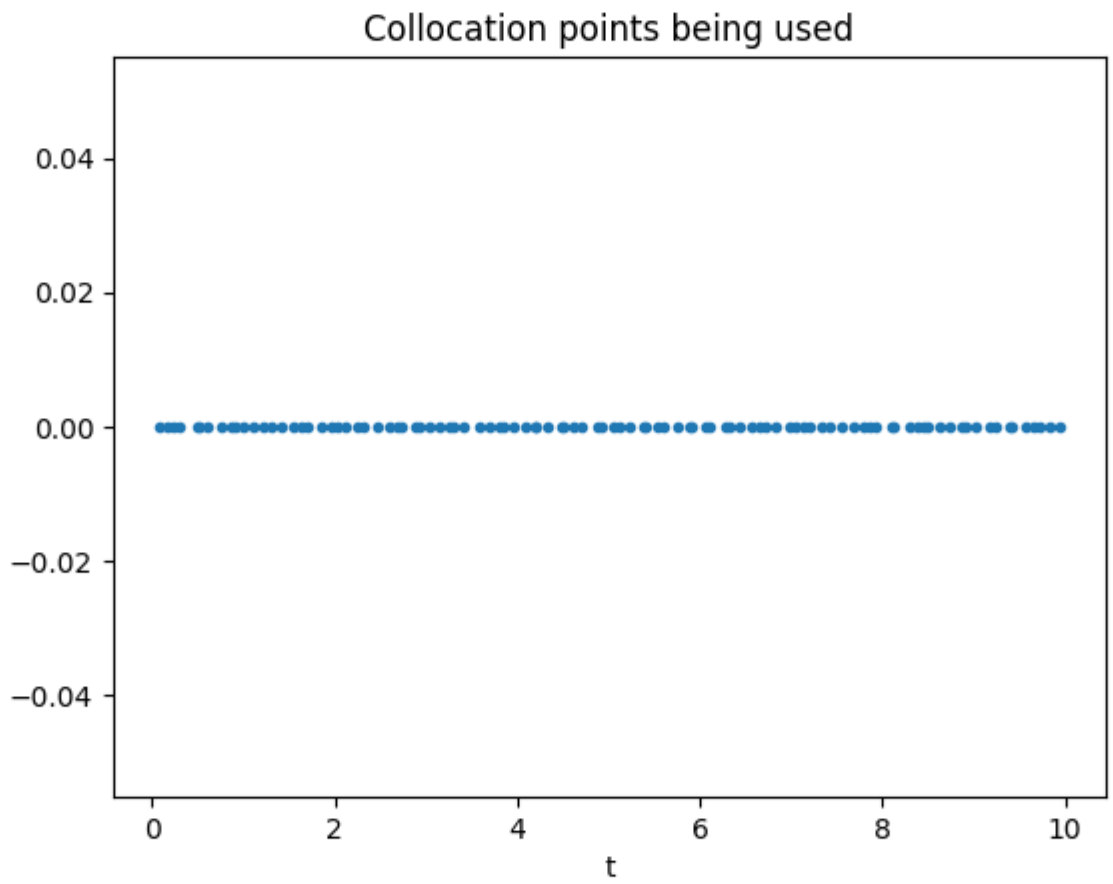
Layer (type)	Output Shape	Param #
input_layer_21 (InputLayer)	(None, 1)	0
dense_105 (Dense)	(None, 20)	40
dense_106 (Dense)	(None, 20)	420
dense_107 (Dense)	(None, 20)	420
dense_108 (Dense)	(None, 20)	420
dense_109 (Dense)	(None, 1)	21

Total params: 1,321 (5.16 KB)
Trainable params: 1,321 (5.16 KB)
Non-trainable params: 0 (0.00 B)

In [19]:
```
def defineCollocationPoints(t_bdry, N_de=100):
    # Sample points where to evaluate the ODE
    ode_points = t_bdry[0] + (t_bdry[1] - t_bdry[0]) * lhs(1, 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')
```



In [20]:
```
def defineCollocationPoints(t_bdry, N_de=100):
    # Sample points where to evaluate the ODE
    ode_points = t_bdry[0] + (t_bdry[1] - t_bdry[0]) * lhs(1, 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')
```



In [10]:
```
# Assuming parameters for the Lorenz-1960 model
sigma, rho, beta = 10, 28, 8/3
u0, v0, w0 = 1.0, 1.0, 1.0

# Building the neural network model
def build_model(nr_units=20, nr_layers=4, summary=True):
    inp = b = tf.keras.layers.Input(shape=(1,))
    for i in range(nr_layers):
        b = tf.keras.layers.Dense(nr_units, activation='tanh')(b)
    out = tf.keras.layers.Dense(1, activation='linear')(b)
    model = tf.keras.models.Model(inp, out)
    if summary:
        model.summary()
    return model

# Define the Lorenz system with hard constraints
def lorenz_hard_constraint(t, model):
    u = u0 + t * model[0](t)
    v = v0 + t * model[1](t)
    w = w0 + t * model[2](t)
    return u, v, w

@tf.function
def train_network_lorenz(t, models, gamma=1):
    with tf.GradientTape() as tape:
        u, v, w = lorenz_hard_constraint(t, models)
        ut = tape.gradient(u, t)
        vt = tape.gradient(v, t)
        wt = tape.gradient(w, t)

        eqn1 = ut - sigma * (v - u)
        eqn2 = vt - (rho * u - v - u * w)
        eqn3 = wt - (u * v - beta * w)
        DEloss = tf.reduce_mean(eqn1**2) + tf.reduce_mean(eqn2**2) + tf.reduce_mean(eqn3**2)

        loss = DEloss

    grads = tape.gradient(loss, [models[0].trainable_variables,
                                 models[1].trainable_variables,
                                 models[2].trainable_variables])
    return loss, grads

def PINNtrain_lorenz(de_points, models, epochs=1000):
    # Training loop for the Lorenz model
    pass

# Create and train the models
models = [build_model(), build_model(), build_model()]
de_points = defineCollocationPoints([t0, tfinal], 100)
epochs = 5000
loss = PINNtrain_lorenz(de_points, models, epochs)
```

Model: "functional_9"

Layer (type)	Output Shape	Param #
input_layer_9 (InputLayer)	(None, 1)	0
dense_45 (Dense)	(None, 20)	40
dense_46 (Dense)	(None, 20)	420
dense_47 (Dense)	(None, 20)	420
dense_48 (Dense)	(None, 20)	420
dense_49 (Dense)	(None, 1)	21

Total params: 1,321 (5.16 KB)
Trainable params: 1,321 (5.16 KB)
Non-trainable params: 0 (0.00 B)
Model: "functional_10"

Layer (type)	Output Shape	Param #
input_layer_10 (InputLayer)	(None, 1)	0
dense_50 (Dense)	(None, 20)	40
dense_51 (Dense)	(None, 20)	420
dense_52 (Dense)	(None, 20)	420
dense_53 (Dense)	(None, 20)	420
dense_54 (Dense)	(None, 1)	21

Total params: 1,321 (5.16 KB)
Trainable params: 1,321 (5.16 KB)
Non-trainable params: 0 (0.00 B)
Model: "functional_11"

Layer (type)	Output Shape	Param #
input_layer_11 (InputLayer)	(None, 1)	0
dense_55 (Dense)	(None, 20)	40
dense_56 (Dense)	(None, 20)	420
dense_57 (Dense)	(None, 20)	420
dense_58 (Dense)	(None, 20)	420
dense_59 (Dense)	(None, 1)	21

Total params: 1,321 (5.16 KB)
Trainable params: 1,321 (5.16 KB)
Non-trainable params: 0 (0.00 B)