Physics-Informed Neural Networks (PINNs) Implementation

This notebook implements Physics-Informed Neural Networks (PINNs) for solving different types of differential equations:

1. Lorenz-1960 System
2. Harmonic Oscillator (1st and 2nd order)
3. Hard-Constrained PINNs

The implementation uses TensorFlow 2.x and is designed to run in Google Colab.

In [15]:
```
# Install required packages
!pip install pyDOE
```

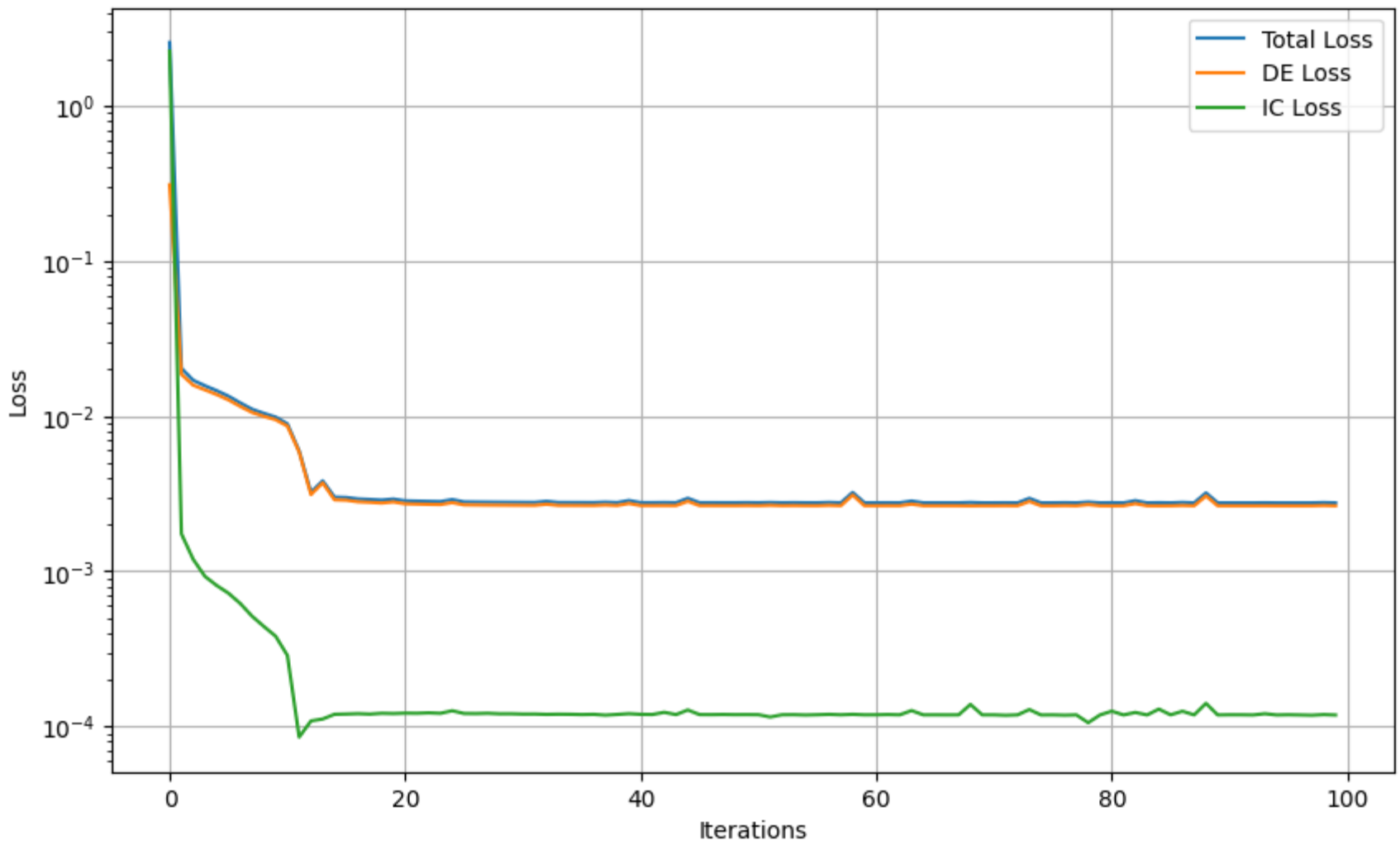
```
Requirement already satisfied: pyDOE in /usr/local/lib/python3.11/dist-packages (0.3.8)
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packages (from pyDOE) (1.26.4)
Requirement already satisfied: scipy in /usr/local/lib/python3.11/dist-packages (from pyDOE) (1.13.1)
```

In [17]:
```python
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from pyDOE import lhs
from scipy.integrate import solve_ivp

# Set random seeds for reproducibility
np.random.seed(1234)
tf.random.set_seed(1234)

# Utility Functions
def get_collocation_points(n_points, t_final):
    """Generate collocation points using Latin Hypercube Sampling"""
    t = lhs(1, n_points).flatten()
    t = t_final * t
    t = t.reshape(-1, 1)  # Reshape to (n_points, 1)
    return tf.convert_to_tensor(t, dtype=tf.float32)

class LossHistory:
    """Track and plot training losses"""
    def __init__(self):
        self.total_losses = []
        self.de_losses = []
        self.ic_losses = []

    def update(self, total_loss, de_loss, ic_loss):
        self.total_losses.append(float(total_loss))
        self.de_losses.append(float(de_loss))
        self.ic_losses.append(float(ic_loss))

    def plot(self):
        plt.figure(figsize=(10, 6))
        plt.semilogy(self.total_losses, label='Total Loss')
        plt.semilogy(self.de_losses, label='DE Loss')
        plt.semilogy(self.ic_losses, label='IC Loss')
        plt.xlabel('Iterations')
        plt.ylabel('Loss')
        plt.legend()
        plt.grid(True)
        plt.show()

# Base PINN Implementation
class PINN(tf.keras.Model):
    """Base class for Physics-Informed Neural Networks"""
    def __init__(self, layers, activation='tanh'):
        super().__init__()
        self.network_layers = []
        for units in layers[1:-1]:
            self.network_layers.append(tf.keras.layers.Dense(units, activation=activation))
        self.network_layers.append(tf.keras.layers.Dense(layers[-1]))

    def call(self, x):
        for layer in self.network_layers:
            x = layer(x)
        return x

# Lorenz-1960 System Implementation (Fixed)
class LorenzPINN(PINN):
    def __init__(self, layers=[1, 20, 20, 20, 20, 3]):
        super().__init__(layers)
        self.k = 1.0
        self.l = 2.0

    @tf.function
    def get_residuals(self, t):
        with tf.GradientTape(persistent=True) as tape:
            tape.watch(t)
            u = self(t)  # Shape: (n_points, 3)
            x, y, z = tf.split(u, 3, axis=1)

        # Compute gradients for each component
        dx_dt = tape.gradient(x, t)
        dy_dt = tape.gradient(y, t)
        dz_dt = tape.gradient(z, t)
        del tape  # Manually delete persistent tape

        k, l = self.k, self.l
        f_x = k * l * (1.0 / (k**2 + l**2) - 1.0 / k**2) * y * z - dx_dt
        f_y = k * l * (1.0 / l**2 - 1.0 / (k**2 + l**2)) * x * z - dy_dt
        f_z = k * l**2 * (1.0 / k**2 - 1.0 / l**2) * x * y - dz_dt

        return f_x, f_y, f_z

# Training Function for Lorenz System
def train_lorenz(t_final, n_points=1000, n_iter=10000):
    # Initial conditions
    x0, y0, z0 = 1.0, 0.5, 1.0

    # Create model and optimizer
    model = LorenzPINN()
    optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
```

```
# Generate collocation points
t = get_collocation_points(n_points, t_final)
t0 = tf.zeros((1, 1), dtype=tf.float32)

# Loss history tracking
history = LossHistory()

for i in range(n_iter):
    with tf.GradientTape() as tape:
        # Compute residuals
        f_x, f_y, f_z = model.get_residuals(t)

        # Initial conditions loss
        u0 = model(t0)
        x0_pred, y0_pred, z0_pred = tf.split(u0, 3, axis=1)
        ic_loss = tf.reduce_mean((x0_pred - x0)**2 +
                                 (y0_pred - y0)**2 +
                                 (z0_pred - z0)**2)

        # Differential equation loss
        de_loss = tf.reduce_mean(f_x**2 + f_y**2 + f_z**2)

        # Total loss
        loss = de_loss + ic_loss

    # Gradient descent step
    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))

    if i % 100 == 0:
        history.update(loss, de_loss, ic_loss)
        print(f'Iteration {i}, Loss: {loss:.6f}')

history.plot()
return model

# Train the model
model = train_lorenz(t_final=10.0, n_points=1000, n_iter=10000)
```

```
Iteration 0, Loss: 2.559804
Iteration 100, Loss: 0.020361
Iteration 200, Loss: 0.017041
Iteration 300, Loss: 0.015709
Iteration 400, Loss: 0.014597
Iteration 500, Loss: 0.013455
Iteration 600, Loss: 0.012178
Iteration 700, Loss: 0.011088
Iteration 800, Loss: 0.010429
Iteration 900, Loss: 0.009873
Iteration 1000, Loss: 0.008924
Iteration 1100, Loss: 0.005967
Iteration 1200, Loss: 0.003234
Iteration 1300, Loss: 0.003830
Iteration 1400, Loss: 0.003011
Iteration 1500, Loss: 0.002994
Iteration 1600, Loss: 0.002928
Iteration 1700, Loss: 0.002906
Iteration 1800, Loss: 0.002880
Iteration 1900, Loss: 0.002923
Iteration 2000, Loss: 0.002848
Iteration 2100, Loss: 0.002836
Iteration 2200, Loss: 0.002826
Iteration 2300, Loss: 0.002818
Iteration 2400, Loss: 0.002901
Iteration 2500, Loss: 0.002805
Iteration 2600, Loss: 0.002799
Iteration 2700, Loss: 0.002795
Iteration 2800, Loss: 0.002792
Iteration 2900, Loss: 0.002789
Iteration 3000, Loss: 0.002786
Iteration 3100, Loss: 0.002784
Iteration 3200, Loss: 0.002829
Iteration 3300, Loss: 0.002780
Iteration 3400, Loss: 0.002779
Iteration 3500, Loss: 0.002778
Iteration 3600, Loss: 0.002777
Iteration 3700, Loss: 0.002794
Iteration 3800, Loss: 0.002775
Iteration 3900, Loss: 0.002862
Iteration 4000, Loss: 0.002774
Iteration 4100, Loss: 0.002774
Iteration 4200, Loss: 0.002779
Iteration 4300, Loss: 0.002773
Iteration 4400, Loss: 0.002957
Iteration 4500, Loss: 0.002772
Iteration 4600, Loss: 0.002773
Iteration 4700, Loss: 0.002771
Iteration 4800, Loss: 0.002771
Iteration 4900, Loss: 0.002775
Iteration 5000, Loss: 0.002770
Iteration 5100, Loss: 0.002780
```

```
Iteration 5200, Loss: 0.002770
Iteration 5300, Loss: 0.002774
Iteration 5400, Loss: 0.002769
Iteration 5500, Loss: 0.002769
Iteration 5600, Loss: 0.002783
Iteration 5700, Loss: 0.002768
Iteration 5800, Loss: 0.003225
Iteration 5900, Loss: 0.002768
Iteration 6000, Loss: 0.002768
Iteration 6100, Loss: 0.002769
Iteration 6200, Loss: 0.002767
Iteration 6300, Loss: 0.002840
Iteration 6400, Loss: 0.002767
Iteration 6500, Loss: 0.002767
Iteration 6600, Loss: 0.002767
Iteration 6700, Loss: 0.002766
Iteration 6800, Loss: 0.002782
Iteration 6900, Loss: 0.002766
Iteration 7000, Loss: 0.002766
Iteration 7100, Loss: 0.002768
Iteration 7200, Loss: 0.002766
Iteration 7300, Loss: 0.002955
Iteration 7400, Loss: 0.002765
Iteration 7500, Loss: 0.002765
Iteration 7600, Loss: 0.002772
Iteration 7700, Loss: 0.002765
Iteration 7800, Loss: 0.002798
Iteration 7900, Loss: 0.002765
Iteration 8000, Loss: 0.002769
Iteration 8100, Loss: 0.002765
Iteration 8200, Loss: 0.002857
Iteration 8300, Loss: 0.002764
Iteration 8400, Loss: 0.002772
Iteration 8500, Loss: 0.002764
Iteration 8600, Loss: 0.002783
Iteration 8700, Loss: 0.002764
Iteration 8800, Loss: 0.003204
Iteration 8900, Loss: 0.002764
Iteration 9000, Loss: 0.002764
Iteration 9100, Loss: 0.002764
Iteration 9200, Loss: 0.002764
Iteration 9300, Loss: 0.002768
Iteration 9400, Loss: 0.002764
Iteration 9500, Loss: 0.002764
Iteration 9600, Loss: 0.002764
Iteration 9700, Loss: 0.002764
Iteration 9800, Loss: 0.002776
Iteration 9900, Loss: 0.002763
```



In [3]:
```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from pyDOE import lhs
from scipy.integrate import solve_ivp
import time

# Set random seeds for reproducibility
np.random.seed(1234)
tf.random.set_seed(1234)
```

Utility Functions

In [4]:
```
def get_collocation_points(n_points, t_final):
    """Generate collocation points using Latin Hypercube Sampling"""
    t = lhs(1, n_points).flatten()
    t = t_final * t
    return tf.convert_to_tensor(t, dtype=tf.float32)

class LossHistory:
    """Track and plot training losses"""
    def __init__(self):
        self.total_losses = []
        self.de_losses = []
        self.ic_losses = []

    def update(self, total_loss, de_loss, ic_loss):
        self.total_losses.append(float(total_loss))
        self.de_losses.append(float(de_loss))
        self.ic_losses.append(float(ic_loss))

    def plot(self):
        plt.figure(figsize=(10, 6))
        plt.semilogy(self.total_losses, label='Total Loss')
        plt.semilogy(self.de_losses, label='DE Loss')
        plt.semilogy(self.ic_losses, label='IC Loss')
        plt.xlabel('Iterations')
        plt.ylabel('Loss')
        plt.legend()
        plt.grid(True)
        plt.show()
```

Base PINN Implementation

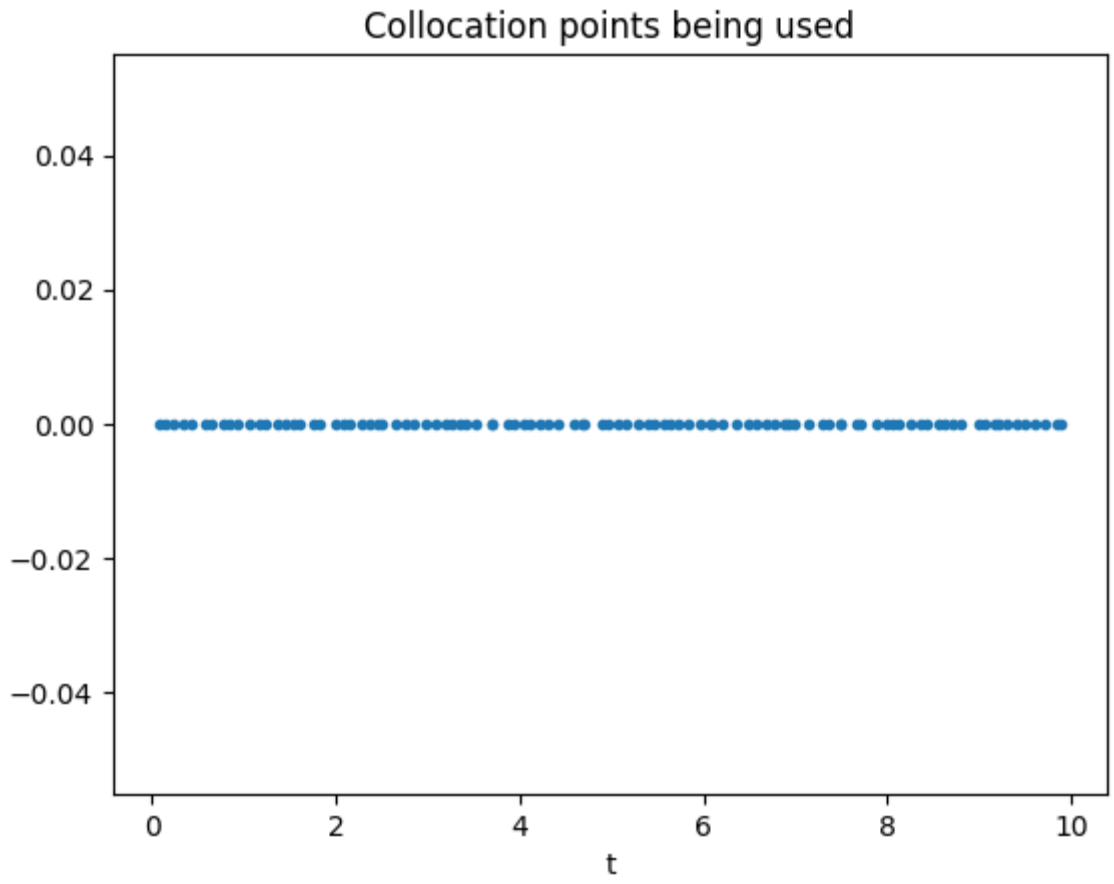
In [ ]:
```
class PINN(tf.keras.Model):
    """Base class for Physics-Informed Neural Networks"""
    def __init__(self, layers, activation='tanh'):
        super().__init__()
        self.network_layers = []
        for units in layers[1:-1]:
            self.network_layers.append(tf.keras.layers.Dense(units, activation=activation))
        self.network_layers.append(tf.keras.layers.Dense(layers[-1]))

    def call(self, x):
        for layer in self.network_layers:
            x = layer(x)
        return x
```

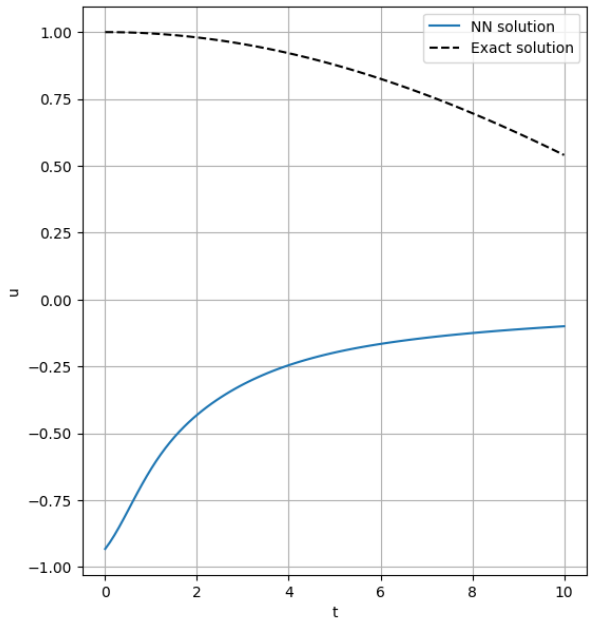
Lorenz-1960 System Implementation

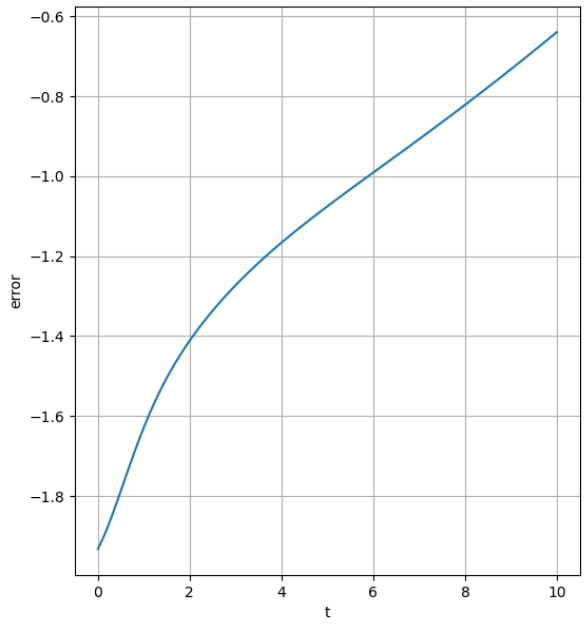
In [19]:
```python
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import qmc

# 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]
```

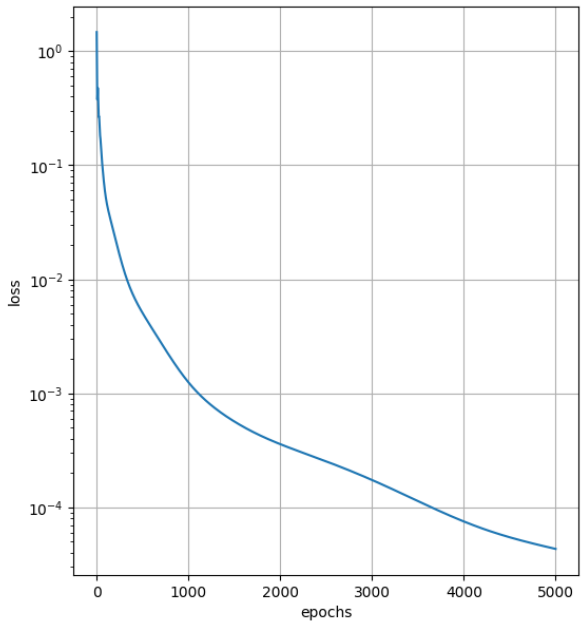
```
# Exact solution
uexact = u0 * np.cos(np.sqrt(k/m) * t) + (u_prime0/np.sqrt(k/m)) * np.sin(np.sqrt(k/m) * t)

# 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:  1.4717
Loss 100th epoch:  0.0518
Loss 200th epoch:  0.0232
Loss 300th epoch:  0.0118
Loss 400th epoch:  0.0072
Loss 500th epoch:  0.0051
Loss 600th epoch:  0.0037
Loss 700th epoch:  0.0028
Loss 800th epoch:  0.0021
Loss 900th epoch:  0.0016
Loss 1000th epoch:  0.0012
Loss 1100th epoch:  0.0010
Loss 1200th epoch:  0.0008
Loss 1300th epoch:  0.0007
Loss 1400th epoch:  0.0006
Loss 1500th epoch:  0.0006
Loss 1600th epoch:  0.0005
Loss 1700th epoch:  0.0005
Loss 1800th epoch:  0.0004
Loss 1900th epoch:  0.0004
Loss 2000th epoch:  0.0004
Loss 2100th epoch:  0.0003
Loss 2200th epoch:  0.0003
Loss 2300th epoch:  0.0003
Loss 2400th epoch:  0.0003
Loss 2500th epoch:  0.0003
Loss 2600th epoch:  0.0002
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:  0.0000
Loss 4800th epoch:  0.0000
Loss 4900th epoch:  0.0000
```









Harmonic Oscillator Implementation

In [8]:
```
class OscillatorPINN(PINN):
    def __init__(self, layers=[1, 20, 20, 20, 20, 1], second_order=False):
        super().__init__(layers)
        self.second_order = second_order
        self.m = 1.0
        self.k = 2.0

    @tf.function
    def get_residuals(self, t):
        if self.second_order:
            with tf.GradientTape() as tape2:
                tape2.watch(t)
                with tf.GradientTape() as tape1:
                    tape1.watch(t)
                    u = self(t)
                du_dt = tape1.gradient(u, t)
            d2u_dt2 = tape2.gradient(du_dt, t)

            return self.m * d2u_dt2 + self.k * u
        else:
            with tf.GradientTape() as tape:
                tape.watch(t)
                u = self(t)
                x, v = tf.split(u, 2, axis=1)

            du_dt = tape.gradient(u, t)
            dx_dt, dv_dt = tf.split(du_dt, 2, axis=1)

            f_x = dx_dt - v
            f_v = dv_dt + (self.k/self.m) * x

            return f_x, f_v

def train_oscillator(t_final, second_order=False, n_points=1000, n_iter=10000):
    # Initial conditions
    x0, v0 = 1.0, 1.0

    # Create model and optimizer
    if second_order:
        model = OscillatorPINN(layers=[1, 20, 20, 20, 20, 1], second_order=True)
    else:
        model = OscillatorPINN(layers=[1, 20, 20, 20, 20, 2])

    optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)

    # Generate collocation points
    t = get_collocation_points(n_points, t_final)
    t0 = tf.zeros((1, 1), dtype=tf.float32)

    history = LossHistory()

    for i in range(n_iter):
        with tf.GradientTape() as tape:
            if second_order:
                # Second order implementation
                residual = model.get_residuals(t)
                de_loss = tf.reduce_mean(residual**2)

                # Initial conditions
                with tf.GradientTape() as tape_ic:
                    tape_ic.watch(t0)
                    u0 = model(t0)
                du0_dt = tape_ic.gradient(u0, t0)

                ic_loss = tf.reduce_mean((u0 - x0)**2 + (du0_dt - v0)**2)
            else:
                # First order system implementation
                f_x, f_v = model.get_residuals(t)
                de_loss = tf.reduce_mean(f_x**2 + f_v**2)

                # Initial conditions
                u0 = model(t0)
                x0_pred, v0_pred = tf.split(u0, 2, axis=1)
                ic_loss = tf.reduce_mean((x0_pred - x0)**2 + (v0_pred - v0)**2)

            # Total loss
            loss = de_loss + ic_loss

        # Gradient descent step
        grads = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(grads, model.trainable_variables))

        if i % 100 == 0:
            history.update(loss, de_loss, ic_loss)
            print(f'Iteration {i}, Loss: {loss:.6f}')

    history.plot()
    return model
```

Hard-Constrained PINN Implementation

In [9]:
```python
class HardConstrainedPINN(PINN):
    """Hard-constrained Physics-Informed Neural Network"""
    def __init__(self, layers=[1, 20, 20, 20, 20, 3], initial_conditions=None):
        super().__init__(layers)
        self.initial_conditions = initial_conditions or {}

    def call(self, t):
        # Base network output
        N = super().call(t)

        # Apply hard constraints using t·N(t) formulation
        outputs = []
        for i, (name, ic) in enumerate(self.initial_conditions.items()):
            y = ic + t * N[:, i:i+1]
            outputs.append(y)

        return tf.concat(outputs, axis=1)

class TaylorHardConstrainedPINN(PINN):
    """Second-order Taylor expansion based hard-constrained PINN"""
    def __init__(self, layers=[1, 20, 20, 20, 20, 1], u0=0.0, v0=0.0):
        super().__init__(layers)
        self.u0 = u0
        self.v0 = v0

    def call(self, t):
        # Base network output
        N = super().call(t)

        # Second-order Taylor expansion
        y = self.u0 + self.v0 * t + t**2 * N
        return y

def train_hard_constrained_lorenz(t_final, n_points=1000, n_iter=10000):
    x0, y0, z0 = 1.0, 0.5, 1.0

    # Create model with hard constraints
    model = HardConstrainedPINN(initial_conditions={'x': x0, 'y': y0, 'z': z0})
    optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)

    # Generate collocation points
    t = get_collocation_points(n_points, t_final)

    history = LossHistory()

    for i in range(n_iter):
        with tf.GradientTape() as tape:
            f_x, f_y, f_z = model.get_residuals(t)
            de_loss = tf.reduce_mean(f_x**2 + f_y**2 + f_z**2)
            loss = de_loss  # No IC loss needed for hard constraints

        grads = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(grads, model.trainable_variables))

        if i % 100 == 0:
            history.update(loss, de_loss, 0.0)
            print(f'Iteration {i}, Loss: {loss:.6f}')

    history.plot()
    return model
```

Example Usage and Comparison

In [18]:
```python
def get_collocation_points(n_points, t_final):
    """Generate collocation points using Latin Hypercube Sampling"""
    t = lhs(1, n_points).flatten()
    t = t_final * t
    # Reshape t to have an extra dimension for the batch size
    t = t.reshape(-1, 1)  # Reshape to (n_points, 1)
    return tf.convert_to_tensor(t, dtype=tf.float32)
```