Inverse Physics-Informed Neural Network

John Morrow

1 Introduction: What is physics-informed?

Many relationships in physics, biology, chemistry, economics, engineering, etc., are defined by differential equations. (Check here for an extensive list.) In general, a differential equation (DE) describes how variables are affected by the rate of change of other variables. For instance, a DE explains how the position of a mass vibrating on spring changes with time in relation to the mass's velocity and acceleration. A physics-informed neural network (PINN) produces responses that adhere to the relationship described by a DE (whether the subject is physics, engineering, economics, etc.). In contrast, an inverse physics-informed neural network (iPINN) acts on a response and determines the parameters of the DE that produced it. PINNs and iPINNs are trained by including a constraint during training that forces the relationship between the input and output of the neural network to conform to the DE being modeled.

This article begins with the implementation of a PINN, then builds on the PINN model to implement an iPINN. The analytical solution for the modeled DE is included for comparison to the responses produced by the PINN and iPINN.

2 Second-order differential equations

This article focuses on a PINN and iPINN for DEs that describe damped harmonic motion, e.g., a spring-mass system with damping (Figure 1) and an electronic circuit comprising series-connected components of resistance, inductance, and capacitance (RLC) (Figure 2).

These applications are defined by second-order DEs, which include second derivatives with respect to time. Equation 1 is the second-order differential equation for a spring-mass system, where the parameters m, c, and k are, respectively, mass, damping coefficient, and spring constant. The displacement of the mass is represented by x, and time by t. The second derivative of x with respect to t is the acceleration of the mass,

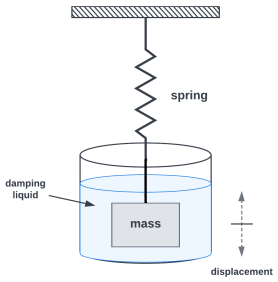

Figure 1: **Vibrating Mass & Spring**

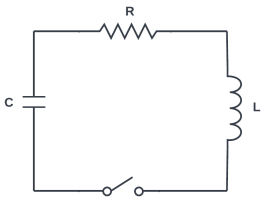

Figure 2: **RLC Circuit**

and the first derivative is the velocity of the mass.

$$m\frac{d^2x}{dt^2} + c\frac{dx}{dt} + kx = 0 \tag{1}$$

Equation 2 is the second-order DE for the RLC circuit, where R, L, and C are, respectively, resistance, inductance, and capacitance. The current in the circuit is represented by i, and the time by t.

$$L\frac{d^2i}{dt^2} + R\frac{di}{dt} + \frac{1}{C}i = 0 \tag{2}$$

Both of these DEs produce similar responses, i.e., the motion of the mass when it is displaced from a resting position, then released, and the variation of the current over time when the switch is closed after pre-charging the capacitor with an initial voltage. The following section presents details of the responses of the RLC circuit.

3 RLC circuit response

Following is an overview of the possible responses of the RLC circuit in Figure 2, including the equation of the analytical solution to Equation 2 for each response. (A derivation of the analytical solution by the author is available for download here.) The analytical responses will later be compared to PINN-derived and iPINN-derived responses.

Depending upon the values of the components, this RLC circuit can produce three different types of responses: under-damped, critically damped, and over-damped. All three responses are based on the capacitor

charged to a voltage, V_0, prior to switch closure and the following initial conditions:

$$i = 0 \quad at\ t = 0 \tag{3}$$

$$L\frac{di}{dt} = V_0 \quad at\ t = 0 \tag{4}$$

3.1 Under-damped response

An under-damped response occurs when the values of R, L, and C produce the following condition:

$$\frac{R}{2L} < \frac{1}{\sqrt{LC}} \tag{5}$$

As an example, let $R = 1.2$ (ohms), $L = 1.5$ (henries), $C = 0.3$ (farads), and $V_0 = 12$ (volts). The analytically-derived response to Equation 2 with these values is:

$$i(t) = 5.57e^{-0.4t}sin(1.44t) \tag{6}$$

The following is a plot of the response from Equation 6.

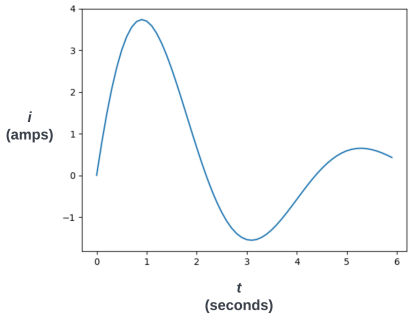

Figure 3: **Under-damped response**

3.2 Critically damped response

A critically-damped response occurs when the values of R, L, and C produce the following condition:

$$\frac{R}{2L} = \frac{1}{\sqrt{LC}} \tag{7}$$

As an example, let $R = 4.47$ (ohms), $L = 1.5$ (henries), $C = 0.3$ (farads), and $V_0 = 12$ (volts). The analytically-

derived response to Equation 2 with these values is:

$$i(t) = 8.0te^{-1.49t} \tag{8}$$

The following is a plot of the response from Equation 8.

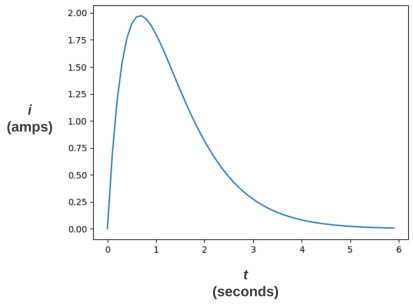

Figure 4: **Critically-damped response**

3.3 Over-damped response

An over-damped response occurs when the values of R, L, and C produce the following condition:

$$\frac{R}{2L} > \frac{1}{\sqrt{LC}} \tag{9}$$

As an example, let $R = 6.0$ (ohms), $L = 1.5$ (henries), $C = 0.3$ (farads), and $V_0 = 12$ (volts). The analytically-derived response to Equation 2 with these values is:

$$i(t) = 3.0(e^{-0.67t} - e^{-3.33t}) \tag{10}$$

The following is a plot of the response from Equation 10.

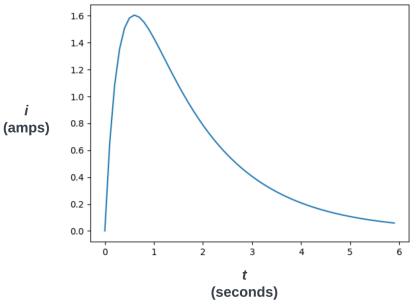

Figure 5: **Over-damped response**

4 PINN structure

Typically, a neural network is trained with pairs of known input and output data. The training input data is presented to the neural network, and the resulting output is compared to the training output data using a loss function. The loss returned by this function is used via backpropagation to adjust the network's weights to reduce the loss. PINNs and iPINNs use custom loss functions that include additional loss components for constraining the neural network to produce outputs that comply with the DE being modeled.

A PINN model of the DE in Equation 2 accepts time, t, as input to the neural network and produces a corresponding current, i, as output. Training the PINN to comply with the DE requires both the first and second derivatives of the output with respect to the input, i.e., di/dt and d^2i/dt^2. These derivatives are available in TensorFlow and PyTorch through each platform's automatic differentiation function. In this article, the PINN and iPINN are developed with TensorFlow GradientTape.

For each training input to the PINN, the first and second derivatives from GradientTape are combined with R, L, and C according to the DE in Equation 2 to produce a result that should equal zero. The difference between the actual result and zero is known as the *residual*. The residual becomes a component of the loss function used to train the PINN.

A second-order DE, such as Equation 2, also requires that the solution complies with two initial conditions. In this case, the first condition is the value of i at $t = 0$ (Equation 3) and the second is the value of di/dt at $t = 0$ (Equation 4). Each initial condition is included as a component of the loss function.

Figure 6 illustrates the composition of the total loss. Loss 2 is from the residual. Loss 1 and loss 3 are from the initial conditions. During training, backpropagation is used to reduce the total loss. The PINN outputs for d^2i/dt^2 and di/dt are provided by GradientTape.

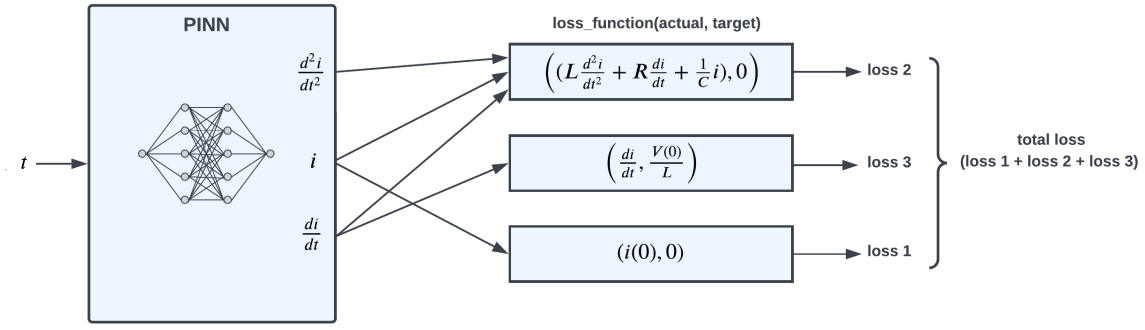

Figure 6: **PINN loss function**

5 PINN implementation

Following is the python code for the PINN implementation. The complete code for the PINN implementation is available here.

5.1 Neural network model definition

The neural network for the PINN has two fully-connected hidden layers, each with 128 neurons. There is a single input for time points and a single output for the response points.

Listing 1: PINN TensorFlow model

```
# Hyperparameters
batch_size = 1
epochs = 600
optimizer = Adam(learning_rate=0.001)
weight_init = RandomNormal()

# Build model
inputs = tf.keras.Input(shape=(1,))
x = layers.Dense(128, activation='gelu', kernel_initializer=weight_init,
                 kernel_regularizer=None)(inputs)
x = layers.Dense(128, activation='gelu', kernel_initializer=weight_init,
                 kernel_regularizer=None)(x)
output = layers.Dense(1, activation='linear', kernel_initializer=weight_init)(x)
model = tf.keras.Model(inputs, output)
```

5.2 PINN initialization

In the PINN model, the R, L, and C component values and the initial capacitor voltage (lines 2-5) are constants that determine the response of the DE. The co-location points, specified in the time domain (line 8), are the points where the residual is calculated. The initial conditions (lines 11 and 15) are from Equation 3 and Equation 4.

Listing 2: PINN initialization

```
1  # circuit parameters (under-damped)
2  R = 1.2 # resistance, ohm
3  L = 1.5 # inductance, H
4  C = 0.3 # capacitance, F
5  VC_0 = 12.0 # volts, initial capacitor voltage
6
7  # co-location points
8  t_coloc = np.arange(0, 6.0, 0.3)
9
10 # initial conditions:
11    # f(t) initial conditions
```

```
t_init = np.array([0.0]) # time, sec
i_init = np.array([0.0]) # current, A

   # f'(t) initial conditions
t_init2 = np.array([0.0]) # time, sec
v_init2 = np.array([VC_0]) # d/dt(i) * L = VC_0
```

5.3 PINN training step

Following is the python code for the training step function. For each training batch, the step function calculates the three components of loss, then uses the total loss to update the weights in the neural network.

loss 1: The initial condition from Equation 3 is compared to the output of the network, $pred\_y$ (line 9). The square of the difference is $model\_loss1$ (line 10).

loss 2: The residual (line 30) is calculated at the co-location points. It uses the first-order gradient, $dfdx$ (line 17), and the second-order gradient, $dfdx2$ (line 26), from GradientTape, along with the output of the network, $pred\_y$ (line 29), to calculate the left-hand side of Equation 2. This value squared is $model\_loss2$ (line 31).

loss 3: The initial condition from Equation 4 compares the product of L and the first-order gradient, $dfdx$ (line 17), to v_init2. The square of the difference is $model\_loss3$ (line 19).

The total of the three loss components, $model\_loss$ (line 35), is used to calculate the gradients of the loss with respect to the neural network's weights (line 38). The optimizer then updates the weights (line 41).

Listing 3: PINN training step

```
# Step function
def step(t_coloc, t_init, i_init, t_init2, i_init2):
    t_coloc = tf.convert_to_tensor(t_coloc)
    t_coloc = tf.reshape(t_coloc, [batch_size, 1])
    t_coloc = tf.Variable(t_coloc, name='t_coloc')
    with tf.GradientTape(persistent=True) as tape:

        # model_loss1: initial condition i_init @ t_init -> f(t) initial condition
        pred_init = model(t_init)
        model_loss1 = math_ops.squared_difference(pred_init, i_init)

        # model_loss3: initial condition i_init2 @ t_init2 -> f'(t) initial condition
        t_init2 = tf.convert_to_tensor(t_init2)
        t_init2 = tf.reshape(t_init2, [1, 1])
        t_init2 = tf.Variable(t_init2, name='t_init2')
        pred_init2 = model(t_init2)
        dfdx = tape.gradient(pred_init2, t_init2) # f'(t)
        vl = dfdx * tf.cast(L, tf.float64) # inductor voltage due to dfdx
        model_loss3 = math_ops.squared_difference(vl, v_init2)
```

```
        model_loss3 = tf.cast(model_loss3, tf.float32)

        # 1st and 2nd order gradients for co-location pts
        with tf.GradientTape(persistent=True) as tape2:
            pred_y = model(t_coloc)
            dfdx = tape2.gradient(pred_y, t_coloc) # f'(t)
        dfdx2 = tape2.gradient(dfdx, t_coloc) # f"(t)

        # model_loss2: collocation points
        pred_y = tf.cast(pred_y, tf.float64)
        residual = dfdx2 + dfdx * (R / L) + pred_y / (L * C)
        model_loss2 = K.mean(math_ops.square(residual), axis=-1)
        model_loss2 = tf.cast(model_loss2, tf.float32)

        #total loss
        model_loss = model_loss1 + model_loss2 + model_loss3

        trainable = model.trainable_variables
        model_gradients = tape.gradient(model_loss, trainable)

        # Update model
        optimizer.apply_gradients(zip(model_gradients, trainable))
```

6 PINN results

The results of training a PINN for three test cases follow. The tests are for the conditions of section 3: under-damped, critically-damped, and over-damped. Each plot below presents three traces:

- the response of the analytical equation (blue)
- the co-location points (green)
- the output response of the trained PINN (red)

Under-damped test case:

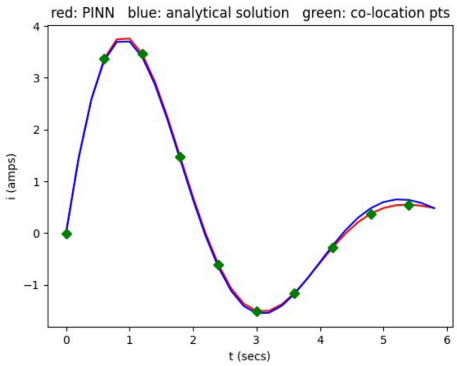

Figure 7: **Under-damped response**

Critically-damped test case:

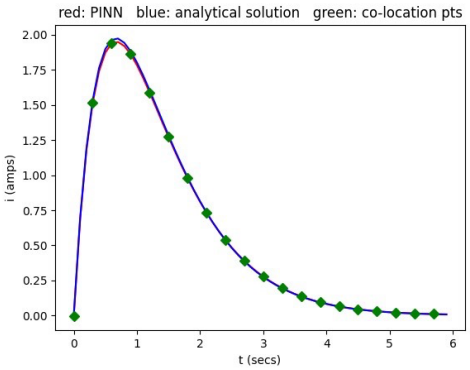

Figure 8: **Critically-damped response**

Over-damped test case:

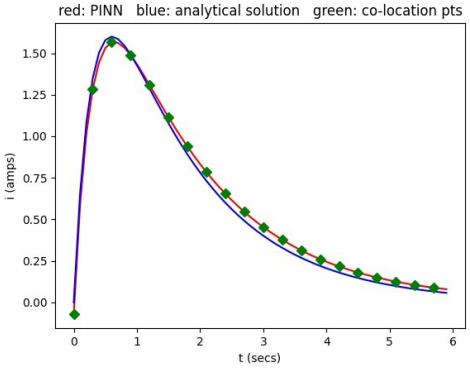

Figure 9: **Over-damped response**

7 iPINN structure

Figure 10 illustrates the composition of the total loss. Like the PINN model, Loss 2 is from the residual, except that R, L, and C are variables whose values are determined during training. In contrast, R, L, and C are constants in the PINN model. As in the PINN, loss 1 and loss 3 force compliance with the initial conditions of Equation 3 and Equation 4.

The iPINN model includes an additional loss function, loss 4, that forces the output response to match the response of the DE under investigation.

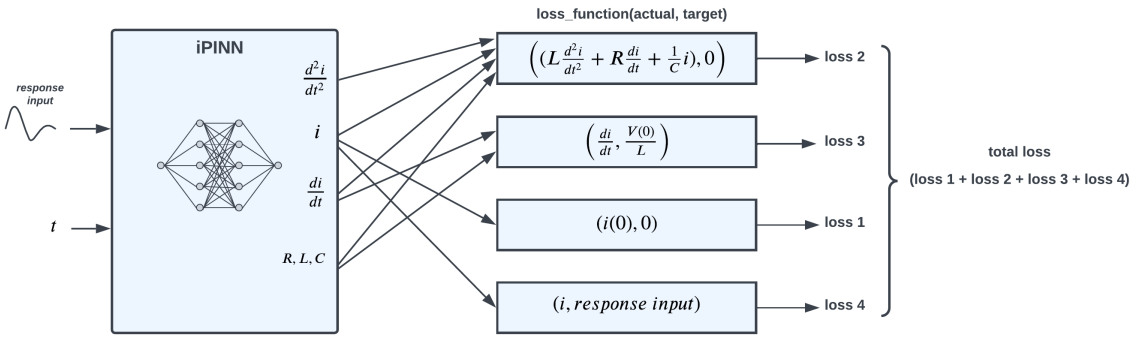

Figure 10: **iPINN loss function**

8 iPINN implementation

Following is the python code for the PINN implementation. The complete code for the iPINN implementation is available here.

The neural network model definition for iPINN is identical to the PINN network defined above, i.e., two fully-connected hidden layers, each with 128 neurons. There is a single input for time points and a single output for the response points.

8.1 iPINN initialization

The response of the DE under investigation is loaded in line 4. The two initial conditions (lines 9 and 13) are the same as in the PINN model. As discussed above, R, L, and C are trainable variables in the iPINN model (lines 18-20).

Listing 4: iPINN initialization

```
1  VC_0 = 12.0 # volts, initial capacitor voltage
2
3  # Load response data file
4  pinn_data = np.load('[insert path here]/pinn_data.npy')
5  t_coloc = pinn_data[:, 0]
6  i_coloc = pinn_data[:, 1]
7
8  # initial conditions
9     # f(t) initial conditions
10 t_init = np.array([0.0]) # time, sec
11 i_init = np.array([0.0]) # current, A
12
13    # f'(t) initial conditions
14 t_init2 = np.array([0.0]) # time, sec
15 v_init2 = np.array([VC_0]) # d/dt(i) * L = VC_0
16
```

```
# additional variables added to gradient tracking
R = tf.Variable([0.1], trainable=True)
L = tf.Variable([0.1], trainable=True)
C = tf.Variable([0.1], trainable=True)
```

8.2 iPINN training step

loss 1: The initial condition from Equation 3 is compared to the output of the network, $pred\_y$ (line 10). The square of the difference is $model\_loss1$ (line 11).

loss 2: As in the PINN training step function, the residual (line 34) is calculated at the co-location points defined by t_coloc to produce $model\_loss2$ (line 35). R, L, and C are trainable variables.

loss 3: The initial condition from Equation 4 compares the product of L, a trainable variable, and the first-order gradient, $dfdx$ (line 18), to v_init2. The square of the difference is $model\_loss3$ (line 20).

loss 4: This loss component compares the output of the network (line 39) to the response of the DE under investigation, i_coloc, to produce $model\_loss4$ (line 40).

The total of the four loss components, $model\_loss$ (line 43), is used to calculate the gradients of the loss with respect to the neural network's weights and the three trainable variables: R, L, and C (line 49). The optimizer then updates the network's weights (line 52), and the R, L, and C values are updated in lines 53-55.

Listing 5: iPINN training step

```
# Step function
def step(t_coloc, i_coloc, t_init, i_init, t_init2, v_init2, rlc_lr):

    t_coloc = tf.convert_to_tensor(t_coloc)
    t_coloc = tf.reshape(t_coloc, [batch_size, 1])
    t_coloc = tf.Variable(t_coloc, name='t_coloc')
    with tf.GradientTape(persistent=True) as tape:

        # model_loss1: initial condition i_init @ t_init -> f(t) initial condition
        pred_init = model(t_init)
        model_loss1 = math_ops.squared_difference(pred_init, i_init)

        # model_loss3: initial condition v_init2 @ t_init2 -> f'(t) initial condition
        t_init2 = tf.convert_to_tensor(t_init2)
        t_init2 = tf.reshape(t_init2, [1, 1])
        t_init2 = tf.Variable(t_init2, name='t_init2')
        pred_init2 = model(t_init2)
        dfdx = tape.gradient(pred_init2, t_init2) # f'(t)
        vl = dfdx * tf.cast(L, tf.float64) # inductor voltage due to dfdx
        model_loss3 = math_ops.squared_difference(vl, v_init2)
        model_loss3 = tf.cast(model_loss3, tf.float32)

        # 1st and 2nd order gradients for co-location pts
```

```
        with tf.GradientTape(persistent=True) as tape2:
            pred_y = model(t_coloc)
            dfdx = tape2.gradient(pred_y, t_coloc) # f'(x)
        dfdx2 = tape2.gradient(dfdx, t_coloc) # f"(t)

        # model_loss2: collocation points
        pred_y = tf.cast(pred_y, tf.float64)
        R64 = tf.cast(R, tf.float64)
        L64 = tf.cast(L, tf.float64)
        C64 = tf.cast(C, tf.float64)
        residual = dfdx2 + (R64 / L64) * dfdx + pred_y / (L64 * C64
        model_loss2 = K.mean(math_ops.square(residual), axis=-1)
        model_loss2 = tf.cast(model_loss2, tf.float32)

        # model_loss4: response
        pred_response = model(t_coloc)
        model_loss4 = K.mean(math_ops.squared_difference(pred_response, i_coloc), axis=-1)

        #total loss
        model_loss = model_loss1 + model_loss2 + model_loss3 + model_loss4 * 10

        trainable = model.trainable_variables
        trainable.append(R)
        trainable.append(L)
        trainable.append(C)
        model_gradients = tape.gradient(model_loss, trainable)

        # Update model
        optimizer.apply_gradients(zip(model_gradients, trainable))
        R.assign_sub(model_gradients[6] * rlc_lr)
        L.assign_sub(model_gradients[7] * rlc_lr)
        C.assign_sub(model_gradients[8] * rlc_lr)

    return R, L, C, model_loss
```

9 iPINN results

The results of using an iPINN to identify three unknown test responses follow. The test responses presented to the iPINN were generated with the conditions of section 3: under-damped, critically-damped, and over-damped. The tables below compare the R, L, and C component values used to generate the test response to the values determined by the iPINN. Each plot below presents three traces:

- the response curve of the analytical equation (blue)
- the response data (60 points) to be identified by the iPINN (green)
- the output response of the trained iPINN (red)

Under-damped test case:

Generating values vs iPINN-determined values		
circuit parameter	**generating value**	**iPINN value**
R (ohms)	1.20	1.20
L (henries)	1.50	1.60
C (farads)	0.30	0.31

Table 1: **Under-damped test case**

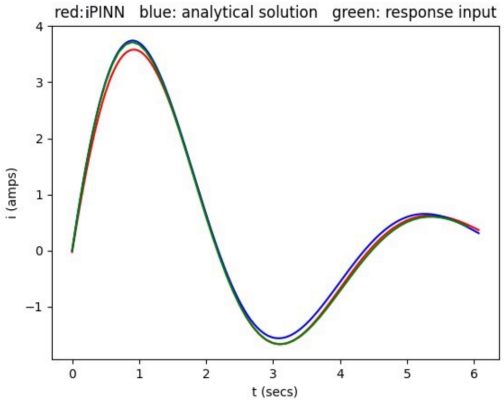

Figure 11: **Under-damped response**

Critically-damped test case:

Generating values vs iPINN-determined values		
circuit parameter	**generating value**	**iPINN value**
R (ohms)	4.47	4.54
L (henries)	1.50	1.58
C (farads)	0.30	0.32

Table 2: **Critically-damped test case**

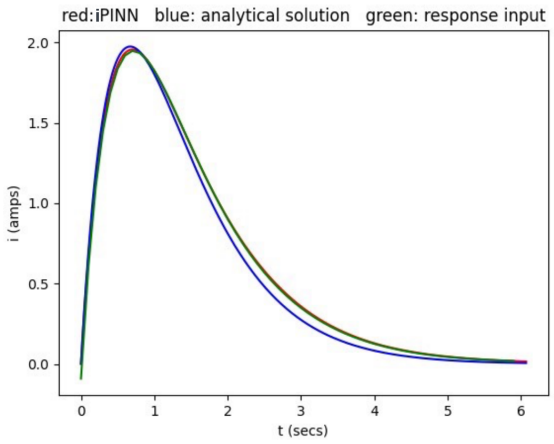

Figure 12: **Critically-damped response**

Over-damped test case:

Generating values vs iPINN-determined values		
circuit parameter	**generating value**	**iPINN value**
R (ohms)	6.00	6.12
L (henries)	1.50	1.66
C (farads)	0.30	0.31

Table 3: **Over-damped test case**

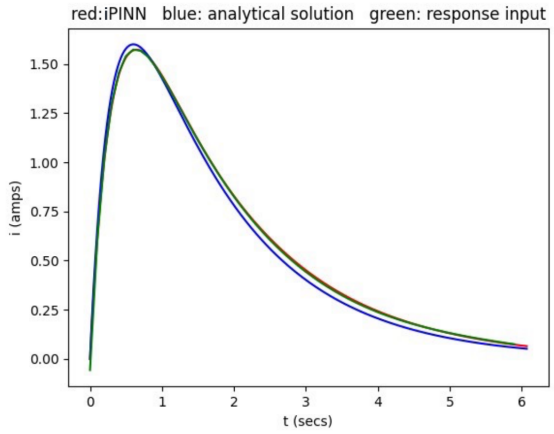

Figure 13: **Over-damped response**

10 Conclusion

This study demonstrates that a neural network can successfully solve differential equations, which describe many relationships in numerous fields of science, engineering, and economics. A physics-informed neural network is trained to solve the second-order differential equation of an electronic circuit resulting in a neural network that produces the same response to an input signal as the actual circuit.

This study also demonstrates that a neural network can determine the parameters of an unknown differential equation. Specifically, an *inverse* physics-informed neural network is trained to determine the unknown component values of an electronic circuit using only a sample response from the circuit. Further, after determining the unknown component values, the resulting neural network can produce the same response to an input signal as the actual circuit.

Bibliography

[1] M. Raissi, P. Perdikaris, and G. E. Karniadakis, "Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations," 2017. [Online]. Available: https://arxiv.org/abs/1711.10561

[2] M. Raissi, P. Perdikaris, and G. E. Karniadakis, "Physics informed deep learning (part ii): Data-driven discovery of nonlinear partial differential equations," 2017. [Online]. Available: https://arxiv.org/abs/1711.10566