Test the new features of the `Screwdriver`, `Screw`, and `MetricScrew` classes with the following steps:

a. Create an instance `ms1` of `MetricScrew` with right-handedness, a flat head, initial angle 0 rad, and thread pitch 2 mm (corresponding to an M14 metric screw)
b. Create an instance `sd1` of `Screwdriver` with a flat head
c. Turn the `ms1` screw 5 complete *clockwise* revolutions with the `sd1` screwdriver and print the resulting angle and depth of `ms1`
d. Turn the `ms1` screw 3 complete *counterclockwise* revolutions with the `sd1` screwdriver and print the resulting angle and depth of `ms1`
e. Create an instance `ms2` of `MetricScrew` that is the same as `ms1`, but with *left*-handedness
f. Turn the `ms2` screw 4 complete *counterclockwise* revolutions with the `sd1` screwdriver and print the resulting angle and depth of `ms2`
g. Turn the `ms2` screw 2 complete *clockwise* revolutions with the `sd1` screwdriver and print the resulting angle and depth of `ms2`
h. Create an instance `sd2` of `Screwdriver` with a hex head and try to turn the `sd1` screw and catch and print the exception

Solution 2.5 UZ Load the NumPy package:

```
import numpy as np
```

Define Classes The following `Screwdriver` class meets the requirements:

```
class Screwdriver:
    """Represents a screwdriver tool"""
    operates_on = "Screw"  # Class data attributes
    operated_by = "Hand"

    def __init__(self, head, length):
        self.head = head  # Instance data attributes
        self.length = length

    def drive(self, screw, angle):  # Method definition
        """Returns a screw object turned by the given angle"""
        if screw.head != self.head:
            raise TypeError(f"{self.head} screwdriver "
                f"can't turn a {screw.head} screw.")
        screw.turn(angle)
        return screw
```

The following `Screw` class meets the requirements:

```python
class Screw:
    """Represents a screw fastener"""
    def __init__(self, head, pitch, depth=0, angle=0, handed="Right"):
        self.head = head
        self.pitch = pitch
        self.depth = depth
        self.angle = angle
        self.handed = handed

    def turn(self, angle):
        """Mutates angle and depth for a turn of angle rad"""
        if self.handed == "Right":
            handed_sign = 1
        else:
            handed_sign = -1
        self.angle += angle
        self.depth += handed_sign * self.pitch * angle / (2*np.pi)
```

The following `MetricScrew` class meets the requirements:

```python
class MetricScrew(Screw):
    """Represents a metric screw fastener"""
    kind = "Metric"
    # No constructor necessary because we aren't
    # changing instance attributes
```

Test the New Features Create a `MetricScrew` instance as follows:

```python
ms1 = MetricScrew(head="Flat", pitch=2)
```

Create a flathead screwdriver instance:

```python
sd1 = Screwdriver(head="Flat", length=6)
```

Turn the screw 5 complete clockwise revolutions with the screwdriver and print the resulting angle and depth as follows:

```python
sd1.drive(ms1, 5*2*np.pi)
print(f"Angle: {ms1.angle:.3g} rad \nDepth: {ms1.depth} mm")
```

```
<__main__.MetricScrew at 0x11d678750>
```

```
Angle: 31.4 rad
Depth: 10.0 mm
```

Turn the screw 3 complete counterclockwise revolutions with the screwdriver and print the resulting angle and depth as follows:

```python
sd1.drive(ms1, -3*2*np.pi)
print(f"Angle: {ms1.angle:.3g} rad \nDepth: {ms1.depth} mm")
```

```
<__main__.MetricScrew at 0x11d678750>
```

```
Angle: 12.6 rad
Depth: 4.0 mm
```

Create a left-handed `MetricScrew` instance as follows:

```
ms2 = MetricScrew(head="Flat", pitch=2, handed="Left")
```

Turn the `ms2` screw 4 complete counterclockwise revolutions with the `sd1` screwdriver and print the resulting angle and depth of `ms2` as follows:

```
sd1.drive(ms2, -3*2*np.pi)
print(f"Angle: {ms2.angle:.3g} rad \nDepth: {ms2.depth} mm")
```

```
<__main__.MetricScrew at 0x11d67a1d0>
```

```
Angle: -18.8 rad
Depth: 6.0 mm
```

Turn the `ms2` screw 2 complete clockwise revolutions with the `sd1` screwdriver and print the resulting angle and depth of `ms2` as follows:

```
sd1.drive(ms2, 2*2*np.pi)
print(f"Angle: {ms2.angle:.3g} rad \nDepth: {ms2.depth} mm")
```

```
<__main__.MetricScrew at 0x11d67a1d0>
```

```
Angle: -6.28 rad
Depth: 2.0 mm
```

Create an instance `sd2` of `Screwdriver` with a hex head and try to turn the `sd1` screw and catch and print the exception as follows:

```
sd2 = Screwdriver(head="Hex", length=6)
try:
    sd2.drive(ms1, 1)  # Should raise an exception
except Exception as err:
    print(f"Unexpected {type(err)}: {err}")  # Print the exception
```

```
Unexpected <class 'TypeError'>: Hex screwdriver can't turn a Flat
↪  screw.
```

Problem 2.6 VX Improve the bubble sort algorithm of algorithm 1 by adding a test that can return the list if it is sorted before completing all the loops. Implement the improved bubble sort algorithm in a program that it meets the following requirements:

a. It defines a function `bubble_sort(l: list) -> list` that implements the bubble sort algorithm.
b. It demonstrates the `bubble_sort()` function works on three different lists of numbers.

```
t = sp.symbols("t", real=True)
F = sp.Matrix([
  [sp.exp(-t)],
  [sp.exp(-t)],
  [1 - sp.exp(-t)],
  [1 - sp.exp(-t)]
])
G = sp.Matrix([
  [sp.exp(-t)],
  [sp.exp(-t)],
  [1 - sp.exp(-t)],
  [sp.exp(-t)]
])
```

Now compute the energy:

```
E = energy(F, G).simplify()
print(E)
```

$$\left[\begin{matrix}\frac{1}{2} - \frac{e^{-2t}}{2}\\ \frac{1}{2} - \frac{e^{-2t}}{2}\\ t - \frac{3}{2} + 2e^{-t} - \frac{e^{-2t}}{2}\\ \frac{1}{2} - e^{-t} + \frac{e^{-2t}}{2}\end{matrix}\right]$$

Problem 4.8 FJ For the circuit and state-space model given in problem 4.6, use SymPy to solve for $\boldsymbol{x}(t)$ and $\boldsymbol{y}(t)$ given the following:

- A constant input voltage $V_S(t) = \overline{V_S}$
- Initial condition $\boldsymbol{x}(0) = \boldsymbol{0}$

Substitute the following parameters into the solution for $\boldsymbol{y}(t)$ and create numerically evaluable functions of time for each variable in $\boldsymbol{y}(t)$:

$$R = 50\ \Omega,\ L = 10 \cdot 10^{-6}\ \text{H},\ C = 1 \cdot 10^{-9}\ \text{F},\ \overline{V_S} = 10\ \text{V}.$$

Plot the outputs in $\boldsymbol{y}(t)$ as functions of time, making sure to choose a range of time over which the response is best presented. *Hint:* An appropriate amount of time is on the scale of microseconds.

Solution 4.8 FJ Load packages:

```
import numpy as np
import sympy as sp
import matplotlib.pyplot as plt
```

Define Classes We begin by defining the parameters and functions of time as SymPy symbolic variables and unspecified functions as follows:

```
R, L, C = sp.symbols("R, L, C", positive=True)
v_C, i_L, v_L, V_S = sp.symbols(
    "v_C, i_L, v_L, V_S", cls=sp.Function, real=True
)  # v_C, i_L, V_S
t = sp.symbols("t", real=True)
```

Now we can form the symbolic matrices and vectors:

```
A_ = sp.Matrix([[0, 1/C], [-1/L, -R/L]])  # A
B_ = sp.Matrix([[0], [1/L]])  # B
C_ = sp.Matrix([[1, 0], [-1, -R]])  # C
D_ = sp.Matrix([[0], [1]])  # D
x = sp.Matrix([[v_C(t)], [i_L(t)]])  # x
u = sp.Matrix([[V_S(t)]])  # u
y = sp.Matrix([[v_C(t)], [v_L(t)]])  # y
```

The input and initial conditions can be encoded as follows:

```
u_subs = {V_S(t): 10}
ics = {v_C(0): 0, i_L(0): 0}
```

The set of first-order ODEs comprising the state equation can be defined as follows:

```
odes = x.diff(t) - A_*x - B_*u
print(odes)
```

$$\left[\begin{matrix}\frac{d}{dt} v_C(t) - \frac{i_L(t)}{C}\\ \frac{d}{dt} i_L(t) + \frac{R i_L(t)}{L} - \frac{V_S(t)}{L} + \frac{v_C(t)}{L}\end{matrix}\right]$$

```
x_sol = sp.dsolve(list(odes.subs(u_subs)), list(x), ics=ics)
```

The symbolic solutions for $\boldsymbol{x}(t)$ are lengthy expressions, so we don't print them here. Now we can compute the output $\boldsymbol{y}(t)$ from equation (4.24b) as follows:

```
x_sol_dict = {}  # Initialize
for eq in x_sol:
    x_sol_dict[eq.lhs] = eq.rhs  # Make a dict of solutions for subs
y_sol = (C_*x + D_*u).subs(x_sol_dict)  # Subs into output equation

# We will graph the output for the following set of parameters:
```

```
params = {
    R: 50,  # (Ohms)
    L: 10e-6,  # (H)
    C: 1e-9, # (F)
}
```

Create a numerically evaluable version of each function as follows:

```
v_C_ = sp.lambdify(
    t, y_sol[0].subs(params).subs(u_subs), modules="numpy"
)
v_L_ = sp.lambdify(
    t, y_sol[1].subs(params).subs(u_subs), modules="numpy"
)
```

Graph each solution as follows:

```
t_ = np.linspace(0, 0.000002, 201)
fig, axs = plt.subplots(2, sharex=True)
axs[0].plot(t_, v_C_(t_))
axs[1].plot(t_, v_L_(t_))
axs[1].set_xlabel("Time (s)")
axs[0].set_ylabel("$v_C(t)$ (rad/s)")
axs[1].set_ylabel("$v_L(t)$ (A)")
plt.show()
```

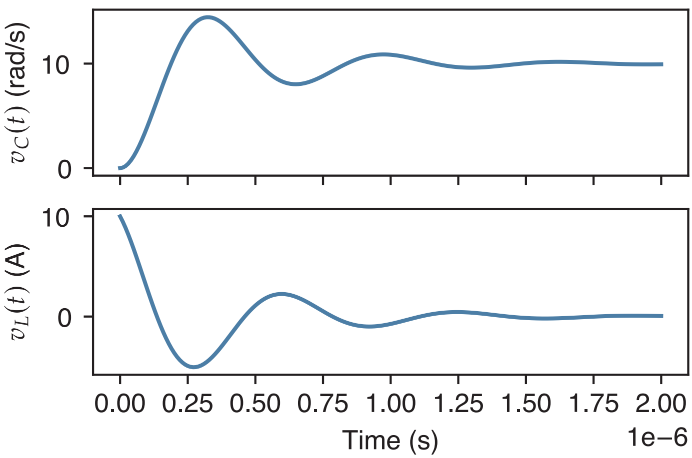

Figure S4.4. The state response to a 10 V step input.

The output equation is trivial in this case, yielding only the state variable $\Omega_J(t)$, for which we have already solved. Therefore, we have completed the analysis.