Example 3.2

February 20, 2025

Gabe Morris

[1]:
```
# npb
import sympy as sp
import numpy as np
import matplotlib.pyplot as plt

plt.style.use('../maroon_ipynb.mplstyle')
```

Example 3.2

Solve the following ODE with $f(t) = 200u(t)$ and $\dot{x}(0) = v_0$. All other initial conditions are zero.

$$2\ddddot{x} + 14\dddot{x} + 46.5\ddot{x} + 127\dot{x} + 148x = f(t)$$

Important Note with Sympy

One of the major caveats of `sympy` is that you should never include a decimal value in a symbolic expression. This may lead to errors depending on what you are trying to accomplish. You can work around this by representing the decimal value as a fraction or using `Rational` from `sympy`.

For example, let's say you want to evaluate $\frac{0.1}{0.3}x$.

[2]:
```
# do not do this
x = sp.Symbol('x')
expr = 0.1/0.3*x
expr
```

[2]: $0.333333333333333x$

This might seem ok, but if you expand that floating point value, you will see floating point error. You can adjust the amount of decimals that `sympy` will output with the `n()` method.

[3]:
```
expr.n(20)
```

[3]: $0.33333333333333337034x$

Here is the fix using `Rational`.

[4]:
```
numerator = sp.Rational(1, 10)
numerator  # this will correctly represent 0.1 as a fraction
```

[4]: $\frac{1}{10}$

[5]:
```python
denominator = sp.Rational(3, 10)
expr_correct = numerator/denominator*x
expr_correct
```

[5]: $\frac{x}{3}$

[6]:
```python
expr_correct.n(20)
```

[6]: $0.33333333333333333333x$

Solving the Easy Way

The best way to solve this with `sympy` would be to use the `dsolve` function.

[7]:
```python
t, v0 = sp.symbols('t v0', real=True)  # somtimes it's necessary to specify␣
↪that the symbol is real
x = sp.Function('x')(t)
eq = sp.Eq(2*x.diff(t, 4) + 14*x.diff(t, 3) + sp.Rational(93, 2)*x.diff(t, 2) +␣
↪127*x.diff(t) + 148*x, 200*sp.Heaviside(t))
eq
```

[7]:

$$148x(t) + 127\frac{d}{dt}x(t) + \frac{93\frac{d^2}{dt^2}x(t)}{2} + 14\frac{d^3}{dt^3}x(t) + 2\frac{d^4}{dt^4}x(t) = 200\theta\left(t\right)$$

[8]:
```python
sol = sp.dsolve(eq, ics={
    x.subs(t, 0): 0,
    x.diff(t, 1).subs(t, 0): v0,
    x.diff(t, 2).subs(t, 0): 0,
    x.diff(t, 3).subs(t, 0): 0
})
sol
```

[8]:

$$x(t) = \left(-\frac{9v_0}{34} + \frac{10\theta\left(t\right)}{17}\right)e^{-4t} + \left(\frac{53v_0}{90} - \frac{20\theta\left(t\right)}{9}\right)e^{-2t} + \left(\left(-\frac{248v_0}{765} + \frac{1600\theta\left(t\right)}{5661}\right)\cos\left(3t\right) + \left(\frac{244v_0}{765} - \frac{3680\theta\left(t\right)}{5661}\right)\right.$$

$$\frac{50\theta\left(t\right)}{37}$$

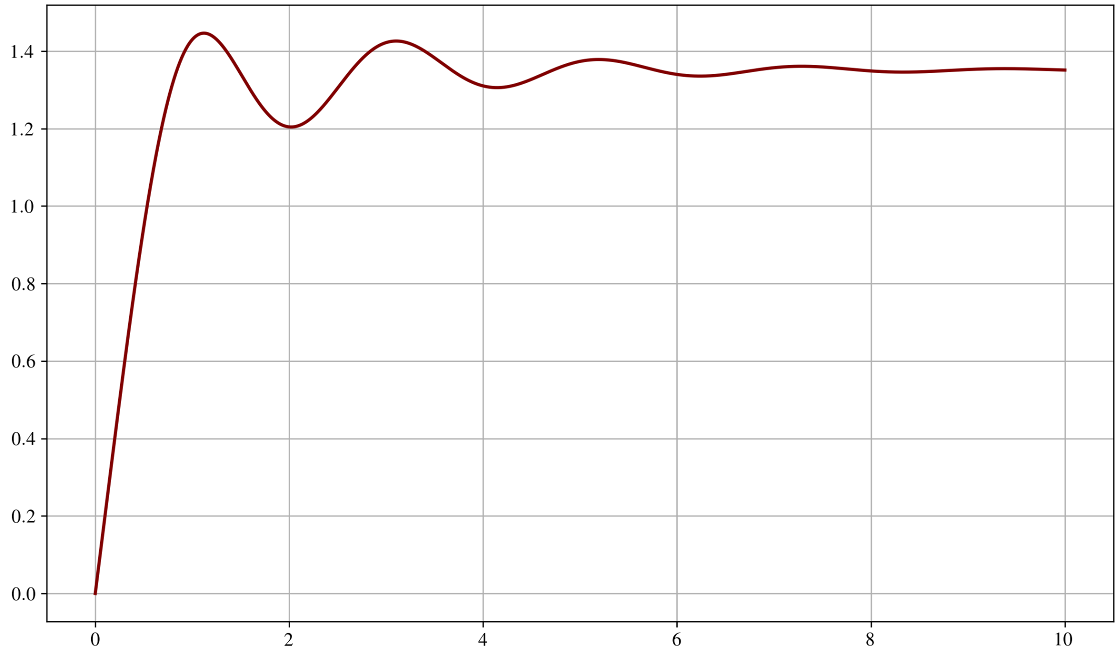
It is very nice that it was able to solve this and retain the v_0 symbol in the solution. We can bring this to the numerical world with `lambdify`.

[9]:
```python
# Substitute v0 with 2
sol_sub = sol.rhs.subs(v0, 2)

t_array = np.linspace(0, 10, 1000)
x_lamb = sp.lambdify(t, sol_sub, modules='numpy')

fig, ax = plt.subplots()
ax.plot(t_array, x_lamb(t_array))
```

```
plt.show()
```



Solving with Laplace Transforms

[10]:
```
# Putting the original equation into the Laplace domain
s = sp.Symbol('s', real=True)
eq_s = sp.Eq(sp.laplace_transform(eq.lhs, t, s)[0], sp.laplace_transform(eq.
    ↪rhs, t, s)[0])
eq_s
```

[10]:

$$2s^4\mathcal{L}_t\left[x(t)\right](s) + 14s^3\mathcal{L}_t\left[x(t)\right](s) - 2s^3x(0) + \frac{93s^2\mathcal{L}_t\left[x(t)\right](s)}{2} - 14s^2x(0) - 2s^2\left.\frac{d}{dt}x(t)\right|_{t=0} + 127s\mathcal{L}_t\left[x(t)\right](s) - \frac{93sx(0)}{2} - 14s\left.\frac{d}{dt}x(t)\right|_{t=0} - 2s\left.\frac{d^2}{dt^2}x(t)\right|_{t=0} + 148\mathcal{L}_t\left[x(t)\right](s) - 127x(0) - \frac{93\left.\frac{d}{dt}x(t)\right|_{t=0}}{2} - 14\left.\frac{d^2}{dt^2}x(t)\right|_{t=0} - 2\left.\frac{d^3}{dt^3}x(t)\right|_{t=0} = \frac{200}{s}$$

In the past, `sympy`'s laplace transforms would not apply the rule with the derivatives, but this changed in 2023 as you can clearly see the s terms.

[11]:
```
# Applying the initial conditions
eq_s = eq_s.subs([
    (x.subs(t, 0), 0),
    (x.diff(t, 1).subs(t, 0), v0),
    (x.diff(t, 2).subs(t, 0), 0),
```

```
    (x.diff(t, 3).subs(t, 0), 0)
])
eq_s
```

[11]:

$$2s^4\mathcal{L}_t\left[x(t)\right](s) + 14s^3\mathcal{L}_t\left[x(t)\right](s) - 2s^2v_0 + \frac{93s^2\mathcal{L}_t\left[x(t)\right](s)}{2} - 14sv_0 + 127s\mathcal{L}_t\left[x(t)\right](s) - \frac{93v_0}{2} + 148\mathcal{L}_t\left[x(t)\right](s) = \frac{200}{s}$$

[12]:
```
# Solving for X(s)
X_s = sp.solve(eq_s, sp.laplace_transform(x, t, s)[0])[0]
X_s
```

[12]:

$$\frac{4s^3v_0 + 28s^2v_0 + 93sv_0 + 400}{s\left(4s^4 + 28s^3 + 93s^2 + 254s + 296\right)}$$

Though I don't think there is a need, you can still use the partial fraction form if you wanted to do this by hand. Sometimes, `sympy` will solve the inverse laplace transform better in this form, but I haven't had any issues with inverse laplace transforms for a while.

[13]:
```
X_s_partial = sp.apart(X_s, s)  # must include the 's' because we have v0 in the expression
X_s_partial
```

[13]:

$$-\frac{32\cdot\left(1147sv_0 - 1000s - 2812v_0 + 6400\right)}{28305\cdot\left(4s^2 + 4s + 37\right)} - \frac{9v_0 - 20}{34\left(s + 4\right)} + \frac{53v_0 - 200}{90\left(s + 2\right)} + \frac{50}{37s}$$

Pro tip

Let's say we wanted to extract the characteristic polynomial from the above $X(s)$ expression and find its roots. First off, it would have been better to leave $f(t)$ in its symbolic form and find the transfer function, but we can get around this with what we have already done by dividing out $f(t)$.

[14]:
```
# Get transfer function
T_s = X_s/(200/s)
T_s
```

[14]:

$$\frac{4s^3v_0 + 28s^2v_0 + 93sv_0 + 400}{200\cdot\left(4s^4 + 28s^3 + 93s^2 + 254s + 296\right)}$$

It may not be in the form that we would perceive as the characteristic polynomial, but it's roots will still be the same.

[15]:
```
# Use sp.fraction to get the numerator and denominator
num, den = sp.fraction(T_s)
den
```

[15]:

$$800s^4 + 5600s^3 + 18600s^2 + 50800s + 59200$$

[16]:
```
cp = den/400  # returns original characteristic polynomial
cp
```

[16]:

$$2s^4 + 14s^3 + \frac{93s^2}{2} + 127s + 148$$

[17]:
```
# you can use sp.solve to get the roots
# or you can create a polynomial object and use the nroots method
cp_poly = sp.Poly(cp, s)
roots = cp_poly.nroots()
display(*roots)
```

-4.0

-2.0

$-0.5 - 3.0i$

$-0.5 + 3.0i$

Inverse Laplace Transform

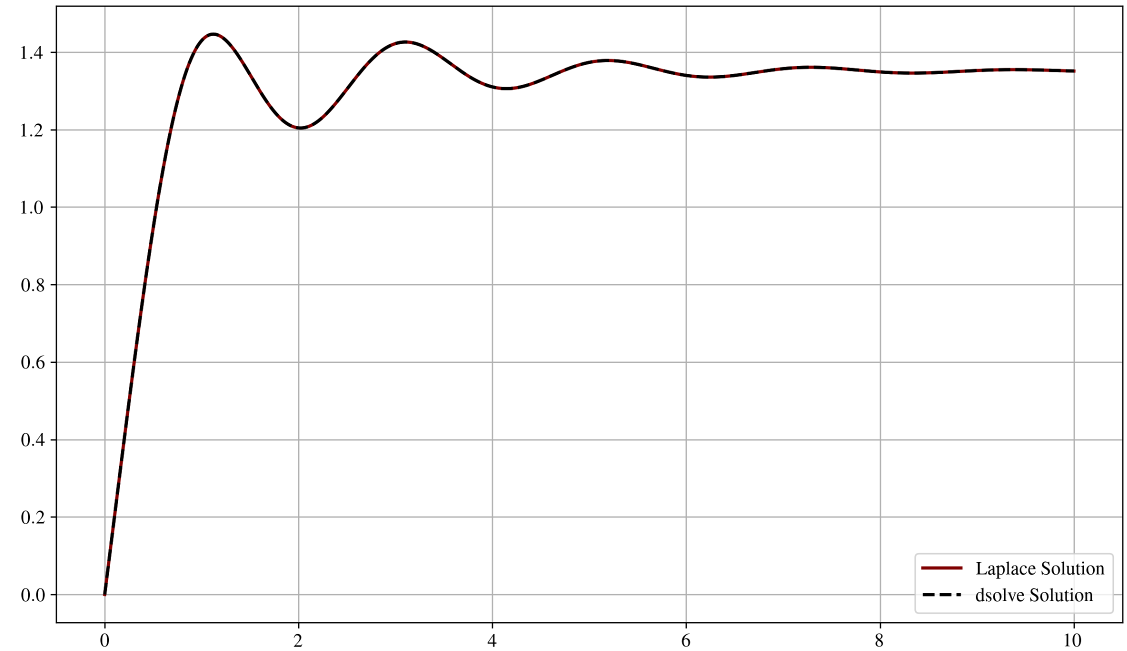
Going back to solving, all we have to do now is take the inverse laplace transform.

[18]:
```
sol = sp.inverse_laplace_transform(X_s, s, t)
sol
```

[18]:
$$\left(\frac{10}{17} - \frac{9v_0}{34}\right) e^{-4t}\theta\left(t\right)+\left(\frac{53v_0}{90} - \frac{20}{9}\right) e^{-2t}\theta\left(t\right)+\left(\left(\frac{1600}{5661} - \frac{248v_0}{765}\right) e^{-\frac{t}{2}}\cos\left(3t\right) + \left(\frac{244v_0}{765} - \frac{3680}{5661}\right) e^{-\frac{t}{2}}\sin\left(3t\right)\right)\theta\left(t\right)$$
$$\frac{50\theta\left(t\right)}{37}$$

[19]:
```
# Substitute v0 with 2
# Let's see if it's the same as the dsolve solution
x_t = sol.subs(v0, 2)
x_t_lamb = sp.lambdify(t, x_t, modules='numpy')

fig, ax = plt.subplots()
ax.plot(t_array, x_t_lamb(t_array), label='Laplace Solution')
ax.plot(t_array, x_lamb(t_array), ls='--', label='dsolve Solution')
ax.legend()
plt.show()
```



A couple of things to note about this result:

- The dominant root is the root whose real part has the smallest magnitude. This results in the largest time constant. The time constant then for this ODE is $-\frac{1}{-0.5} = 2$. Notice that the solution arrives close to steady state at $4 \cdot 2 = 8$ seconds.
- Since our characteristic polynomial has imaginary roots, we see oscillation. We can recognize that the oscillations occur at $3\,rad/s$. The period in seconds is $2\pi/3 \approx 2.1\,s$, and we can see that the peak to peak distance in the output matches this value.