```
[(3/2 - sqrt(17)/2,
  1,
  [Matrix([
   [-sqrt(17)/4 - 3/4],
   [                1]])]),
 (3/2 + sqrt(17)/2,
  1,
  [Matrix([
   [-3/4 + sqrt(17)/4],
   [                1]])])]
```

4.6 Calculus

Engineering analysis regularly includes calculus. Derivatives with respect to time and differential equations (i.e., equations including derivatives) are the key mathematical models of rigid-body mechanics (e.g., statics and dynamics), solid mechanics (e.g., mechanics of materials), fluid mechanics, heat transfer, and electromagnetism. Integration is necessary for solving differential equations and computing important quantities of interest. Limits and series expansions are frequently used to in the analytic process to simplify equations and to estimate unkown quantities. In other words, calculus is central to the enterprise of engineering analysis.

4.6.1 Derivatives

In SymPy, it is possible to compute the derivative of an expression using the `diff()` function and method, as follows:

```
x, y = sp.symbols("x, y", real=True)
expr = x**2 + x*y + y**2
expr.diff(x)  # Or sp.diff(expr, x)
expr.diff(y)  # Or sp.diff(expr, y)
```

$2x + y$

$x + 2y$

Higher-order derivatives can be computed by adding the corresponding integer, as in the following second derivative:

```
expr.diff(x, 2)  # Or sp.diff(expr, x, 2)
```

2

We can see that the partial derivative is applied to a multivariate expression. The differentiation can be mixed, as well, as in the following example:

```
expr = x * y**2/(x**2 + y**2)
expr.diff(x, 1, y, 2).simplify()  # ∂³/∂x∂y²
```

$$\frac{2x^2\left(-x^4+14x^2y^2-9y^4\right)}{x^8+4x^6y^2+6x^4y^4+4x^2y^6+y^8}$$

The option `evaluate=False` will leave the derivative unevaluated until the `doit()` method is called, as in the following example:

```
expr = sp.sin(x)
expr2 = expr.diff(x, evaluate=False); expr2
expr2.doit()
```

$$\frac{d}{dx}\sin(x)$$
$$\cos(x)$$

The derivative of an undefined function is left unevaluated, as in the following case:

```
f = sp.Function("f", real=True)
expr = 3*f(x) + f(x)**2
expr.diff(x)
```

$$2f(x)\frac{d}{dx}f(x)+3\frac{d}{dx}f(x)$$

As we can see, the chain rule of differentiation was applied automatically.

Differentiation works element-wise on matrices and vectors, just as it works mathematically. For instance,

```
v = sp.Matrix([[x**2], [x*y]])
v.diff(x)
```

$$\begin{bmatrix}2x\\ y\end{bmatrix}$$

4.6.2 Integrals

To a symbolic integral in SymPy, use the `integrate()` function or method. For an indefinite integral, pass only the variable over which to integrate, as in

```
x, y = sp.symbols("x, y", real=True)
expr = x + y
expr.integrate(x)  # Or sp.integrate(expr, x); ∫ x + y dx
```

$$\frac{x^2}{2}+xy$$

Note that no constant of integration is added, so you may need to add your own.

The definite integral can be computed by providing a triple, as in the following example,

```
sp.integrate(expr, (x, 0, 3))  # ∫_0^3 x + y dx
sp.integrate(expr, (x, 1, y))  # ∫_1^y x + y dx
```

↳ $3y + \frac{9}{2}$

↳ $\frac{3y^2}{2} - y - \frac{1}{2}$

Multiple integrals can be computed in a similar fashion, as in the following examples:

```
sp.integrate(expr, (x, 0, 4), (y, 2, 3))  # ∫_2^3 ∫_0^4 x + y dxdy
```

↳ 18

To create an unevaluated integral object, use the `sp.Integral()` constructor. To evaluate an unevaluated integral, use the `doit()` method, as follows:

```
expr2 = sp.Integral(expr, x); expr2  # Unevaluated
expr2.doit()  # Evaluate
```

↳ $\int (x + y)\, dx$

↳ $\frac{x^2}{2} + xy$

Integration works over piecewise functions, as in the following example:

```
f = sp.Piecewise((0, x < 0), (1, x >= 0)); f
sp.integrate(f, (x, -5, 5))
```

↳ $\begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{otherwise} \end{cases}$

↳ 5

The `integrate()` function and method is very powerful, but it may not be able to integrate some functions. In such cases, it returns an unevaluated integral.

4.6.3 Limits

In SymPy, a limit can be computed via the `limit()` function and method. The $\lim_{x \to 0}$ can be computed as follows:

```
sp.limit(sp.tanh(x)/x, x, 0)  # lim_{x→0} tanh(x)/x
```

↳ 1

The limit to infinity or negative infinity can be denoted using the `sp.oo` symbol, as follows:

```
sp.limit(2 - x * sp.exp(-x), x, sp.oo)  # lim_{x→∞}(1 - xe^{-x})
```

↳ 2

The limit can be left unevaluated using the `sp.Limit()` constructor, as follows:

```
lim = sp.Limit(2 - x * sp.exp(-x), x, sp.oo); expr  # Unevaluated
lim.doit()  # Evaluate
```

$x + y$
2

The limit can be taken from a direction using the optional fourth argument, as follows:

```
expr = 1/x
lim_neg = sp.Limit(expr, x, 0, "-"); lim_neg
lim_pos = sp.Limit(expr, x, 0, "+"); lim_pos
lim_neg.doit()
lim_pos.doit()
```

$\lim_{x \to 0^-} \frac{1}{x}$
$\lim_{x \to 0^+} \frac{1}{x}$
$-\infty$
∞

4.6.4 Taylor Series

A Taylor series (i.e., Taylor expansion) is an infinite power series approximation of an infinitely differentiable function near some point. For a function $f(x)$, the Taylor series at point x_0 is given by

$$\sum_{n=0}^{\infty} \frac{f^{(n)}}{n!}(x - x_0)^n = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \cdots .$$

We often represent terms with power order m and greater with the **big-O notation** $O((x - x_0)^m)$. For instance, for an expansion about $x_0 = 0$,

$$\sum_{n=0}^{\infty} \frac{f^{(n)}}{n!}(x)^n = f(0) + f'(x_0)(x - x_0) + O(x^2).$$

In SymPy, the Taylor series can be found via the `series()` function or method. For instance,

```
f = sp.sin(x)
f.series(x0=0, n=4)  # Or sp.series(f, x0=0, n=4)
```

$x - \frac{x^3}{6} + O\left(x^4\right)$

The `sp.O()` function, which appears in this result, automatically absorbs higher-order terms. For instance,

```
x**2 + x**4 + x** 5 + sp.O(x**4)
```

$x^2 + O\left(x^4\right)$

To remove the `sp.O()` function from an expression, call the `removeO()` method, as follows:

```
f.series(x0=0, n=4).removeO()
```

$$-\frac{x^3}{6}+x$$

Removing the higher-order terms is frequently useful when we would like to use the nth-order **Taylor polynomial**, a truncated Taylor series, as an approximation of a function.

4.7 Solving Ordinary Differential Equations

Engineering analysis regularly includes the solution of differential equations. **Differential equations** are those equations that contain derivatives. An **ordinary differential equation (ODE)** is a differential equation that contains only ordinary, as opposed to partial, derivatives. A **linear ODE**—one for which constant multiples and sums of solutions are also solutions—is an important type that represent **linear, time-varying (LTV) systems**. For this class of ODEs, it has been proven that for a set of initial conditions, a unique solution exists (Kreyszig 2010; p. 108).

A **constant-coefficient, linear ODE** can represent **linear, time-invariant (LTI) systems**. An LTV or LTI system model can be represented as a scalar nth-order ODE, or as a system of n 1st-order ODEs. As a scalar nth-order linear ODE, with independent time variable t, output function $y(t)$, forcing function $f(t)$, and constant coefficients a_i, has the form

$$y^{(n)}(t)+a_{n-1}y^{(n-1)}(t)+\cdots a_1y'(t)+a_0y(t)=f(t). \tag{4.23}$$

The forcing function $f(t)$ can be written as a linear combination of derivatives of the input function $u(t)$ with $m+1\leq n+1$ constant coefficients b_j, as follows:

$$f(t)=b_mu^{(m)}(t)+b_{m-1}u^{(m-1)}(t)+\cdots+b_1u'(t)+b_0u(t).$$

Alternatively, the same LTI system model can be represented by a system of n 1st-order ODEs, which can be written in vector form as

$$\boldsymbol{x}'(t)=\boldsymbol{A}\boldsymbol{x}(t)+\boldsymbol{B}\boldsymbol{u}(t) \tag{4.24a}$$

$$\boldsymbol{y}(t)=\boldsymbol{C}\boldsymbol{x}(t)+\boldsymbol{D}\boldsymbol{u}(t), \tag{4.24b}$$

where $\boldsymbol{x}(t)$ is called the state vector, $\boldsymbol{u}(t)$ is called the input vector, and $\boldsymbol{y}(t)$ is called the output vector (they are actually vector-valued functions of time), and $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$, and $\boldsymbol{D}$ are matrices containing constants derived from system parameters (e.g., a mass, a spring constant, a capacitance, etc.). Equation (4.24) is called an LTI **state-space model**, and it is used to model a great many engineering systems.

Solving ODEs and systems of ODEs is a major topic of mathematical engineering analysis. It is typically the primary topic of one required course and a secondary