Table 4.1: Elementary mathematical functions in SymPy.

Kind	SymPy Functions (`sp.` prefix suppressed)
Complex	`Abs()`, `arg()`, `conjugate()`, `im()`, `re()`, `sign()`
Trigonmetric	`sin()`, `cos()`, `tan()`, `sec()`, `csc()`, `cot()`
Inverse Trigonometric	`asin()`, `acos()`, `atan()`, `atan2()`, `asec()`, `acsc()`, `acot()`
Hyperbolic	`sinh()`, `cosh()`, `tanh()`, `coth()`, `sech()`, `csch()`
Inverse Hyperbolic	`asinh()`, `acosh()`, `atanh()`, `acoth()`, `asech()`
Integer	`ceiling()`, `floor()`, `frac()` `get_integer_part()`
Exponential	`exp()`, `log()`
Miscellaneous	`Min()`, `Max()`, `root()`, `sqrt()`

In rare cases, we must define a **custom function**; that is, a subclass of the `sp.Function` class. Such a function needs to have its behavior thoroughly defined. Once it is completed, it should behave just as built-in functions like `sp.sin()`. For a tutorial on writing custom functions, see SymPy Development Team (2023d).

4.2 Manipulating Symbolic Expressions

In engineering symbolic analysis, the need to manipulate, often algebraically, mathematical expressions arises constantly. SymPy has several powerful tools for manipulating symbolic expressions, the most useful of which we will consider here.

4.2.1 The `simplify()` Function and Method

A built-in Sympy function and method, `sp.simplify()`, is a common SymPy tool for manipulation because simplification is often what we want. Recall that some basic simplification occurs automatically; however, in many cases this automatic simplification is insufficient. Applying `sp.simplify()` typically results in an expression as simple as or simpler than its input; however, the precise meaning of "simpler" is quite vague, which can lead to frustrating cases in which a version of an expression we consider to be simpler is not chosen by the `sp.simplify()` algorithm. In such cases, we will often use the more manual techniques considered later in this section.

The predicates (i.e., assumptions) used to define the symbolic variables and functions that appear in a symbolic expression are respected by `sp.simplify()`. Consider the following example:

```
x = sp.symbols("x", real=True)
e0 = (x**2 + 2*x + 3*x)/(x**2 + 2*x); e0  # For display
e0.simplify()  # Returns simplified expression, leaves e0 unchanged
```

$$\frac{x^2+5x}{x^2+2x}$$
$$\frac{x+5}{x+2}$$

Note that `e0` was slightly simplified automatically. The `simplify()` method further simplified by canceling an `x`. The use of the method does not affect the object, so it the same as the use of the function.

There are a few "knobs" to turn in the form of optional arguments to `sp.simplify()`:

- `measure` (default: `sp.count_ops()`): A function that serves as a heuristic complexity metric. The default `sp.count_ops()` counts the operations in the expression.
- `ratio` (default: `1.7`): The maximum ratio of the measures, output `out` over input `inp`, `measure(out)/measure(inp)`. Anything over `1` allows the output to be potentially more complex than the input, but it may still be simpler because the metric is just a heuristic.
- `rational` (default: **`False`**): By default (**`False`**), floating-point numbers are left alone. If `rational=`**`True`**, floating-point numbers are recast as rational numbers. If `rational=`**`None`**, floating-point numbers are recast as rational numbers during simplification, but recast to floating-point numbers in the result.
- `inverse` (default: **`False`**): If **`True`**, allows inverse functions to be cancelled in any order without knowing if the inner argument falls in the domain for which the inverse holds.[3] For instance, this allows $\arccos(\cos x) \to x$ without knowing if $x \in [0, \pi]$.
- `force` (default: **`False`**): If **`True`**, predicates (assumptions) of the variables will be ignored.

4.2.2 Polynomial and Rational Expression Manipulation

Here we consider a few SymPy functions and methods that manipulate polynomials and rational expressions.

4.2.2.1 The `expand()` Function and Method

The `expand()` function and method expresses a polynomial in the canonical form of a sum of monomials. A monomial is a polynomial with exactly one additive term. For instance,

```
sp.expand((x + 3)**2)  ## Using the real x from above
```

3. The usual way of defining the inverse $y = \arccos x$ is to retrict y in $x = \cos y$ to $[0, \pi]$. This is because cos is not one-to-one (e.g., $\cos 0 = \cos 2\pi = 1$), so its domain must be restricted for a proper inverse to exist. The conventional choice of domain restriction to $[0, \pi]$ is called the selection of a principal branch.

$x^2 + 6x + 9$

We can also expand a numerator or denominator without expanding the entire expression, as follows for $(x+3)^2/(x-2)^2$:

```
frac = (x + 3)**2/(x - 2)**2
frac.expand()
frac.expand(numer=True)
frac.expand(denom=True)
frac.expand(numer=True).expand(denom=True)
```

$$\frac{x^2}{x^2-4x+4} + \frac{6x}{x^2-4x+4} + \frac{9}{x^2-4x+4}$$

$$\frac{x^2+6x+9}{(x-2)^2}$$

$$\frac{(x+3)^2}{x^2-4x+4}$$

$$\frac{x^2+6x+9}{x^2-4x+4}$$

There are several additional options for `expand()`, including:

- `mul` (default: **`True`**): If **`True`**, distributes multiplication over addition (e.g., $5(x+1) \to 5x+5$.
- `multinomial` (default: **`True`**): If **`True`**, expands multinomial (polynomial that is not a monomial) terms into sums of monomials (e.g., $(x+y)^2 \to x^2 + 2xy + y^2$).
- `power_exp` (default: **`True`**): If **`True`**, expands sums in exponents to products of exponentials (e.g., $e^{3+x} \to e^3 e^x$).
- `log` (default: **`True`**): If **`True`**, split log products into sums and extract log exponents to multiplicative constants (e.g., for $x, y > 0$, $\ln(x^3 y) \to 3\ln x + \ln y$).
- `deep` (default: **`True`**): If **`True`**, expands all levels of the expression tree; if **`False`**, expands only the top level (e.g., $x(x+(y+1)^2) \to x^2 + x(y+1)^2$).
- `complex` (default: **`False`**): If **`True`**, collect real and imaginary parts (e.g., $x + y \to \Re(x) + \Re(y) + j(\Im(x) + \Im(y))$).
- `func` (default: **`False`**): If **`True`**, expand nonpolynomial functions (e.g., for the gamma function Γ, $\Gamma(x+2) \to x^2\Gamma(x) + x\Gamma(x)$).
- `trig` (default: **`False`**): If **`True`**, expand trigonometric functions (e.g., $\sin(x+y) \to \sin x \cos y - \sin y \cos x$).

4.2.2.2 The `factor()` Function and Method The `factor()` function and method returns a factorization into irreducibles factors. For polynomials, this is the reverse of `expand()`. Irreducibility of the factors is guaranteed for polynomials. Consider the following polynomial example:

```
x, y = sp.symbols("x, y", real=True)
e0 = (x + 1)**2 * (x**2 + 2*x*y + y**2); e0
e0.expand()
e0.expand().factor()
```

$$(x+1)^2\left(x^2+2xy+y^2\right)$$
$$x^4+2x^3y+2x^3+x^2y^2+4x^2y+x^2+2xy^2+2xy+y^2$$
$$(x+1)^2(x+y)^2$$

Factorization can also be performed over nonpolynomial expressions, as in the following example:

```
e1 = sp.sin(x) * (sp.cos(x) + sp.sin(x))**2; e1  # Using above real x
e1.expand()
e1.expand().factor()
```

$$(\sin(x)+\cos(x))^2\sin(x)$$
$$\sin^3(x)+2\sin^2(x)\cos(x)+\sin(x)\cos^2(x)$$
$$(\sin(x)+\cos(x))^2\sin(x)$$

There are two options of note:

- `deep` (default: **`False`**): If **`True`**, inner expression tree elements will also be factored (e.g., $\exp(x^2+4x+4) \rightarrow \exp((x+2)^2)$).
- `fraction` (default: **`True`**): If **`True`**, rational expressions will be combined.

An example of the latter option is given here:

```
e2 = x - 5*sp.exp(3 - x); e2  # Using real x from above
e2.factor(deep=True)
e2.factor(deep=True, fraction=False)
```

$$x-5e^{3-x}$$
$$\left(xe^x-5e^3\right)e^{-x}$$
$$x-5e^3e^{-x}$$

4.2.2.3 The `collect()` Function and Method The `collect()` function and method returns an expression with specific terms collected. For instance,

```
x, y, a, b = sp.symbols("x, y, a, b", real=True)
e3 = a * x + b * x * y + a**2 * x**2 + 3 * y**2 + x * y + 8; e3
e3.collect(x)
```

$$a^2x^2+ax+bxy+xy+3y^2+8$$
$$a^2x^2+x(a+by+y)+3y^2+8$$

More complicated expressions can be collected as well, as in the following example:

```
e4 = a*sp.cos(4*x) + b*sp.cos(4*x) + b*sp.cos(6*x) + a * sp.sin(x); e4
e4.collect(sp.cos(4*x))
```

$$a\sin(x) + a\cos(4x) + b\cos(4x) + b\cos(6x)$$
$$a\sin(x) + b\cos(6x) + (a+b)\cos(4x)$$

Derivatives of an undefined symbolic function, as would appear in a differential equation, can be collected. If the function is passed to `collect()`, as in the following example, it and its derivatives are collected:

```
f = sp.Function("f")(x)  ## Applied undefined function
e5 = a*f.diff(x, 2) + a**2*f.diff(x) + b**2*f.diff(x) + a**3*f; e5
e5.collect(f)
```

$$a^3 f(x) + a^2 \frac{d}{dx} f(x) + a \frac{d^2}{dx^2} f(x) + b^2 \frac{d}{dx} f(x)$$
$$a^3 f(x) + a \frac{d^2}{dx^2} f(x) + \left(a^2 + b^2\right) \frac{d}{dx} f(x)$$

The `rcollect()` function (not available as a method) recursively applies `collect()`. For instance,

```
e6 = (a * x**2 + b*x*y + a*b*x)/(a*x**2 + b*x**2); e6
sp.rcollect(e6, x)  # Collects in numerator and denominator
```

$$\frac{abx + ax^2 + bxy}{ax^2 + bx^2}$$
$$\frac{ax^2 + x(ab + by)}{x^2(a+b)}$$

Before collection, an expression may need to be expanded via `expand()`.

4.2.2.4 The `cancel()` Function and Method The `cancel()` function and method will return an expression in the form p/q, where p and q are polynomials that have been expanded and have integer leading coefficients. This is typically used to cancel terms that can be factored from the numerator and denominator of a rational expression, as in the following example:

```
e7 = (x**3 - a**3)/(x**2 - a**2); e7
e7.cancel()
```

$$\frac{-a^3 + x^3}{-a^2 + x^2}$$
$$\frac{a^2 + ax + x^2}{a + x}$$

Note that there is an implicit assumption here that $x \neq a$. However, the cancelation is still valid for the limit as $x \to a$.

4.2.2.5 The `apart()` and `together()` Functions and Methods The `apart()` function and method returns a **partial fraction expansion** of a rational expression. A partial fraction expansion rewrites a ratio as a sum of a polynomial and one or

more ratios with irreducible denominators. It is of particular use for computing the inverse Laplace transform. The `together()` function is the complement of `apart()`. Here is an example of a partial fraction expansion:

```
s = sp.symbols("s")
e8 = (s**3 + 6*s**2 + 16*s + 16)/(s**3 + 4*s**2 + 10*s + 7); e8
e8.apart()  # Partial fraction expansion
e8.apart().together().cancel()  # Putting it back together
```

$$\frac{s^3 + 6s^2 + 16s + 16}{s^3 + 4s^2 + 10s + 7}$$

$$\frac{s+2}{s^2 + 3s + 7} + 1 + \frac{1}{s+1}$$

$$\frac{s^3 + 6s^2 + 16s + 16}{s^3 + 4s^2 + 10s + 7}$$

4.2.3 Trigonometric Expression Manipulation

As we saw in section 4.2.2, expressions including trigonometric terms can be manipulated with the SymPy functions and methods that are nominally for polynomial and rational expressions. In addition to these, considered here are two important SymPy functions and methods for manipulating expressions including trigonometric terms, with a focus on the trigonometric terms themselves.

4.2.3.1 The `trigsimp()` Function and Method The `trigsimp()` function and method attempts to simplify a symbolic expression via trigonometric identities. For instance, it will apply the double-angle formulas, as follows:

```
x = sp.symbols("x", real=True)
e9 = 2 * sp.sin(x) * sp.cos(x); e9
e9.trigsimp()
```

$$2\sin(x)\cos(x)$$

$$\sin(2x)$$

Here is a more involved expression:

```
e10 = sp.cos(x)**4 - 2*sp.sin(x)**2*sp.cos(x)**2 + sp.sin(x)**4; e10
e10.trigsimp()
```

$$\sin^4(x) - 2\sin^2(x)\cos^2(x) + \cos^4(x)$$

$$\frac{\cos(4x)}{2} + \frac{1}{2}$$

The hyperbolic trignometric functions are also handled by `trigsimp()`, as in the following example:

```
e11 = sp.cosh(x) * sp.tanh(x); e11
e11.trigsimp()
```

$\cosh(x)\tanh(x)$
$\sinh(x)$

4.2.3.2 The `expand_trig()` Function The `sp.expand_trig()` function applies the double-angle or sum identity in the expansive direction, opposite the direction of `trig_simp()`; that is,

```
e12 = sp.cos(x + y); e12
sp.expand_trig(e12)
```

$\cos(x+y)$
$-\sin(x)\sin(y)+\cos(x)\cos(y)$

4.2.4 Power Expression Manipulation

There are three important power identities:

$$x^a x^b = x^{a+b} \text{ for } x \neq 0, a, b \in \mathbb{C} \tag{4.1}$$

$$u^c v^c = (uv)^c \text{ for } u, v \geq 0 \text{ and } c \in \mathbb{R} \tag{4.2}$$

$$(z^d)^n = z^{dn} \text{ for } z, d \in \mathbb{C} \text{ and } n \in \mathbb{Z}. \tag{4.3}$$

Equations (4.1) to (4.3) are applied in several power expression simplification functions and methods considered here.

4.2.4.1 The `powsimp()` Function and Method The `powsimp()` function and method applies the identities of equations (4.1) and (4.2) from left-to-right (replacing the left pattern with the right). It will only apply the identity if it holds. Consider the following, applying equation (4.1):

```
x = sp.symbols("x", complex=True, nonzero=True)
a, b = sp.symbols("a, b", complex=True)
e13 = x**a * x**b; e13
e13.powsimp()
```

$x^a x^b$
x^{a+b}

Applying equation (4.2),

```
u, v = sp.symbols("u, v", nonnegative=True)
c = sp.symbols("c", real=True)
e14 = u**c * v**c; e14
e14.powsimp()
```

$u^c v^c$
$(uv)^c$

Under certain conditions (i.e., $c \in \mathbb{Q}$, a literal rational exponent), equation (4.2) is applied right-to-left automatically, so `powsimp()` appears to have no effect. For instance,

```
e15 = u**3 * v**3; e15
e15.powsimp()
```

$u^3 v^3$
$u^3 v^3$

For expressions for which the conditions for an identity does not hold, it can still be applied (at your own risk) via the `force=True` argument.

4.2.4.2 The `expand_power_exp()` and `expand_power_base()` Functions The `expand_power_exp()` function applies equation (4.1) from right-to-left (opposite of `powsimp()`), as follows:

```
e16 = x**(a + b); e16
sp.expand_power_exp(e16)
```

x^{a+b}
$x^a x^b$

Similarly, `expand_power_base()` applies equation (4.2) from right-to-left (opposite of `powsimp()`, as follows:

```
e17 = (u * v)**c; e17
sp.expand_power_base(e17)
```

$(uv)^c$
$u^c v^c$

Again, the identity will not be applied if its conditions do not hold for the expression; however, with the parameter `force=True`, it will be applied in any case.

4.2.4.3 The `powdenest()` Function The `powdenest()` function applies equation (4.3) from left-to-right. For instance,

```
z, d = sp.symbols("z, d", complex=True)
n = sp.symbols("n", integer=True)
e18 = (z**d)**n; e18
sp.powdenest(e18)
```

z^{dn}
z^{dn}

However, as we see from `e18`, the denesting is automatically applied. There may be situations in which `powdenest()` must still be applied manually.

4.2.5 Exponential and Logarithmic Expression Manipulation

For $x, y \geq 0$ and $n \in \mathbb{R}$, the following identities hold:

$$\log(xy) = \log(x) + \log(y) \tag{4.4}$$

$$\log(x^n) = n \log(x) \tag{4.5}$$

These can be applied with the `expand_log()` and `logcombine()` functions.

4.2.5.1 The `expand_log()` Function The `expand_log()` function applies equations (4.4) and (4.5) from left-to-right. In the following example, it applies equation (4.4):

```
x, y = sp.symbols("x, y", positive=True)
n = sp.symbols("n", real=True)
e19 = sp.log(x * y); e19
sp.expand_log(e19)
```

$$\log(xy)$$

$$\log(x) + \log(y)$$

In the following example, it applies equation (4.4):

```
e20 = sp.log(x**n); e20
sp.expand_log(e20)
```

$$\log(x^n)$$

$$n \log(x)$$

4.2.5.2 The `logcombine()` Function The `logcombine()` function applies equations (4.4) and (4.5) from right-to-left. In the following example, it applies equation (4.4):

```
e21 = sp.log(x) + sp.log(y); e21
sp.logcombine(e21)
```

$$\log(x) + \log(y)$$

$$\log(xy)$$

In the following example, it applies equation (4.4):

```
e22 = n * sp.log(x); e22
sp.logcombine(e22)
```

$$n \log(x)$$

$$\log(x^n)$$

4.2.6 Rewriting Expressions in Terms of Other Functions

At times, there are identities that can translate an expression in terms of one function (or set of functions) into an expression in terms of another function (or set of functions). In SymPy, the `rewrite()` method can perform this translation. For instance, Euler's formula, $e^{jx} = \cos x + j \sin x$ can be applied:

```
x = sp.symbols("x", complex=True)
e23 = sp.exp(1j * x); e23
e24 = e23.rewrite(sp.cos); e24  # Apply left-to-right
e24.rewrite(sp.exp)  # Apply right-to-left
```

$$e^{1.0ix}$$
$$i \sin(1.0x) + \cos(1.0x)$$
$$e^{1.0ix}$$

Here is an example with a hyperbolic trigonometric function:

```
e25 = sp.tanh(x); e25
e25.rewrite(sp.exp)
```

$$\tanh(x)$$
$$\frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Finally, consider the following example with trigonometric functions:

```
x, y = sp.symbols("x, y", real=True)
e26 = sp.tan(x + y)**2; e26
e26.rewrite(sp.cos)
```

$$\tan^2(x+y)$$
$$\frac{\cos^2\left(x+y-\frac{\pi}{2}\right)}{\cos^2(x+y)}$$

4.2.7 Substituting and Replacing Expressions

One expression can be substituted for another via a few different methods, the two most useful of which are considered here.

4.2.7.1 The `subs()` Method The `subs()` method returns a copy of an expression with specific subexpressions replaced. There are three ways to specify substitutions for an expression `expr`:

- `expr.subs(old, new)`, in which `old` is replaced with `new`
- `expr.subs(iterable)`, in which `iterable` (e.g., a `list`) contains `old`/`new` pairs like `[(old0, new0), (old1, new1), ...]`
- `expr.subs(dictionary)`, in which `dictionary` contains `old`/`new` pairs like `{old0: new0, old1: new1, ...}`

Consider the following simple examples:

```
x, y, z = sp.symbols("x, y, z")
sp.sqrt(x + y).subs(x, 5)
(x + y**2 + z).subs({x: z, y: 2*z})
```

$\sqrt{y+5}$

$4z^2 + 2z$

By default, when an ordered iterable like a `list` or `tuple` is provided, substitutions are performed in the order given, as in the following example:

```
(x + y).subs(((x, y), (y, z)))
```

$2z$

We see that the second substitution $y \to z$ is applied after the first, $x \to y$. The parameter `simultaneous`, by default **`False`**, can be passed as **`True`** so that new subexpressions are ignored by later substitutions, as in the following example:

```
(x + y).subs(((x, y), (y, z)), simultaneous=True)
```

$y + z$

For dictionary substitutions, which are unordered, a canonical ordering based on the number of operations is used for reproducibility. We do not recommend relying on this canonical ordering, so if the order of substitutions is important, we recommend using an ordered iterable.

If the substitutions result in a numerical value, it will by default remain a symbolic expression:

```
sp.srepr((x + y).subs(((x, 1), (y, 3))))
```

```
'Integer(4)'
```

To get a numeric type from the result, the `evalf()` method can be used:

```
(1/y).subs(y, 3.0).evalf(n=20)  # subs() first (20 decimal places)
(1/y).evalf(subs={y: 3.0}, n=20)  # evalfr() subs (20 decimal places)
```

```
0.33333333333333331483
0.33333333333333333333
```

Note that passing the substitutions to through `evalf()` can result in a more accurate representation, so this technique is preferred. We will later [TODO: ref] return to more powerful techniques for numerical evaluation that convert SymPy expressions to numerically evaluable functions.

4.2.7.2 The `replace()` Method The `replace()` method is similar to `subs()`, but it has matching capabilities. Common usage of the `replace()` method uses **wildcard variables** of class `sp.core.symbol.Wild` that match anything in a pattern. For instance,

```
w = sp.symbols("w", cls=sp.Wild)
expr = sp.sin(x) + sp.sin(3*x)**2; expr
expr.replace(sp.sin(w), sp.cos(w)/w)
```

$$\sin(x) + \sin^2(3x)$$

$$\frac{\cos(x)}{x} + \frac{\cos^2(3x)}{9x^2}$$

Note that the wildcard variable `w` was able to match both `x` and `3*x`, and that the the wildcard could be used in the new expression as well. In this example, and in general, these replacement rules are applied without head to their validity, so they must be used with caution. For more advanced usage, see the documentation on wildcard matching, SymPy Development Team (2023b; § 6 Symbol (`sympy.core.symbol`, `Wild` class)) and the documentation for replacement, SymPy Development Team (2023b; § Basic (`sympy.core.basic.Basic`, `replace()` method)).

Box 4.1 Further Reading

- SymPy Development Team (2023c), A tutorial introduction to simplification in SymPy
- SymPy Development Team (2023a), A tutorial on advanced SymPy expression manipulation, including information about expression trees
- SymPy Development Team (2023b; § Basic (`sympy.core.basic.Basic`, `subs()` method)), SymPy documentation on the `subs()` method
- SymPy Development Team (2023b; § Basic (`sympy.core.basic.Basic`, `replace()` method)), SymPy documentation on the `replace()` method, including more advanced usage
- SymPy Development Team (2023b; 6 Symbol (`sympy.core.symbol`, `Wild` class)), SymPy documentation on the `Wild` class, including more advanced pattern matching

4.3 Solving Equations Algebraically

Virtually every engineering analysis requires the algebraic solution of an equation or a, more generally, a **system of equations** (i.e., a set of equations) to be solved simultaneously. For the engineer, this set of equations typically encodes a set of design constraints, design heuristics, and physical laws. In general, a system S of m equations in n unknown variables $x_0, \cdots, x_{n-1} \in \mathbb{C}$ and with m functions $f_0, \cdots, f_{m-1}$ can be represented as the set

$$S = \begin{cases} f_0(x_0, \cdots, x_n) = 0 \\ \vdots \\ f_m(x_0, \cdots, x_n) = 0 \end{cases}.$$

A **solution** for S is an n-tuple of values for x_i that satisfies every equation in S. There are three possible cases for a given system S of equations:

1. The system S has no solutions.
2. The system S has exactly one solution, said to be **unique**.
3. The system S has more than one solution (potentially infinitely many).

For some systems, a solution exists, but cannot be expressed in a closed-form or symbolic ("analytic") way. For such systems, a numerical solution is appropriate (see chapter 5). In some cases (e.g., n linear, independent equations and n unknown variables), a unique solution is guaranteed to exist.

There are two high-level SymPy function for solving equations algebraically, `sp.solve()` and `sp.solveset()`. The former is older, but remains the more useful for us; the latter has a simpler interface and is somewhat more mathematically rigorous, but it is often difficult to use its results programmatically. We will focus on `sp.solve()`. Neither function guarantees that it will find a solution, even if it exists, except in special cases.

Representing an equation in SymPy can be done explicitly or an expression can be treated as one side of an equation, with the other side implicitly 0. In other words, the following are equivalent ways of defining the equation $x^2 - y^2 = 2$:

```
x, y = sp.symbols("x, y")
x**2 - y**2 - 2  # == 0 Implicit equation
sp.Eq(x**2 - y**2, 2)  # Explicit equation
```