See also:

collect (page 672), *collect_sqrt* (page 675), *rcollect* (page 674)

fraction

sympy.simplify.radsimp.**fraction**(*expr*, *exact=False*)

Returns a pair with expression's numerator and denominator. If the given expression is not a fraction then this function will return the tuple (expr, 1).

This function will not make any attempt to simplify nested fractions or to do any term rewriting at all.

If only one of the numerator/denominator pair is needed then use numer(expr) or denom(expr) functions respectively.

```
>>> from sympy import fraction, Rational, Symbol
>>> from sympy.abc import x, y
```

```
>>> fraction(x/y)
(x, y)
>>> fraction(x)
(x, 1)
```

```
>>> fraction(1/y**2)
(1, y**2)
```

```
>>> fraction(x*y/2)
(x*y, 2)
>>> fraction(Rational(1, 2))
(1, 2)
```

This function will also work fine with assumptions:

```
>>> k = Symbol('k', negative=True)
>>> fraction(x * y**k)
(x, y**(-k))
```

If we know nothing about sign of some exponent and `exact` flag is unset, then structure this exponent's structure will be analyzed and pretty fraction will be returned:

```
>>> from sympy import exp, Mul
>>> fraction(2*x**(-y))
(2, x**y)
```

```
>>> fraction(exp(-x))
(1, exp(x))
```

```
>>> fraction(exp(-x), exact=True)
(exp(-x), 1)
```

The `exact` flag will also keep any unevaluated Muls from being evaluated:

```
>>> u = Mul(2, x + 1, evaluate=False)
>>> fraction(u)
(2*x + 2, 1)
>>> fraction(u, exact=True)
(2*(x  + 1), 1)
```

Ratsimp

ratsimp

sympy.simplify.ratsimp.**ratsimp**(*expr*)

Put an expression over a common denominator, cancel and reduce.

Examples

```
>>> from sympy import ratsimp
>>> from sympy.abc import x, y
>>> ratsimp(1/x + 1/y)
(x + y)/(x*y)
```

ratsimpmodprime

sympy.simplify.ratsimp.**ratsimpmodprime**(*expr, G, *gens, quick=True, polynomial=False, **args*)

Simplifies a rational expression `expr` modulo the prime ideal generated by `G`. `G` should be a Groebner basis of the ideal.

Examples

```
>>> from sympy.simplify.ratsimp import ratsimpmodprime
>>> from sympy.abc import x, y
>>> eq = (x + y**5 + y)/(x - y)
>>> ratsimpmodprime(eq, [x*y**5 - x - y], x, y, order='lex')
(-x**2 - x*y - x - y)/(-x**2 + x*y)
```

If `polynomial` is `False`, the algorithm computes a rational simplification which minimizes the sum of the total degrees of the numerator and the denominator.

If `polynomial` is `True`, this function just brings numerator and denominator into a canonical form. This is much faster, but has potentially worse results.

References

[R762]

Trigonometric simplification

trigsimp

sympy.simplify.trigsimp.**trigsimp**(*expr*, ***opts*)

Returns a reduced expression by using known trig identities.

Parameters

method : string, optional

Specifies the method to use. Valid choices are:

- `'matching'`, default
- `'groebner'`
- `'combined'`
- `'fu'`
- `'old'`

If `'matching'`, simplify the expression recursively by targeting common patterns. If `'groebner'`, apply an experimental groebner basis algorithm. In this case further options are forwarded to `trigsimp_groebner`, please refer to its docstring. If `'combined'`, it first runs the groebner basis algorithm with small default parameters, then runs the `'matching'` algorithm. If `'fu'`, run the collection of trigonometric transformations described by Fu, et al. (see the `fu()` (page 712) docstring). If `'old'`, the original SymPy trig simplication function is run.

opts :

Optional keyword arguments passed to the method. See each method's function docstring for details.

Examples

```
>>> from sympy import trigsimp, sin, cos, log
>>> from sympy.abc import x
>>> e = 2*sin(x)**2 + 2*cos(x)**2
>>> trigsimp(e)
2
```

Simplification occurs wherever trigonometric functions are located.

```
>>> trigsimp(log(e))
log(2)
```

Using `method='groebner'` (or `method='combined'`) might lead to greater simplification.

The old trigsimp routine can be accessed as with method `method='old'`.

```
>>> from sympy import coth, tanh
>>> t = 3*tanh(x)**7 - 2/coth(x)**7
>>> trigsimp(t, method='old') == t
True
>>> trigsimp(t)
tanh(x)**7
```

Power simplification

powsimp

sympy.simplify.powsimp.**powsimp**(*expr*, *deep=False*, *combine='all'*, *force=False*, *measure=<function count_ops>*)

reduces expression by combining powers with similar bases and exponents.

Explanation

If deep is True then powsimp() will also simplify arguments of functions. By default deep is set to False.

If force is True then bases will be combined without checking for assumptions, e.g. sqrt(x)*sqrt(y) -> sqrt(x*y) which is not true if x and y are both negative.

You can make powsimp() only combine bases or only combine exponents by changing combine='base' or combine='exp'. By default, combine='all', which does both. combine='base' will only combine:

```
 a   a          a                          2x      x
x * y  =>  (x*y)   as well as things like 2   =>  4
```

and combine='exp' will only combine

```
 a   b       (a + b)
x * x  =>  x
```

combine='exp' will strictly only combine exponents in the way that used to be automatic. Also use deep=True if you need the old behavior.

When combine='all', 'exp' is evaluated first. Consider the first example below for when there could be an ambiguity relating to this. This is done so things like the second example can be completely combined. If you want 'base' combined first, do something like powsimp(powsimp(expr, combine='base'), combine='exp').

Examples

```
>>> from sympy import powsimp, exp, log, symbols
>>> from sympy.abc import x, y, z, n
>>> powsimp(x**y*x**z*y**z, combine='all')
x**(y + z)*y**z
>>> powsimp(x**y*x**z*y**z, combine='exp')
x**(y + z)*y**z
>>> powsimp(x**y*x**z*y**z, combine='base', force=True)
x**y*(x*y)**z
```

```
>>> powsimp(x**z*x**y*n**z*n**y, combine='all', force=True)
(n*x)**(y + z)
>>> powsimp(x**z*x**y*n**z*n**y, combine='exp')
n**(y + z)*x**(y + z)
>>> powsimp(x**z*x**y*n**z*n**y, combine='base', force=True)
(n*x)**y*(n*x)**z
```

```
>>> x, y = symbols('x y', positive=True)
>>> powsimp(log(exp(x)*exp(y)))
log(exp(x)*exp(y))
>>> powsimp(log(exp(x)*exp(y)), deep=True)
x + y
```

Radicals with Mul bases will be combined if combine='exp'

```
>>> from sympy import sqrt
>>> x, y = symbols('x y')
```

Two radicals are automatically joined through Mul:

```
>>> a=sqrt(x*sqrt(y))
>>> a*a**3 == a**4
True
```

But if an integer power of that radical has been autoexpanded then Mul does not join the resulting factors:

```
>>> a**4 # auto expands to a Mul, no longer a Pow
x**2*y
>>> _*a # so Mul doesn't combine them
x**2*y*sqrt(x*sqrt(y))
>>> powsimp(_) # but powsimp will
(x*sqrt(y))**(5/2)
>>> powsimp(x*y*a) # but won't when doing so would violate assumptions
x*y*sqrt(x*sqrt(y))
```

powdenest

`sympy.simplify.powsimp.`**`powdenest`**(*eq*, *force=False*, *polar=False*)

Collect exponents on powers as assumptions allow.

Explanation

Given `(bb**be)**e`, this can be simplified as follows:

- if `bb` is positive, or
- `e` is an integer, or
- `|be| < 1` then this simplifies to `bb**(be*e)`

Given a product of powers raised to a power, `(bb1**be1 * bb2**be2...)**e`, simplification can be done as follows:

- if e is positive, the gcd of all bei can be joined with e;
- all non-negative bb can be separated from those that are negative and their gcd can be joined with e; autosimplification already handles this separation.
- integer factors from powers that have integers in the denominator of the exponent can be removed from any term and the gcd of such integers can be joined with e

Setting `force` to `True` will make symbols that are not explicitly negative behave as though they are positive, resulting in more denesting.

Setting `polar` to `True` will do simplifications on the Riemann surface of the logarithm, also resulting in more denestings.

When there are sums of logs in exp() then a product of powers may be obtained e.g. `exp(3*(log(a) + 2*log(b)))` - > `a**3*b**6`.

Examples

```
>>> from sympy.abc import a, b, x, y, z
>>> from sympy import Symbol, exp, log, sqrt, symbols, powdenest
```

```
>>> powdenest((x**(2*a/3))**(3*x))
(x**(2*a/3))**(3*x)
>>> powdenest(exp(3*x*log(2)))
2**(3*x)
```

Assumptions may prevent expansion:

```
>>> powdenest(sqrt(x**2))
sqrt(x**2)
```

```
>>> p = symbols('p', positive=True)
>>> powdenest(sqrt(p**2))
p
```

No other expansion is done.

```
>>> i, j = symbols('i,j', integer=True)
>>> powdenest((x**x)**(i + j)) # -X-> (x**x)**i*(x**x)**j
x**(x*(i + j))
```

But exp() will be denested by moving all non-log terms outside of the function; this may result in the collapsing of the exp to a power with a different base:

```
>>> powdenest(exp(3*y*log(x)))
x**(3*y)
>>> powdenest(exp(y*(log(a) + log(b))))
(a*b)**y
>>> powdenest(exp(3*(log(a) + log(b))))
a**3*b**3
```

If assumptions allow, symbols can also be moved to the outermost exponent:

```
>>> i = Symbol('i', integer=True)
>>> powdenest(((x**(2*i))**(3*y))**x)
((x**(2*i))**(3*y))**x
>>> powdenest(((x**(2*i))**(3*y))**x, force=True)
x**(6*i*x*y)
```

```
>>> powdenest(((x**(2*a/3))**(3*y/i))**x)
((x**(2*a/3))**(3*y/i))**x
>>> powdenest((x**(2*i)*y**(4*i))**z, force=True)
(x*y**2)**(2*i*z)
```

```
>>> n = Symbol('n', negative=True)
```

```
>>> powdenest((x**i)**y, force=True)
x**(i*y)
>>> powdenest((n**i)**x, force=True)
(n**i)**x
```

Combinatorial simplification

combsimp

sympy.simplify.combsimp.**combsimp**(*expr*)

Simplify combinatorial expressions.

Explanation

This function takes as input an expression containing factorials, binomials, Pochhammer symbol and other "combinatorial" functions, and tries to minimize the number of those functions and reduce the size of their arguments.

The algorithm works by rewriting all combinatorial functions as gamma functions and applying gammasimp() except simplification steps that may make an integer argument non-integer. See docstring of gammasimp for more information.

Then it rewrites expression in terms of factorials and binomials by rewriting gammas as factorials and converting (a+b)!/a!b! into binomials.

If expression has gamma functions or combinatorial functions with non-integer argument, it is automatically passed to gammasimp.

Examples

```
>>> from sympy.simplify import combsimp
>>> from sympy import factorial, binomial, symbols
>>> n, k = symbols('n k', integer = True)
```

```
>>> combsimp(factorial(n)/factorial(n - 3))
n*(n - 2)*(n - 1)
>>> combsimp(binomial(n+1, k+1)/binomial(n, k))
(n + 1)/(k + 1)
```

Square Root Denesting

sqrtdenest

sympy.simplify.sqrtdenest.**sqrtdenest**(*expr*, *max_iter=3*)

Denests sqrts in an expression that contain other square roots if possible, otherwise returns the expr unchanged. This is based on the algorithms of [1].

Examples

```
>>> from sympy.simplify.sqrtdenest import sqrtdenest
>>> from sympy import sqrt
>>> sqrtdenest(sqrt(5 + 2 * sqrt(6)))
sqrt(2) + sqrt(3)
```

See also:

sympy.solvers.solvers.unrad (page 851)

References

[R763], [R764]

Common Subexpression Elimination

cse

`sympy.simplify.cse_main.cse(exprs, symbols=None, optimizations=None, postprocess=None, order='canonical', ignore=(), list=True)`

Perform common subexpression elimination on an expression.

Parameters

exprs : list of SymPy expressions, or a single SymPy expression

The expressions to reduce.

symbols : infinite iterator yielding unique Symbols

The symbols used to label the common subexpressions which are pulled out. The `numbered_symbols` generator is useful. The default is a stream of symbols of the form "x0", "x1", etc. This must be an infinite iterator.

optimizations : list of (callable, callable) pairs

The (preprocessor, postprocessor) pairs of external optimization functions. Optionally 'basic' can be passed for a set of predefined basic optimizations. Such 'basic' optimizations were used by default in old implementation, however they can be really slow on larger expressions. Now, no pre or post optimizations are made by default.

postprocess : a function which accepts the two return values of cse and

returns the desired form of output from cse, e.g. if you want the replacements reversed the function might be the following lambda: lambda r, e: return reversed(r), e

order : string, 'none' or 'canonical'

The order by which Mul and Add arguments are processed. If set to 'canonical', arguments will be canonically ordered. If set to 'none', ordering will be faster but dependent on expressions hashes, thus machine dependent and variable. For large expressions where speed is a concern, use the setting order='none'.

ignore : iterable of Symbols

Substitutions containing any Symbol from `ignore` will be ignored.

list : bool, (default True)

Returns expression in list or else with same type as input (when False).

Returns

replacements : list of (Symbol, expression) pairs

All of the common subexpressions that were replaced. Subexpressions earlier in this list might show up in subexpressions later in this list.

reduced_exprs : list of SymPy expressions

The reduced expressions with all of the replacements above.

Examples

```
>>> from sympy import cse, SparseMatrix
>>> from sympy.abc import x, y, z, w
>>> cse(((w + x + y + z)*(w + y + z))/(w + x)**3)
([(x0, y + z), (x1, w + x)], [(w + x0)*(x0 + x1)/x1**3])
```

List of expressions with recursive substitutions:

```
>>> m = SparseMatrix([x + y, x + y + z])
>>> cse([(x+y)**2, x + y + z, y + z, x + z + y, m])
([(x0, x + y), (x1, x0 + z)], [x0**2, x1, y + z, x1, Matrix([
[x0],
[x1]])])
```

Note: the type and mutability of input matrices is retained.

```
>>> isinstance(_[1][-1], SparseMatrix)
True
```

The user may disallow substitutions containing certain symbols:

```
>>> cse([y**2*(x + 1), 3*y**2*(x + 1)], ignore=(y,))
([(x0, x + 1)], [x0*y**2, 3*x0*y**2])
```

The default return value for the reduced expression(s) is a list, even if there is only one expression. The *list* flag preserves the type of the input in the output:

```
>>> cse(x)
([], [x])
>>> cse(x, list=False)
([], x)
```

opt_cse

sympy.simplify.cse_main.**opt_cse**(*exprs*, *order='canonical'*)

Find optimization opportunities in Adds, Muls, Pows and negative coefficient Muls.

Parameters

exprs : list of SymPy expressions

The expressions to optimize.

order : string, 'none' or 'canonical'

The order by which Mul and Add arguments are processed. For large expressions where speed is a concern, use the setting order='none'.

Returns

opt_subs : dictionary of expression substitutions

The expression substitutions which can be useful to optimize CSE.

Examples

```
>>> from sympy.simplify.cse_main import opt_cse
>>> from sympy.abc import x
>>> opt_subs = opt_cse([x**-2])
>>> k, v = list(opt_subs.keys())[0], list(opt_subs.values())[0]
>>> print((k, v.as_unevaluated_basic()))
(x**(-2), 1/(x**2))
```

tree_cse

`sympy.simplify.cse_main.`**`tree_cse`**(*exprs, symbols, opt_subs=None, order='canonical', ignore=()*)

Perform raw CSE on expression tree, taking opt_subs into account.

Parameters

exprs : list of SymPy expressions

The expressions to reduce.

symbols : infinite iterator yielding unique Symbols

The symbols used to label the common subexpressions which are pulled out.

opt_subs : dictionary of expression substitutions

The expressions to be substituted before any CSE action is performed.

order : string, 'none' or 'canonical'

The order by which Mul and Add arguments are processed. For large expressions where speed is a concern, use the setting order='none'.

ignore : iterable of Symbols

Substitutions containing any Symbol from `ignore` will be ignored.

Hypergeometric Function Expansion

hyperexpand

`sympy.simplify.hyperexpand.`**`hyperexpand`**(*f, allow_hyper=False, rewrite='default', place=None*)

Expand hypergeometric functions. If allow_hyper is True, allow partial simplification (that is a result different from input, but still containing hypergeometric functions).

If a G-function has expansions both at zero and at infinity, `place` can be set to `0` or `zoo` to indicate the preferred choice.

Examples

```
>>> from sympy.simplify.hyperexpand import hyperexpand
>>> from sympy.functions import hyper
>>> from sympy.abc import z
>>> hyperexpand(hyper([], [], z))
exp(z)
```

Non-hyperegeometric parts of the expression and hypergeometric expressions that are not recognised are left unchanged:

```
>>> hyperexpand(1 + hyper([1, 1, 1], [], z))
hyper((1, 1, 1), (), z) + 1
```

EPath Tools

EPath class

class sympy.simplify.epathtools.**EPath**(*path*)

Manipulate expressions using paths.

EPath grammar in EBNF notation:

```
literal   ::= /[A-Za-z_][A-Za-z_0-9]*/
number    ::= /-?\d+/
type      ::= literal
attribute ::= literal "?"
all       ::= "*"
slice     ::= "[" number? (":" number? (":" number?)?)? "]"
range     ::= all | slice
query     ::= (type | attribute) ("|" (type | attribute))*
selector  ::= range | query range?
path      ::= "/" selector ("/" selector)*
```

See the docstring of the epath() function.

apply(*expr, func, args=None, kwargs=None*)

Modify parts of an expression selected by a path.

Examples

```
>>> from sympy.simplify.epathtools import EPath
>>> from sympy import sin, cos, E
>>> from sympy.abc import x, y, z, t
```

```
>>> path = EPath("/*/[0]/Symbol")
>>> expr = [((x, 1), 2), ((3, y), z)]
```

```
>>> path.apply(expr, lambda expr: expr**2)
[((x**2, 1), 2), ((3, y**2), z)]
```

```
>>> path = EPath("/*/*/Symbol")
>>> expr = t + sin(x + 1) + cos(x + y + E)
```

```
>>> path.apply(expr, lambda expr: 2*expr)
t + sin(2*x + 1) + cos(2*x + 2*y + E)
```

select(*expr*)

Retrieve parts of an expression selected by a path.

Examples

```
>>> from sympy.simplify.epathtools import EPath
>>> from sympy import sin, cos, E
>>> from sympy.abc import x, y, z, t
```

```
>>> path = EPath("/*/[0]/Symbol")
>>> expr = [((x, 1), 2), ((3, y), z)]
```

```
>>> path.select(expr)
[x, y]
```

```
>>> path = EPath("/*/*/Symbol")
>>> expr = t + sin(x + 1) + cos(x + y + E)
```

```
>>> path.select(expr)
[x, x, y]
```

epath

sympy.simplify.epathtools.**epath**(*path*, *expr=None*, *func=None*, *args=None*, *kwargs=None*)

Manipulate parts of an expression selected by a path.

Parameters

path : str | EPath

A path as a string or a compiled EPath.

expr : Basic | iterable

An expression or a container of expressions.

func : callable (optional)

A callable that will be applied to matching parts.

args : tuple (optional)

Additional positional arguments to `func`.

kwargs : dict (optional)

Additional keyword arguments to `func`.

Explanation

This function allows to manipulate large nested expressions in single line of code, utilizing techniques to those applied in XML processing standards (e.g. XPath).

If `func` is `None`, *epath()* (page 689) retrieves elements selected by the `path`. Otherwise it applies `func` to each matching element.

Note that it is more efficient to create an EPath object and use the select and apply methods of that object, since this will compile the path string only once. This function should only be used as a convenient shortcut for interactive use.

This is the supported syntax:

- **select all: `/*`**
  Equivalent of `for arg in args:`.
- **select slice: `/[0]` or `/[1:5]` or `/[1:5:2]`**
  Supports standard Python's slice syntax.
- **select by type: `/list` or `/list|tuple`**
  Emulates `isinstance()`.
- **select by attribute: `/__iter__?`**
  Emulates `hasattr()`.

Examples

```
>>> from sympy.simplify.epathtools import epath
>>> from sympy import sin, cos, E
>>> from sympy.abc import x, y, z, t
```

```
>>> path = "/*/[0]/Symbol"
>>> expr = [((x, 1), 2), ((3, y), z)]
```

```
>>> epath(path, expr)
[x, y]
>>> epath(path, expr, lambda expr: expr**2)
[((x**2, 1), 2), ((3, y**2), z)]
```

```
>>> path = "/*/*/Symbol"
>>> expr = t + sin(x + 1) + cos(x + y + E)
```

```
>>> epath(path, expr)
[x, x, y]
>>> epath(path, expr, lambda expr: 2*expr)
t + sin(2*x + 1) + cos(2*x + 2*y + E)
```

Hypergeometric Expansion

This page describes how the function *hyperexpand()* (page 687) and related code work. For usage, see the documentation of the symplify module.

Hypergeometric Function Expansion Algorithm

This section describes the algorithm used to expand hypergeometric functions. Most of it is based on the papers [Roach1996] and [Roach1997].

Recall that the hypergeometric function is (initially) defined as

$$ {}_pF_q\left(\begin{matrix} a_1, \cdots, a_p \\ b_1, \cdots, b_q \end{matrix} \middle| z \right) = \sum_{n=0}^{\infty} \frac{(a_1)_n \cdots (a_p)_n}{(b_1)_n \cdots (b_q)_n} \frac{z^n}{n!}. $$

It turns out that there are certain differential operators that can change the a_p and p_q parameters by integers. If a sequence of such operators is known that converts the set of indices a_r^0 and b_s^0 into a_p and b_q, then we shall say the pair a_p, b_q is reachable from a_r^0, b_s^0. Our general strategy is thus as follows: given a set a_p, b_q of parameters, try to look up an origin a_r^0, b_s^0 for which we know an expression, and then apply the sequence of differential operators to the known expression to find an expression for the Hypergeometric function we are interested in.

Notation

In the following, the symbol a will always denote a numerator parameter and the symbol b will always denote a denominator parameter. The subscripts p, q, r, s denote vectors of that length, so e.g. a_p denotes a vector of p numerator parameters. The subscripts i and j denote "running indices", so they should usually be used in conjunction with a "for all i". E.g. $a_i < 4$ for all i. Uppercase subscripts I and J denote a chosen, fixed index. So for example $a_I > 0$ is true if the inequality holds for the one index I we are currently interested in.

Incrementing and decrementing indices

Suppose $a_i \neq 0$. Set $A(a_i) = \frac{z}{a_i}\frac{\mathrm{d}}{dz} + 1$. It is then easy to show that $A(a_i)\, {}_pF_q\left({a_p \atop b_q} \middle| z\right) = {}_pF_q\left({a_p + e_i \atop b_q} \middle| z\right)$, where e_i is the i-th unit vector. Similarly for $b_j \neq 1$ we set $B(b_j) = \frac{z}{b_j - 1}\frac{\mathrm{d}}{dz} + 1$ and find $B(b_j)\, {}_pF_q\left({a_p \atop b_q} \middle| z\right) = {}_pF_q\left({a_p \atop b_q - e_i} \middle| z\right)$. Thus we can increment upper and decrement lower indices at will, as long as we don't go through zero. The $A(a_i)$ and $B(b_j)$ are called shift operators.

It is also easy to show that $\frac{\mathrm{d}}{dz}\, {}_pF_q\left({a_p \atop b_q} \middle| z\right) = \frac{a_1 \cdots a_p}{b_1 \cdots b_q}\, {}_pF_q\left({a_p + 1 \atop b_q + 1} \middle| z\right)$, where $a_p + 1$ is the vector $a_1 + 1, a_2 + 1, \ldots$ and similarly for $b_q + 1$. Combining this with the shift operators, we arrive at one form of the Hypergeometric differential equation: $\left[\frac{\mathrm{d}}{dz} \prod_{j=1}^{q} B(b_j) - \frac{a_1 \cdots a_p}{(b_1 - 1) \cdots (b_q - 1)} \prod_{i=1}^{p} A(a_i)\right] {}_pF_q\left({a_p \atop b_q} \middle| z\right) = 0$. This holds if all shift operators are defined, i.e. if no $a_i = 0$ and no $b_j = 1$. Clearing denominators and multiplying through by z we arrive at the following equation: $\left[z\frac{\mathrm{d}}{dz} \prod_{j=1}^{q} \left(z\frac{\mathrm{d}}{dz} + b_j - 1\right) - z \prod_{i=1}^{p} \left(z\frac{\mathrm{d}}{dz} + a_i\right)\right] {}_pF_q\left({a_p \atop b_q} \middle| z\right) = 0$. Even though our derivation does not show it, it can be checked that this equation holds whenever the ${}_pF_q$ is defined.

Notice that, under suitable conditions on a_I, b_J, each of the operators $A(a_i)$, $B(b_j)$ and $z\frac{\mathrm{d}}{dz}$ can be expressed in terms of $A(a_I)$ or $B(b_J)$. Our next aim is to write the Hypergeometric differential equation as follows: $[XA(a_I) - r]_pF_q\left(\begin{matrix} a_p \\ b_q \end{matrix}\middle| z\right) = 0$, for some operator X and some constant r to be determined. If $r \neq 0$, then we can write this as $\frac{-1}{r}X_pF_q\left(\begin{matrix} a_p+e_I \\ b_q \end{matrix}\middle| z\right) = {}_pF_q\left(\begin{matrix} a_p \\ b_q \end{matrix}\middle| z\right)$, and so $\frac{-1}{r}X$ undoes the shifting of $A(a_I)$, whence it will be called an inverse-shift operator.

Now $A(a_I)$ exists if $a_I \neq 0$, and then $z\frac{\mathrm{d}}{\mathrm{d}z} = a_I A(a_I) - a_I$. Observe also that all the operators $A(a_i)$, $B(b_j)$ and $z\frac{\mathrm{d}}{\mathrm{d}z}$ commute. We have $\prod_{i=1}^{p}\left(z\frac{\mathrm{d}}{\mathrm{d}z} + a_i\right) = \left(\prod_{i=1, i\neq I}^{p}\left(z\frac{\mathrm{d}}{\mathrm{d}z} + a_i\right)\right) a_I A(a_I)$, so this gives us the first half of X. The other half does not have such a nice expression. We find $z\frac{d}{dz}\prod_{j=1}^{q}\left(z\frac{d}{dz} + b_j - 1\right) = (a_I A(a_I) - a_I)\prod_{j=1}^{q}(a_I A(a_I) - a_I + b_j - 1)$. Since the first half had no constant term, we infer $r = -a_I\prod_{j=1}^{q}(b_j - 1 - a_I)$.

This tells us under which conditions we can "un-shift" $A(a_I)$, namely when $a_I \neq 0$ and $r \neq 0$. Substituting $a_I - 1$ for a_I then tells us under what conditions we can decrement the index a_I. Doing a similar analysis for $B(a_J)$, we arrive at the following rules:

- An index a_I can be decremented if $a_I \neq 1$ and $a_I \neq b_j$ for all b_j.
- An index b_J can be incremented if $b_J \neq -1$ and $b_J \neq a_i$ for all a_i.

Combined with the conditions (stated above) for the existence of shift operators, we have thus established the rules of the game!

Reduction of Order

Notice that, quite trivially, if $a_I = b_J$, we have ${}_pF_q\left(\begin{matrix} a_p \\ b_q \end{matrix}\middle| z\right) = {}_{p-1}F_{q-1}\left(\begin{matrix} a_p^* \\ b_q^* \end{matrix}\middle| z\right)$, where a_p^* means a_p with a_I omitted, and similarly for b_q^*. We call this reduction of order.

In fact, we can do even better. If $a_I - b_J \in \mathbb{Z}_{>0}$, then it is easy to see that $\frac{(a_I)_n}{(b_J)_n}$ is actually a polynomial in n. It is also easy to see that $(z\frac{\mathrm{d}}{\mathrm{d}z})^k z^n = n^k z^n$. Combining these two remarks we find:

> If $a_I - b_J \in \mathbb{Z}_{>0}$, then there exists a polynomial $p(n) = p_0 + p_1 n + \cdots$ (of degree $a_I - b_J$) such that $\frac{(a_I)_n}{(b_J)_n} = p(n)$ and ${}_pF_q\left(\begin{matrix} a_p \\ b_q \end{matrix}\middle| z\right) = \left(p_0 + p_1 z\frac{\mathrm{d}}{\mathrm{d}z} + p_2\left(z\frac{\mathrm{d}}{\mathrm{d}z}\right)^2 + \cdots\right){}_{p-1}F_{q-1}\left(\begin{matrix} a_p^* \\ b_q^* \end{matrix}\middle| z\right)$.

Thus any set of parameters a_p, b_q is reachable from a set of parameters c_r, d_s where $c_i - d_j \in \mathbb{Z}$ implies $c_i < d_j$. Such a set of parameters c_r, d_s is called suitable. Our database of known formulae should only contain suitable origins. The reasons are twofold: firstly, working from suitable origins is easier, and secondly, a formula for a non-suitable origin can be deduced from a lower order formula, and we should put this one into the database instead.

Moving Around in the Parameter Space

It remains to investigate the following question: suppose a_p, b_q and a_p^0, b_q^0 are both suitable, and also $a_i - a_i^0 \in \mathbb{Z}$, $b_j - b_j^0 \in \mathbb{Z}$. When is a_p, b_q reachable from a_p^0, b_q^0? It is clear that we can treat all parameters independently that are incongruent mod 1. So assume that a_i and b_j are congruent to r mod 1, for all i and j. The same then follows for a_i^0 and b_j^0.

If $r \neq 0$, then any such a_p, b_q is reachable from any a_p^0, b_q^0. To see this notice that there exist constants c, c^0, congruent mod 1, such that $a_i < c < b_j$ for all i and j, and similarly $a_i^0 < c^0 < b_j^0$. If $n = c - c^0 > 0$ then we first inverse-shift all the b_j^0 n times up, and then similarly shift up all the a_i^0 n times. If $n < 0$ then we first inverse-shift down the a_i^0 and then shift down the b_j^0. This reduces to the case $c = c^0$. But evidently we can now shift or inverse-shift around the a_i^0

arbitrarily so long as we keep them less than c, and similarly for the b_j^0 so long as we keep them bigger than c. Thus a_p, b_q is reachable from a_p^0, b_q^0.

If $r = 0$ then the problem is slightly more involved. WLOG no parameter is zero. We now have one additional complication: no parameter can ever move through zero. Hence a_p, b_q is reachable from a_p^0, b_q^0 if and only if the number of $a_i < 0$ equals the number of $a_i^0 < 0$, and similarly for the b_i and b_i^0. But in a suitable set of parameters, all $b_j > 0$! This is because the Hypergeometric function is undefined if one of the b_j is a non-positive integer and all a_i are smaller than the b_j. Hence the number of $b_j \le 0$ is always zero.

We can thus associate to every suitable set of parameters a_p, b_q, where no $a_i = 0$, the following invariants:

- For every $r \in [0, 1)$ the number α_r of parameters $a_i \equiv r \pmod 1$, and similarly the number β_r of parameters $b_i \equiv r \pmod 1$.
- The number γ of integers a_i with $a_i < 0$.

The above reasoning shows that a_p, b_q is reachable from a_p^0, b_q^0 if and only if the invariants $\alpha_r, \beta_r, \gamma$ all agree. Thus in particular "being reachable from" is a symmetric relation on suitable parameters without zeros.

Applying the Operators

If all goes well then for a given set of parameters we find an origin in our database for which we have a nice formula. We now have to apply (potentially) many differential operators to it. If we do this blindly then the result will be very messy. This is because with Hypergeometric type functions, the derivative is usually expressed as a sum of two contiguous functions. Hence if we compute N derivatives, then the answer will involve $2N$ contiguous functions! This is clearly undesirable. In fact we know from the Hypergeometric differential equation that we need at most $\max(p, q + 1)$ contiguous functions to express all derivatives.

Hence instead of differentiating blindly, we will work with a $\mathbb{C}(z)$-module basis: for an origin a_r^0, b_s^0 we either store (for particularly pretty answers) or compute a set of N functions (typically $N = \max(r, s + 1)$) with the property that the derivative of any of them is a $\mathbb{C}(z)$-linear combination of them. In formulae, we store a vector B of N functions, a matrix M and a vector C (the latter two with entries in $\mathbb{C}(z)$), with the following properties:

- ${}_rF_s\left(\begin{matrix} a_r^0 \\ b_s^0 \end{matrix} \middle| z\right) = CB$
- $z\frac{\mathrm{d}}{\mathrm{d}z}B = MB$.

Then we can compute as many derivatives as we want and we will always end up with $\mathbb{C}(z)$-linear combination of at most N special functions.

As hinted above, B, M and C can either all be stored (for particularly pretty answers) or computed from a single ${}_pF_q$ formula.

Loose Ends

This describes the bulk of the hypergeometric function algorithm. There a few further tricks, described in the hyperexpand.py source file. The extension to Meijer G-functions is also described there.

Meijer G-Functions of Finite Confluence

Slater's theorem essentially evaluates a G-function as a sum of residues. If all poles are simple, the resulting series can be recognised as hypergeometric series. Thus a G-function can be evaluated as a sum of Hypergeometric functions.

If the poles are not simple, the resulting series are not hypergeometric. This is known as the "confluent" or "logarithmic" case (the latter because the resulting series tend to contain logarithms). The answer depends in a complicated way on the multiplicities of various poles, and there is no accepted notation for representing it (as far as I know). However if there are only finitely many multiple poles, we can evaluate the G function as a sum of hypergeometric functions, plus finitely many extra terms. I could not find any good reference for this, which is why I work it out here.

Recall the general setup. We define

$$G(z) = \frac{1}{2\pi i} \int_L \frac{\prod_{j=1}^m \Gamma(b_j - s) \prod_{j=1}^n \Gamma(1 - a_j + s)}{\prod_{j=m+1}^q \Gamma(1 - b_j + s) \prod_{j=n+1}^p \Gamma(a_j - s)} z^s \mathrm{d}s,$$

where L is a contour starting and ending at $+\infty$, enclosing all of the poles of $\Gamma(b_j - s)$ for $j = 1, \ldots, n$ once in the negative direction, and no other poles. Also the integral is assumed absolutely convergent.

In what follows, for any complex numbers a, b, we write $a \equiv b \pmod 1$ if and only if there exists an integer k such that $a - b = k$. Thus there are double poles iff $a_i \equiv a_j \pmod 1$ for some $i \neq j \leq n$.

We now assume that whenever $b_j \equiv a_i \pmod 1$ for $i \leq m$, $j > n$ then $b_j < a_i$. This means that no quotient of the relevant gamma functions is a polynomial, and can always be achieved by "reduction of order". Fix a complex number c such that $\{b_i | b_i \equiv c \pmod 1, i \leq m\}$ is not empty. Enumerate this set as $b, b + k_1, \ldots, b + k_u$, with k_i non-negative integers. Enumerate similarly $\{a_j | a_j \equiv c \pmod 1, j > n\}$ as $b + l_1, \ldots, b + l_v$. Then $l_i > k_j$ for all i, j. For finite confluence, we need to assume $v \geq u$ for all such c.

Let $c_1, \ldots, c_w$ be distinct $\pmod 1$ and exhaust the congruence classes of the b_i. I claim

$$G(z) = -\sum_{j=1}^{w} (F_j(z) + R_j(z)),$$

where $F_j(z)$ is a hypergeometric function and $R_j(z)$ is a finite sum, both to be specified later. Indeed corresponding to every c_j there is a sequence of poles, at mostly finitely many of them multiple poles. This is where the j-th term comes from.

Hence fix again c, enumerate the relevant b_i as $b, b + k_1, \ldots, b + k_u$. We will look at the a_j corresponding to $a + l_1, \ldots, a + l_u$. The other a_i are not treated specially. The corresponding gamma functions have poles at (potentially) $s = b + r$ for $r = 0, 1, \ldots$. For $r \geq l_u$, pole of the integrand is simple. We thus set

$$R(z) = \sum_{r=0}^{l_u - 1} res_{s=r+b}.$$

We finally need to investigate the other poles. Set $r = l_u + t$, $t \geq 0$. A computation shows

$$\frac{\Gamma(k_i - l_u - t)}{\Gamma(l_i - l_u - t)} = \frac{1}{(k_i - l_u - t)_{l_i - k_i}} = \frac{(-1)^{\delta_i}}{(l_u - l_i + 1)_{\delta_i}} \frac{(l_u - l_i + 1)_t}{(l_u - k_i + 1)_t},$$

where $\delta_i = l_i - k_i$.

Also

$$\Gamma(b_j - l_u - b - t) = \frac{\Gamma(b_j - l_u - b)}{(-1)^t (l_u + b + 1 - b_j)_t},$$

$$\Gamma(1 - a_j + l_u + b + t) = \Gamma(1 - a_j + l_u + b)(1 - a_j + l_u + b)_t$$

and

$$res_{s=b+l_u+t}\Gamma(b - s) = -\frac{(-1)^{l_u+t}}{(l_u + t)!} = -\frac{(-1)^{l_u}}{l_u!} \frac{(-1)^t}{(l_u + 1)_t}.$$

Hence

$$\begin{aligned} res_{s=b+l_u+t} =& -z^{b+l_u} \frac{(-1)^{l_u}}{l_u!} \prod_{i=1}^{u} \frac{(-1)^{\delta_i}}{(l_u - k_i + 1)_{\delta_i}} \frac{\prod_{j=1}^{n} \Gamma(1 - a_j + l_u + b) \prod_{j=1}^{m} \Gamma(b_j - l_u - b)^*}{\prod_{j=n+1}^{p} \Gamma(a_j - l_u - b)^* \prod_{j=m+1}^{q} \Gamma(1 - b_j + l_u + b)} \\ &\times z^t \frac{(-1)^t}{(l_u + 1)_t} \prod_{i=1}^{u} \frac{(l_u - l_i + 1)_t}{(l_u - k_i + 1)_t} \frac{\prod_{j=1}^{n} (1 - a_j + l_u + b)_t \prod_{j=n+1}^{p} (-1)^t (l_u + b + 1 - a_j)_t^*}{\prod_{j=1}^{m} (-1)^t (l_u + b + 1 - b_j)_t^* \prod_{j=m+1}^{q} (1 - b_j + l_u + b)_t}, \end{aligned}$$

where the $*$ means to omit the terms we treated specially.

We thus arrive at

$$F(z) = C \times {}_{p+1}F_q \left(\begin{matrix} 1, (1 + l_u - l_i), (1 + l_u + b - a_i)^* \\ 1 + l_u, (1 + l_u - k_i), (1 + l_u + b - b_i)^* \end{matrix} \middle| (-1)^{p-m-n} z \right),$$

where C designates the factor in the residue independent of t. (This result can also be written in slightly simpler form by converting all the l_u etc back to $a_* - b_*$, but doing so is going to require more notation still and is not helpful for computation.)

Extending The Hypergeometric Tables

Adding new formulae to the tables is straightforward. At the top of the file `sympy/simplify/hyperexpand.py`, there is a function called `add_formulae()`. Nested in it are defined two helpers, `add(ap, bq, res)` and `addb(ap, bq, B, C, M)`, as well as dummys `a`, `b`, `c`, and `z`.

The first step in adding a new formula is by using `add(ap, bq, res)`. This declares `hyper(ap, bq, z) == res`. Here `ap` and `bq` may use the dummys `a`, `b`, and `c` as free symbols. For example the well-known formula $\sum_0^\infty \frac{(-a)_n z^n}{n!} = (1 - z)^a$ is declared by the following line: `add((-a, ), (), (1-z)**a)`.

From the information provided, the matrices B, C and M will be computed, and the formula is now available when expanding hypergeometric functions. Next the test file `sympy/simplify/tests/test_hyperexpand.py` should be run, in particular the test `test_formulae()`. This will test the newly added formula numerically. If it fails, there is (presumably) a typo in what was entered.

Since all newly-added formulae are probably relatively complicated, chances are that the automatically computed basis is rather suboptimal (there is no good way of testing this, other than observing very messy output). In this case the matrices B, C and M should be computed by hand. Then the helper `addb` can be used to declare a hypergeometric formula with hand-computed basis.

An example

Because this explanation so far might be very theoretical and difficult to understand, we walk through an explicit example now. We take the Fresnel function $C(z)$ which obeys the following hypergeometric representation:

$$C(z) = z \cdot {}_{1}F_{2}\left(\begin{matrix} \frac{1}{4} \\ \frac{1}{2}, \frac{5}{4} \end{matrix}\middle| -\frac{\pi^2 z^4}{16}\right).$$

First we try to add this formula to the lookup table by using the (simpler) function `add(ap, bq, res)`. The first two arguments are simply the lists containing the parameter sets of ${}_1F_2$. The `res` argument is a little bit more complicated. We only know $C(z)$ in terms of ${}_1F_2(\ldots|f(z))$ with f a function of z, in our case

$$f(z) = -\frac{\pi^2 z^4}{16}.$$

What we need is a formula where the hypergeometric function has only z as argument ${}_1F_2(\ldots|z)$. We introduce the new complex symbol w and search for a function $g(w)$ such that

$$f(g(w)) = w$$

holds. Then we can replace every z in $C(z)$ by $g(w)$. In the case of our example the function g could look like

$$g(w) = \frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) w^{\frac{1}{4}}.$$

We get these functions mainly by guessing and testing the result. Hence we proceed by computing $f(g(w))$ (and simplifying naively)

$$\begin{aligned} f(g(w)) &= -\frac{\pi^2 g(w)^4}{16} \\ &= -\frac{\pi^2 g\left(\frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) w^{\frac{1}{4}}\right)^4}{16} \\ &= -\frac{\pi^2 \frac{2^4}{\sqrt{\pi}^4} \exp\left(\frac{i\pi}{4}\right)^4 w^{\frac{1}{4}4}}{16} \\ &= -\exp(i\pi)\, w \\ &= w \end{aligned}$$

and indeed get back w. (In case of branched functions we have to be aware of branch cuts. In that case we take w to be a positive real number and check the formula. If what we have found works for positive w, then just replace *`exp`* (page 411) inside any branched function by *`exp_polar`* (page 414) and what we get is right for *all* w.) Hence we can write the formula as

$$C(g(w)) = g(w) \cdot {}_{1}F_{2}\left(\begin{matrix} \frac{1}{4} \\ \frac{1}{2}, \frac{5}{4} \end{matrix}\middle| w\right).$$

and trivially

$${}_{1}F_{2}\left(\begin{matrix} \frac{1}{4} \\ \frac{1}{2}, \frac{5}{4} \end{matrix}\middle| w\right) = \frac{C(g(w))}{g(w)} = \frac{C\left(\frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) w^{\frac{1}{4}}\right)}{\frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) w^{\frac{1}{4}}}$$

which is exactly what is needed for the third parameter, `res`, in `add`. Finally, the whole function call to add this rule to the table looks like:

```
add([S(1)/4],
    [S(1)/2, S(5)/4],
    fresnelc(exp(pi*I/4)*root(z,4)*2/sqrt(pi)) / (exp(pi*I/4)*root(z,4)*2/
↪sqrt(pi))
   )
```

Using this rule we will find that it works but the results are not really nice in terms of simplicity and number of special function instances included. We can obtain much better results by adding the formula to the lookup table in another way. For this we use the (more complicated) function `addb(ap, bq, B, C, M)`. The first two arguments are again the lists containing the parameter sets of ${}_1F_2$. The remaining three are the matrices mentioned earlier on this page.

We know that the $n = \max(p, q + 1)$-th derivative can be expressed as a linear combination of lower order derivatives. The matrix B contains the basis $\{B_0, B_1, \ldots\}$ and is of shape $n \times 1$. The best way to get B_i is to take the first $n = \max(p, q + 1)$ derivatives of the expression for ${}_pF_q$ and take out useful pieces. In our case we find that $n = \max(1, 2 + 1) = 3$. For computing the derivatives, we have to use the operator $z\frac{\mathrm{d}}{\mathrm{d}z}$. The first basis element B_0 is set to the expression for ${}_1F_2$ from above:

$$B_0 = \frac{\sqrt{\pi}\exp\left(-\frac{i\pi}{4}\right)C\left(\frac{2}{\sqrt{\pi}}\exp\left(\frac{i\pi}{4}\right)z^{\frac{1}{4}}\right)}{2z^{\frac{1}{4}}}$$

Next we compute $z\frac{\mathrm{d}}{\mathrm{d}z}B_0$. For this we can directly use SymPy!

```
>>> from sympy import Symbol, sqrt, exp, I, pi, fresnelc, root, diff, expand
>>> z = Symbol("z")
>>> B0 = sqrt(pi)*exp(-I*pi/4)*fresnelc(2*root(z,4)*exp(I*pi/4)/sqrt(pi))/\
...          (2*root(z,4))
>>> z * diff(B0, z)
z*(cosh(2*sqrt(z))/(4*z) - sqrt(pi)*exp(-I*pi/4)*fresnelc(2*z**(1/4)*exp(I*pi/
↪4)/sqrt(pi))/(8*z**(5/4)))
>>> expand(_)
cosh(2*sqrt(z))/4 - sqrt(pi)*exp(-I*pi/4)*fresnelc(2*z**(1/4)*exp(I*pi/4)/
↪sqrt(pi))/(8*z**(1/4))
```

Formatting this result nicely we obtain

$$B_1' = -\frac{1}{4}\frac{\sqrt{\pi}\exp\left(-\frac{i\pi}{4}\right)C\left(\frac{2}{\sqrt{\pi}}\exp\left(\frac{i\pi}{4}\right)z^{\frac{1}{4}}\right)}{2z^{\frac{1}{4}}} + \frac{1}{4}\cosh\left(2\sqrt{z}\right)$$

Computing the second derivative we find

```
>>> from sympy import (Symbol, cosh, sqrt, pi, exp, I, fresnelc, root,
...                    diff, expand)
>>> z = Symbol("z")
>>> B1prime = cosh(2*sqrt(z))/4 - sqrt(pi)*exp(-I*pi/4)*\
...           fresnelc(2*root(z,4)*exp(I*pi/4)/sqrt(pi))/(8*root(z,4))
>>> z * diff(B1prime, z)
z*(-cosh(2*sqrt(z))/(16*z) + sinh(2*sqrt(z))/(4*sqrt(z)) + sqrt(pi)*exp(-I*pi/
↪4)*fresnelc(2*z**(1/4)*exp(I*pi/4)/sqrt(pi))/(32*z**(5/4)))
>>> expand(_)
sqrt(z)*sinh(2*sqrt(z))/4 - cosh(2*sqrt(z))/16 + sqrt(pi)*exp(-I*pi/
↪4)*fresnelc(2*z**(1/4)*exp(I*pi/4)/sqrt(pi))/(32*z**(1/4))
```

which can be printed as

$$B_2' = \frac{1}{16} \frac{\sqrt{\pi} \exp\left(-\frac{\imath\pi}{4}\right) C\left(\frac{2}{\sqrt{\pi}} \exp\left(\frac{\imath\pi}{4}\right) z^{\frac{1}{4}}\right)}{2z^{\frac{1}{4}}} - \frac{1}{16} \cosh\left(2\sqrt{z}\right) + \frac{1}{4} \sinh\left(2\sqrt{z}\right)\sqrt{z}$$

We see the common pattern and can collect the pieces. Hence it makes sense to choose B_1 and B_2 as follows

$$B = \begin{pmatrix} B_0 \\ B_1 \\ B_2 \end{pmatrix} = \begin{pmatrix} \frac{\sqrt{\pi} \exp\left(-\frac{\imath\pi}{4}\right) C\left(\frac{2}{\sqrt{\pi}} \exp\left(\frac{\imath\pi}{4}\right) z^{\frac{1}{4}}\right)}{2z^{\frac{1}{4}}} \\ \cosh\left(2\sqrt{z}\right) \\ \sinh\left(2\sqrt{z}\right)\sqrt{z} \end{pmatrix}$$

(This is in contrast to the basis $B = (B_0, B_1', B_2')$ that would have been computed automatically if we used just `add(ap, bq, res)`.)

Because it must hold that ${}_pF_q\left(\cdots|z\right) = CB$ the entries of C are obviously

$$C = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

Finally we have to compute the entries of the 3×3 matrix M such that $z\frac{d}{dz}B = MB$ holds. This is easy. We already computed the first part $z\frac{d}{dz}B_0$ above. This gives us the first row of M. For the second row we have:

```
>>> from sympy import Symbol, cosh, sqrt, diff
>>> z = Symbol("z")
>>> B1 = cosh(2*sqrt(z))
>>> z * diff(B1, z)
sqrt(z)*sinh(2*sqrt(z))
```

and for the third one

```
>>> from sympy import Symbol, sinh, sqrt, expand, diff
>>> z = Symbol("z")
>>> B2 = sinh(2*sqrt(z))*sqrt(z)
>>> expand(z * diff(B2, z))
sqrt(z)*sinh(2*sqrt(z))/2 + z*cosh(2*sqrt(z))
```

Now we have computed the entries of this matrix to be

$$M = \begin{pmatrix} -\frac{1}{4} & \frac{1}{4} & 0 \\ 0 & 0 & 1 \\ 0 & z & \frac{1}{2} \end{pmatrix}$$

Note that the entries of C and M should typically be rational functions in z, with rational coefficients. This is all we need to do in order to add a new formula to the lookup table for `hyperexpand`.

Implemented Hypergeometric Formulae

A vital part of the algorithm is a relatively large table of hypergeometric function representations. The following automatically generated list contains all the representations implemented in SymPy (of course many more are derived from them). These formulae are mostly taken from [Luke1969] and [Prudnikov1990]. They are all tested numerically.

$${}_{0}F_{0}\left(\middle| z\right) = e^{z}$$

$${}_{1}F_{0}\left(a\middle| z\right) = \left(1 - z\right)^{- a}$$

$${}_{2}F_{1}\left(\begin{matrix} a, a - \frac{1}{2} \\ 2 a \end{matrix}\middle| z\right) = 2^{2 a - 1} \left(\sqrt{1 - z} + 1\right)^{1 - 2 a}$$

$${}_{2}F_{1}\left(\begin{matrix} 1, 1 \\ 2 \end{matrix}\middle| z\right) = - \frac{\log{\left(1 - z \right)}}{z}$$

$${}_{2}F_{1}\left(\begin{matrix} \frac{1}{2}, 1 \\ \frac{3}{2} \end{matrix}\middle| z\right) = \frac{\operatorname{atanh}{\left(\sqrt{z} \right)}}{\sqrt{z}}$$

$${}_{2}F_{1}\left(\begin{matrix} \frac{1}{2}, \frac{1}{2} \\ \frac{3}{2} \end{matrix}\middle| z\right) = \frac{\operatorname{asin}{\left(\sqrt{z} \right)}}{\sqrt{z}}$$

$${}_{2}F_{1}\left(\begin{matrix} a, a + \frac{1}{2} \\ \frac{1}{2} \end{matrix}\middle| z\right) = \frac{\left(\sqrt{z} + 1\right)^{- 2 a}}{2} + \frac{\left(1 - \sqrt{z}\right)^{- 2 a}}{2}$$

$${}_{2}F_{1}\left(\begin{matrix} a, - a \\ \frac{1}{2} \end{matrix}\middle| z\right) = \cos{\left(2 a \operatorname{asin}{\left(\sqrt{z} \right)} \right)}$$

$${}_{2}F_{1}\left(\begin{matrix} 1, 1 \\ \frac{3}{2} \end{matrix}\middle| z\right) = \frac{\operatorname{asin}{\left(\sqrt{z} \right)}}{\sqrt{z} \sqrt{1 - z}}$$

$${}_{2}F_{1}\left(\begin{matrix} \frac{1}{2}, \frac{1}{2} \\ 1 \end{matrix}\middle| z\right) = \frac{2 K\left(z\right)}{\pi}$$

$${}_{2}F_{1}\left(\begin{matrix} - \frac{1}{2}, \frac{1}{2} \\ 1 \end{matrix}\middle| z\right) = \frac{2 E\left(z\right)}{\pi}$$

$${}_{3}F_{2}\left(\begin{matrix} - \frac{1}{2}, 1, 1 \\ \frac{1}{2}, 2 \end{matrix}\middle| z\right) = - \frac{2 \sqrt{z} \operatorname{atanh}{\left(\sqrt{z} \right)}}{3} + \frac{2}{3} - \frac{\log{\left(1 - z \right)}}{3 z}$$

$${}_{3}F_{2}\left(\begin{matrix} - \frac{1}{2}, 1, 1 \\ 2, 2 \end{matrix}\middle| z\right) = \left(\frac{4}{9} - \frac{16}{9 z}\right) \sqrt{1 - z} + \frac{4 \log{\left(\frac{\sqrt{1 - z}}{2} + \frac{1}{2} \right)}}{3 z} + \frac{16}{9 z}$$

$${}_{1}F_{1}\left(\begin{matrix} 1 \\ b \end{matrix}\middle| z\right) = z^{1 - b} \left(b - 1\right) e^{z} \gamma\left(b - 1, z\right)$$

$${}_{1}F_{1}\left(\begin{matrix} a \\ 2 a \end{matrix}\middle| z\right) = 4^{a - \frac{1}{2}} z^{\frac{1}{2} - a} e^{\frac{z}{2}} I_{a - \frac{1}{2}}\left(\frac{z}{2}\right) \Gamma\left(a + \frac{1}{2}\right)$$

$${}_{1}F_{1}\left(\begin{matrix} a \\ a + 1 \end{matrix}\middle| z\right) = a \left(z e^{i \pi}\right)^{- a} \gamma\left(a, z e^{i \pi}\right)$$

$${}_{1}F_{1}\left(\begin{matrix} - \frac{1}{2} \\ \frac{1}{2} \end{matrix}\middle| z\right) = \sqrt{z} i \sqrt{\pi} \operatorname{erf}{\left(\sqrt{z} i \right)} + e^{z}$$

$${}_{1}F_{2}\left(\begin{matrix} 1 \\ \frac{3}{4}, \frac{5}{4} \end{matrix}\middle| z\right) = \frac{\sqrt{\pi} \left(i \sinh{\left(2 \sqrt{z} \right)} S\left(\frac{2 \sqrt[4]{z} e^{\frac{i \pi}{4}}}{\sqrt{\pi}}\right) + \cosh{\left(2 \sqrt{z} \right)} C\left(\frac{2 \sqrt[4]{z} e^{\frac{i \pi}{4}}}{\sqrt{\pi}}\right)\right) e^{- \frac{i \pi}{4}}}{2 \sqrt[4]{z}}$$

$${}_{2}F_{2}\left(\begin{matrix} \frac{1}{2}, a \\ \frac{3}{2}, a + 1 \end{matrix}\middle| z\right) = - \frac{a i \sqrt{\pi} \sqrt{\frac{1}{z}} \operatorname{erf}{\left(\sqrt{z} i \right)}}{2 a - 1} - \frac{a \left(z e^{i \pi}\right)^{- a} \gamma\left(a, z e^{i \pi}\right)}{2 a - 1}$$

$${}_{2}F_{2}\left(\begin{matrix} 1, 1 \\ 2, 2 \end{matrix}\middle| z\right) = \frac{- \log{\left(z \right)} + \operatorname{Ei}{\left(z \right)}}{z} - \frac{\gamma}{z}$$

$${}_{0}F_{1}\left(\begin{matrix} \\ \frac{1}{2} \end{matrix}\middle| z\right) = \cosh{\left(2 \sqrt{z} \right)}$$

$${}_{0}F_{1}\left(\begin{matrix} \\ b \end{matrix}\middle| z\right) = z^{\frac{1}{2} - \frac{b}{2}} I_{b - 1}\left(2 \sqrt{z}\right) \Gamma\left(b\right)$$

$${}_{0}F_{3}\left(\begin{matrix} \\ \frac{1}{2}, a, a + \frac{1}{2} \end{matrix}\middle| z\right) = 2^{- 2 a} z^{\frac{1}{4} - \frac{a}{2}} \left(I_{2 a - 1}\left(4 \sqrt[4]{z}\right) + J_{2 a - 1}\left(4 \sqrt[4]{z}\right)\right) \Gamma\left(2 a\right)$$

$${}_{0}F_{3}\left(\begin{matrix} \\ a, a + \frac{1}{2}, 2 a \end{matrix}\middle| z\right) = \left(2 \sqrt{z} e^{\frac{i \pi}{2}}\right)^{1 - 2 a} I_{2 a - 1}\left(2 \sqrt{2} \sqrt[4]{z} e^{\frac{i \pi}{4}}\right) J_{2 a - 1}\left(2 \sqrt{2} \sqrt[4]{z} e^{\frac{i \pi}{4}}\right) \Gamma^{2}\left(2 a\right)$$

$${}_{1}F_{2}\left(\begin{matrix} a \\ a - \frac{1}{2}, 2 a \end{matrix}\middle| z\right) = 2 \cdot 4^{a - 1} z^{1 - a} I_{a - \frac{3}{2}}\left(\sqrt{z}\right) I_{a - \frac{1}{2}}\left(\sqrt{z}\right) \Gamma\left(a - \frac{1}{2}\right) \Gamma\left(a + \frac{1}{2}\right) - 4^{a - \frac{1}{2}} z^{\frac{1}{2} - a} I^{2}_{a - \frac{1}{2}}\left(\sqrt{z}\right) \Gamma^{2}\left(a + \frac{1}{2}\right)$$

$${}_{1}F_{2}\left(\begin{matrix} \frac{1}{2} \\ b, 2 - b \end{matrix}\middle| z\right) = \frac{\pi \left(1 - b\right) I_{1 - b}\left(\sqrt{z}\right) I_{b - 1}\left(\sqrt{z}\right)}{\sin{\left(b \pi \right)}}$$

$${}_{1}F_{2}\left(\begin{matrix} \frac{1}{2} \\ \frac{3}{2}, \frac{3}{2} \end{matrix}\middle| z\right) = \frac{\operatorname{Shi}{\left(2 \sqrt{z} \right)}}{2 \sqrt{z}}$$

$${}_{1}F_{2}\left(\begin{matrix} \frac{3}{4} \\ \frac{3}{2}, \frac{7}{4} \end{matrix}\middle| z\right) = \frac{3 \sqrt{\pi} e^{- \frac{3 i \pi}{4}} S\left(\frac{2 \sqrt[4]{z} e^{\frac{i \pi}{4}}}{\sqrt{\pi}}\right)}{4 z^{\frac{3}{4}}}$$

$${}_{1}F_{2}\left(\begin{matrix} \frac{1}{4} \\ \frac{1}{2}, \frac{5}{4} \end{matrix}\middle| z\right) = \frac{\sqrt{\pi} e^{- \frac{i \pi}{4}} C\left(\frac{2 \sqrt[4]{z} e^{\frac{i \pi}{4}}}{\sqrt{\pi}}\right)}{2 \sqrt[4]{z}}$$

$${}_{2}F_{3}\left(\begin{matrix} a, a + \frac{1}{2} \\ 2 a, b, 2 a - b + 1 \end{matrix}\middle| z\right) = \left(\frac{\sqrt{z}}{2}\right)^{1 - 2 a} I_{2 a - b}\left(\sqrt{z}\right) I_{b - 1}\left(\sqrt{z}\right) \Gamma\left(b\right) \Gamma\left(2 a - b + 1\right)$$

$${}_{2}F_{3}\left(\begin{matrix} 1, 1 \\ 2, 2, \frac{3}{2} \end{matrix}\middle| z\right) = \frac{- \log{\left(2 \sqrt{z} \right)} + \operatorname{Chi}\left(2 \sqrt{z}\right)}{z} - \frac{\gamma}{z}$$

$${}_{3}F_{3}\left(\begin{matrix} 1, 1, a \\ 2, 2, a + 1 \end{matrix}\middle| z\right) = \frac{a \left(- z\right)^{- a} \left(\Gamma\left(a\right) - \Gamma\left(a, - z\right)\right)}{\left(a - 1\right)^{2}} + \frac{a \left(1 - a\right) \left(\log{\left(- z \right)} + \operatorname{E}_{1}\left(- z\right) + \gamma\right)}{z \left(a^{2} - 2 a + 1\right)} - \frac{a e^{z}}{z \left(a^{2} - 2 a + 1\right)} + \frac{a}{z \left(a^{2} - 2 a + 1\right)}$$

References

Hongguang Fu's Trigonometric Simplification

Implementation of the trigsimp algorithm by Fu et al.

The idea behind the Fu algorithm is to use a sequence of rules that students learn during their pre-calculus courses. The rules are applied heuristically and it uses a greedy algorithm to apply multiple rules simultaneously and choose the result with the least leaf counts.

There are transform rules in which a single rule is applied to the expression tree. The following are just mnemonic in nature; see the docstrings for examples.

- `TR0()` (page 704) - simplify expression

- *TR1()* (page 704) - sec-csc to cos-sin
- *TR2()* (page 704) - tan-cot to sin-cos ratio
- *TR2i()* (page 704) - sin-cos ratio to tan
- *TR3()* (page 705) - angle canonicalization
- *TR4()* (page 705) - functions at special angles
- *TR5()* (page 705) - powers of sin to powers of cos
- *TR6()* (page 706) - powers of cos to powers of sin
- *TR7()* (page 706) - reduce cos power (increase angle)
- *TR8()* (page 706) - expand products of sin-cos to sums
- *TR9()* (page 707) - contract sums of sin-cos to products
- *TR10()* (page 707) - separate sin-cos arguments
- *TR10i()* (page 707) - collect sin-cos arguments
- *TR11()* (page 708) - reduce double angles
- *TR12()* (page 708) - separate tan arguments
- *TR12i()* (page 709) - collect tan arguments
- *TR13()* (page 709) - expand product of tan-cot
- *TRmorrie()* (page 709) - prod(cos(x*2**i), (i, 0, k - 1)) -> sin(2**k*x)/(2**k*sin(x))
- *TR14()* (page 711) - factored powers of sin or cos to cos or sin power
- *TR15()* (page 711) - negative powers of sin to cot power
- *TR16()* (page 711) - negative powers of cos to tan power
- *TR22()* (page 712) - tan-cot powers to negative powers of sec-csc functions
- *TR111()* (page 711) - negative sin-cos-tan powers to csc-sec-cot

There are 4 combination transforms (CTR1 - CTR4) in which a sequence of transformations are applied and the simplest expression is selected from a few options.

Finally, there are the 2 rule lists (RL1 and RL2), which apply a sequence of transformations and combined transformations, and the `fu` algorithm itself, which applies rules and rule lists and selects the best expressions. There is also a function `L` which counts the number of trigonometric functions that appear in the expression.

Other than TR0, re-writing of expressions is not done by the transformations. e.g. TR10i finds pairs of terms in a sum that are in the form like `cos(x)*cos(y) + sin(x)*sin(y)`. Such expression are targeted in a bottom-up traversal of the expression, but no manipulation to make them appear is attempted. For example,

Set-up for examples below:

```
>>> from sympy.simplify.fu import fu, L, TR9, TR10i, TR11
>>> from sympy import factor, sin, cos, powsimp
>>> from sympy.abc import x, y, z, a
>>> from time import time
>>> eq = cos(x + y)/cos(x)
>>> TR10i(eq.expand(trig=True))
-sin(x)*sin(y)/cos(x) + cos(y)
```

If the expression is put in "normal" form (with a common denominator) then the transformation is successful:

```
>>> TR10i(_.normal())
cos(x + y)/cos(x)
```

TR11's behavior is similar. It rewrites double angles as smaller angles but doesn't do any simplification of the result.

```
>>> TR11(sin(2)**a*cos(1)**(-a), 1)
(2*sin(1)*cos(1))**a/cos(1)**a
>>> powsimp(_)
(2*sin(1))**a
```

The temptation is to try make these TR rules "smarter" but that should really be done at a higher level; the TR rules should try maintain the "do one thing well" principle. There is one exception, however. In TR10i and TR9 terms are recognized even when they are each multiplied by a common factor:

```
>>> fu(a*cos(x)*cos(y) + a*sin(x)*sin(y))
a*cos(x - y)
```

Factoring with `factor_terms` is used but it is "JIT"-like, being delayed until it is deemed necessary. Furthermore, if the factoring does not help with the simplification, it is not retained, so `a*cos(x)*cos(y) + a*sin(x)*sin(z)` does not become a factored (but unsimplified in the trigonometric sense) expression:

```
>>> fu(a*cos(x)*cos(y) + a*sin(x)*sin(z))
a*sin(x)*sin(z) + a*cos(x)*cos(y)
```

In some cases factoring might be a good idea, but the user is left to make that decision. For example:

```
>>> expr=((15*sin(2*x) + 19*sin(x + y) + 17*sin(x + z) + 19*cos(x - z) +
... 25)*(20*sin(2*x) + 15*sin(x + y) + sin(y + z) + 14*cos(x - z) +
... 14*cos(y - z))*(9*sin(2*y) + 12*sin(y + z) + 10*cos(x - y) + 2*cos(y -
... z) + 18)).expand(trig=True).expand()
```

In the expanded state, there are nearly 1000 trig functions:

```
>>> L(expr)
932
```

If the expression where factored first, this would take time but the resulting expression would be transformed very quickly:

```
>>> def clock(f, n=2):
...    t=time(); f(); return round(time()-t, n)
...
>>> clock(lambda: factor(expr))
```

(continues on next page)

(continued from previous page)

```
0.86
>>> clock(lambda: TR10i(expr), 3)
0.016
```

If the unexpanded expression is used, the transformation takes longer but not as long as it took to factor it and then transform it:

```
>>> clock(lambda: TR10i(expr), 2)
0.28
```

So neither expansion nor factoring is used in `TR10i`: if the expression is already factored (or partially factored) then expansion with `trig=True` would destroy what is already known and take longer; if the expression is expanded, factoring may take longer than simply applying the transformation itself.

Although the algorithms should be canonical, always giving the same result, they may not yield the best result. This, in general, is the nature of simplification where searching all possible transformation paths is very expensive. Here is a simple example. There are 6 terms in the following sum:

```
>>> expr = (sin(x)**2*cos(y)*cos(z) + sin(x)*sin(y)*cos(x)*cos(z) +
... sin(x)*sin(z)*cos(x)*cos(y) + sin(y)*sin(z)*cos(x)**2 + sin(y)*sin(z) +
... cos(y)*cos(z))
>>> args = expr.args
```

Serendipitously, fu gives the best result:

```
>>> fu(expr)
3*cos(y - z)/2 - cos(2*x + y + z)/2
```

But if different terms were combined, a less-optimal result might be obtained, requiring some additional work to get better simplification, but still less than optimal. The following shows an alternative form of `expr` that resists optimal simplification once a given step is taken since it leads to a dead end:

```
>>> TR9(-cos(x)**2*cos(y + z) + 3*cos(y - z)/2 +
...     cos(y + z)/2 + cos(-2*x + y + z)/4 - cos(2*x + y + z)/4)
sin(2*x)*sin(y + z)/2 - cos(x)**2*cos(y + z) + 3*cos(y - z)/2 + cos(y + z)/2
```

Here is a smaller expression that exhibits the same behavior:

```
>>> a = sin(x)*sin(z)*cos(x)*cos(y) + sin(x)*sin(y)*cos(x)*cos(z)
>>> TR10i(a)
sin(x)*sin(y + z)*cos(x)
>>> newa = _
>>> TR10i(expr - a)  # this combines two more of the remaining terms
sin(x)**2*cos(y)*cos(z) + sin(y)*sin(z)*cos(x)**2 + cos(y - z)
>>> TR10i(_ + newa) == _ + newa  # but now there is no more simplification
True
```

Without getting lucky or trying all possible pairings of arguments, the final result may be less than optimal and impossible to find without better heuristics or brute force trial of all possibilities.

Rules

sympy.simplify.fu.**TR0**(*rv*)

Simplification of rational polynomials, trying to simplify the expression, e.g. combine things like 3*x + 2*x, etc....

sympy.simplify.fu.**TR1**(*rv*)

Replace sec, csc with 1/cos, 1/sin

Examples

```
>>> from sympy.simplify.fu import TR1, sec, csc
>>> from sympy.abc import x
>>> TR1(2*csc(x) + sec(x))
1/cos(x) + 2/sin(x)
```

sympy.simplify.fu.**TR2**(*rv*)

Replace tan and cot with sin/cos and cos/sin

Examples

```
>>> from sympy.simplify.fu import TR2
>>> from sympy.abc import x
>>> from sympy import tan, cot, sin, cos
>>> TR2(tan(x))
sin(x)/cos(x)
>>> TR2(cot(x))
cos(x)/sin(x)
>>> TR2(tan(tan(x) - sin(x)/cos(x)))
0
```

sympy.simplify.fu.**TR2i**(*rv*, *half=False*)

Converts ratios involving sin and cos as follows::

sin(x)/cos(x) -> tan(x) sin(x)/(cos(x) + 1) -> tan(x/2) if half=True

Examples

```
>>> from sympy.simplify.fu import TR2i
>>> from sympy.abc import x, a
>>> from sympy import sin, cos
>>> TR2i(sin(x)/cos(x))
tan(x)
```

Powers of the numerator and denominator are also recognized

```
>>> TR2i(sin(x)**2/(cos(x) + 1)**2, half=True)
tan(x/2)**2
```

The transformation does not take place unless assumptions allow (i.e. the base must be positive or the exponent must be an integer for both numerator and denominator)

```
>>> TR2i(sin(x)**a/(cos(x) + 1)**a)
sin(x)**a/(cos(x) + 1)**a
```

sympy.simplify.fu.**TR3**(*rv*)

Induced formula: example sin(-a) = -sin(a)

Examples

```
>>> from sympy.simplify.fu import TR3
>>> from sympy.abc import x, y
>>> from sympy import pi
>>> from sympy import cos
>>> TR3(cos(y - x*(y - x)))
cos(x*(x - y) + y)
>>> cos(pi/2 + x)
-sin(x)
>>> cos(30*pi/2 + x)
-cos(x)
```

sympy.simplify.fu.**TR4**(*rv*)

Identify values of special angles.

A= 0 Pi/6 Pi/4 Pi/3 Pi/2

sin(a) 0 1/2 sqrt(2)/2 sqrt(3)/2 1 cos(a) 1 sqrt(3)/2 sqrt(2)/2 1/2 0 tan(a) 0 sqt(3)/3 1 sqrt(3) –

Examples

```
>>> from sympy import pi
>>> from sympy import cos, sin, tan, cot
>>> for s in (0, pi/6, pi/4, pi/3, pi/2):
...    print('%s %s %s %s' % (cos(s), sin(s), tan(s), cot(s)))
...
1 0 0 zoo
sqrt(3)/2 1/2 sqrt(3)/3 sqrt(3)
sqrt(2)/2 sqrt(2)/2 1 1
1/2 sqrt(3)/2 sqrt(3) sqrt(3)/3
0 1 zoo 0
```

sympy.simplify.fu.**TR5**(*rv*, *max=4*, *pow=False*)

Replacement of sin**2 with 1 - cos(x)**2.

See _TR56 docstring for advanced use of `max` and `pow`.

Examples

```
>>> from sympy.simplify.fu import TR5
>>> from sympy.abc import x
>>> from sympy import sin
>>> TR5(sin(x)**2)
1 - cos(x)**2
>>> TR5(sin(x)**-2)  # unchanged
sin(x)**(-2)
>>> TR5(sin(x)**4)
(1 - cos(x)**2)**2
```

`sympy.simplify.fu.`**`TR6`**(*rv*, *max=4*, *pow=False*)

Replacement of cos**2 with 1 - sin(x)**2.

See _TR56 docstring for advanced use of `max` and `pow`.

Examples

```
>>> from sympy.simplify.fu import TR6
>>> from sympy.abc import x
>>> from sympy import cos
>>> TR6(cos(x)**2)
1 - sin(x)**2
>>> TR6(cos(x)**-2)  #unchanged
cos(x)**(-2)
>>> TR6(cos(x)**4)
(1 - sin(x)**2)**2
```

`sympy.simplify.fu.`**`TR7`**(*rv*)

Lowering the degree of cos(x)**2.

Examples

```
>>> from sympy.simplify.fu import TR7
>>> from sympy.abc import x
>>> from sympy import cos
>>> TR7(cos(x)**2)
cos(2*x)/2 + 1/2
>>> TR7(cos(x)**2 + 1)
cos(2*x)/2 + 3/2
```

`sympy.simplify.fu.`**`TR8`**(*rv*, *first=True*)

Converting products of `cos` and/or `sin` to a sum or difference of `cos` and or `sin` terms.

Examples

```
>>> from sympy.simplify.fu import TR8
>>> from sympy import cos, sin
>>> TR8(cos(2)*cos(3))
cos(5)/2 + cos(1)/2
>>> TR8(cos(2)*sin(3))
sin(5)/2 + sin(1)/2
>>> TR8(sin(2)*sin(3))
-cos(5)/2 + cos(1)/2
```

sympy.simplify.fu.**TR9**(*rv*)

Sum of cos or sin terms as a product of cos or sin.

Examples

```
>>> from sympy.simplify.fu import TR9
>>> from sympy import cos, sin
>>> TR9(cos(1) + cos(2))
2*cos(1/2)*cos(3/2)
>>> TR9(cos(1) + 2*sin(1) + 2*sin(2))
cos(1) + 4*sin(3/2)*cos(1/2)
```

If no change is made by TR9, no re-arrangement of the expression will be made. For example, though factoring of common term is attempted, if the factored expression was not changed, the original expression will be returned:

```
>>> TR9(cos(3) + cos(3)*cos(2))
cos(3) + cos(2)*cos(3)
```

sympy.simplify.fu.**TR10**(*rv*, *first=True*)

Separate sums in cos and sin.

Examples

```
>>> from sympy.simplify.fu import TR10
>>> from sympy.abc import a, b, c
>>> from sympy import cos, sin
>>> TR10(cos(a + b))
-sin(a)*sin(b) + cos(a)*cos(b)
>>> TR10(sin(a + b))
sin(a)*cos(b) + sin(b)*cos(a)
>>> TR10(sin(a + b + c))
(-sin(a)*sin(b) + cos(a)*cos(b))*sin(c) +     (sin(a)*cos(b) +
 ↪sin(b)*cos(a))*cos(c)
```

sympy.simplify.fu.**TR10i**(*rv*)

Sum of products to function of sum.

Examples

```
>>> from sympy.simplify.fu import TR10i
>>> from sympy import cos, sin, sqrt
>>> from sympy.abc import x
```

```
>>> TR10i(cos(1)*cos(3) + sin(1)*sin(3))
cos(2)
>>> TR10i(cos(1)*sin(3) + sin(1)*cos(3) + cos(3))
cos(3) + sin(4)
>>> TR10i(sqrt(2)*cos(x)*x + sqrt(6)*sin(x)*x)
2*sqrt(2)*x*sin(x + pi/6)
```

sympy.simplify.fu.**TR11**(*rv*, *base=None*)

Function of double angle to product. The base argument can be used to indicate what is the un-doubled argument, e.g. if 3*pi/7 is the base then cosine and sine functions with argument 6*pi/7 will be replaced.

Examples

```
>>> from sympy.simplify.fu import TR11
>>> from sympy import cos, sin, pi
>>> from sympy.abc import x
>>> TR11(sin(2*x))
2*sin(x)*cos(x)
>>> TR11(cos(2*x))
-sin(x)**2 + cos(x)**2
>>> TR11(sin(4*x))
4*(-sin(x)**2 + cos(x)**2)*sin(x)*cos(x)
>>> TR11(sin(4*x/3))
4*(-sin(x/3)**2 + cos(x/3)**2)*sin(x/3)*cos(x/3)
```

If the arguments are simply integers, no change is made unless a base is provided:

```
>>> TR11(cos(2))
cos(2)
>>> TR11(cos(4), 2)
-sin(2)**2 + cos(2)**2
```

There is a subtle issue here in that autosimplification will convert some higher angles to lower angles

```
>>> cos(6*pi/7) + cos(3*pi/7)
-cos(pi/7) + cos(3*pi/7)
```

The 6*pi/7 angle is now pi/7 but can be targeted with TR11 by supplying the 3*pi/7 base:

```
>>> TR11(_, 3*pi/7)
-sin(3*pi/7)**2 + cos(3*pi/7)**2 + cos(3*pi/7)
```

sympy.simplify.fu.**TR12**(*rv*, *first=True*)

Separate sums in tan.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy import tan
>>> from sympy.simplify.fu import TR12
>>> TR12(tan(x + y))
(tan(x) + tan(y))/(-tan(x)*tan(y) + 1)
```

sympy.simplify.fu.**TR12i**(*rv*)

Combine tan arguments as (tan(y) + tan(x))/(tan(x)*tan(y) - 1) -> -tan(x + y).

Examples

```
>>> from sympy.simplify.fu import TR12i
>>> from sympy import tan
>>> from sympy.abc import a, b, c
>>> ta, tb, tc = [tan(i) for i in (a, b, c)]
>>> TR12i((ta + tb)/(-ta*tb + 1))
tan(a + b)
>>> TR12i((ta + tb)/(ta*tb - 1))
-tan(a + b)
>>> TR12i((-ta - tb)/(ta*tb - 1))
tan(a + b)
>>> eq = (ta + tb)/(-ta*tb + 1)**2*(-3*ta - 3*tc)/(2*(ta*tc - 1))
>>> TR12i(eq.expand())
-3*tan(a + b)*tan(a + c)/(2*(tan(a) + tan(b) - 1))
```

sympy.simplify.fu.**TR13**(*rv*)

Change products of tan or cot.

Examples

```
>>> from sympy.simplify.fu import TR13
>>> from sympy import tan, cot
>>> TR13(tan(3)*tan(2))
-tan(2)/tan(5) - tan(3)/tan(5) + 1
>>> TR13(cot(3)*cot(2))
cot(2)*cot(5) + 1 + cot(3)*cot(5)
```

sympy.simplify.fu.**TRmorrie**(*rv*)

Returns cos(x)*cos(2*x)*...*cos(2**(k-1)*x) -> sin(2**k*x)/(2**k*sin(x))

Examples

```
>>> from sympy.simplify.fu import TRmorrie, TR8, TR3
>>> from sympy.abc import x
>>> from sympy import Mul, cos, pi
>>> TRmorrie(cos(x)*cos(2*x))
sin(4*x)/(4*sin(x))
>>> TRmorrie(7*Mul(*[cos(x) for x in range(10)]))
7*sin(12)*sin(16)*cos(5)*cos(7)*cos(9)/(64*sin(1)*sin(3))
```

Sometimes autosimplification will cause a power to be not recognized. e.g. in the following, cos(4*pi/7) automatically simplifies to -cos(3*pi/7) so only 2 of the 3 terms are recognized:

```
>>> TRmorrie(cos(pi/7)*cos(2*pi/7)*cos(4*pi/7))
-sin(3*pi/7)*cos(3*pi/7)/(4*sin(pi/7))
```

A touch by TR8 resolves the expression to a Rational

```
>>> TR8(_)
-1/8
```

In this case, if eq is unsimplified, the answer is obtained directly:

```
>>> eq = cos(pi/9)*cos(2*pi/9)*cos(3*pi/9)*cos(4*pi/9)
>>> TRmorrie(eq)
1/16
```

But if angles are made canonical with TR3 then the answer is not simplified without further work:

```
>>> TR3(eq)
sin(pi/18)*cos(pi/9)*cos(2*pi/9)/2
>>> TRmorrie(_)
sin(pi/18)*sin(4*pi/9)/(8*sin(pi/9))
>>> TR8(_)
cos(7*pi/18)/(16*sin(pi/9))
>>> TR3(_)
1/16
```

The original expression would have resolve to 1/16 directly with TR8, however:

```
>>> TR8(eq)
1/16
```

References

[R759]

sympy.simplify.fu.**TR14**(*rv*, *first=True*)

Convert factored powers of sin and cos identities into simpler expressions.

Examples

```
>>> from sympy.simplify.fu import TR14
>>> from sympy.abc import x, y
>>> from sympy import cos, sin
>>> TR14((cos(x) - 1)*(cos(x) + 1))
-sin(x)**2
>>> TR14((sin(x) - 1)*(sin(x) + 1))
-cos(x)**2
>>> p1 = (cos(x) + 1)*(cos(x) - 1)
>>> p2 = (cos(y) - 1)*2*(cos(y) + 1)
>>> p3 = (3*(cos(y) - 1))*(3*(cos(y) + 1))
>>> TR14(p1*p2*p3*(x - 1))
-18*(x - 1)*sin(x)**2*sin(y)**4
```

sympy.simplify.fu.**TR15**(*rv*, *max=4*, *pow=False*)

Convert sin(x)**-2 to 1 + cot(x)**2.

See _TR56 docstring for advanced use of `max` and `pow`.

Examples

```
>>> from sympy.simplify.fu import TR15
>>> from sympy.abc import x
>>> from sympy import sin
>>> TR15(1 - 1/sin(x)**2)
-cot(x)**2
```

sympy.simplify.fu.**TR16**(*rv*, *max=4*, *pow=False*)

Convert cos(x)**-2 to 1 + tan(x)**2.

See _TR56 docstring for advanced use of `max` and `pow`.

Examples

```
>>> from sympy.simplify.fu import TR16
>>> from sympy.abc import x
>>> from sympy import cos
>>> TR16(1 - 1/cos(x)**2)
-tan(x)**2
```

sympy.simplify.fu.**TR111**(*rv*)

Convert f(x)**-i to g(x)**i where either i is an integer or the base is positive and f, g are: tan, cot; sin, csc; or cos, sec.

Examples

```
>>> from sympy.simplify.fu import TR111
>>> from sympy.abc import x
>>> from sympy import tan
>>> TR111(1 - 1/tan(x)**2)
1 - cot(x)**2
```

sympy.simplify.fu.**TR22**(*rv*, *max=4*, *pow=False*)

Convert tan(x)**2 to sec(x)**2 - 1 and cot(x)**2 to csc(x)**2 - 1.

See _TR56 docstring for advanced use of max and pow.

Examples

```
>>> from sympy.simplify.fu import TR22
>>> from sympy.abc import x
>>> from sympy import tan, cot
>>> TR22(1 + tan(x)**2)
sec(x)**2
>>> TR22(1 + cot(x)**2)
csc(x)**2
```

sympy.simplify.fu.**TRpower**(*rv*)

Convert sin(x)**n and cos(x)**n with positive n to sums.

Examples

```
>>> from sympy.simplify.fu import TRpower
>>> from sympy.abc import x
>>> from sympy import cos, sin
>>> TRpower(sin(x)**6)
-15*cos(2*x)/32 + 3*cos(4*x)/16 - cos(6*x)/32 + 5/16
>>> TRpower(sin(x)**3*cos(2*x)**4)
(3*sin(x)/4 - sin(3*x)/4)*(cos(4*x)/2 + cos(8*x)/8 + 3/8)
```

References

[R760]

sympy.simplify.fu.**fu**(*rv*, *measure=<function <lambda>>*)

Attempt to simplify expression by using transformation rules given in the algorithm by Fu et al.

fu() (page 712) will try to minimize the objective function measure. By default this first minimizes the number of trig terms and then minimizes the number of total operations.

Examples

```
>>> from sympy.simplify.fu import fu
>>> from sympy import cos, sin, tan, pi, S, sqrt
>>> from sympy.abc import x, y, a, b
```

```
>>> fu(sin(50)**2 + cos(50)**2 + sin(pi/6))
3/2
>>> fu(sqrt(6)*cos(x) + sqrt(2)*sin(x))
2*sqrt(2)*sin(x + pi/3)
```

CTR1 example

```
>>> eq = sin(x)**4 - cos(y)**2 + sin(y)**2 + 2*cos(x)**2
>>> fu(eq)
cos(x)**4 - 2*cos(y)**2 + 2
```

CTR2 example

```
>>> fu(S.Half - cos(2*x)/2)
sin(x)**2
```

CTR3 example

```
>>> fu(sin(a)*(cos(b) - sin(b)) + cos(a)*(sin(b) + cos(b)))
sqrt(2)*sin(a + b + pi/4)
```

CTR4 example

```
>>> fu(sqrt(3)*cos(x)/2 + sin(x)/2)
sin(x + pi/3)
```

Example 1

```
>>> fu(1-sin(2*x)**2/4-sin(y)**2-cos(x)**4)
-cos(x)**2 + cos(y)**2
```

Example 2

```
>>> fu(cos(4*pi/9))
sin(pi/18)
>>> fu(cos(pi/9)*cos(2*pi/9)*cos(3*pi/9)*cos(4*pi/9))
1/16
```

Example 3

```
>>> fu(tan(7*pi/18)+tan(5*pi/18)-sqrt(3)*tan(5*pi/18)*tan(7*pi/18))
-sqrt(3)
```

Objective function example

```
>>> fu(sin(x)/cos(x))  # default objective function
tan(x)
>>> fu(sin(x)/cos(x), measure=lambda x: -x.count_ops()) # maximize op␣
```

(continues on next page)

(continued from previous page)

```
↪count
sin(x)/cos(x)
```

References

[R761]

Notes

This work was started by Dimitar Vlahovski at the Technological School "Electronic systems" (30.11.2011).

Beyond TR13, other rules are not from the original paper, but extended in SymPy.

References

Solvers

This module documentation contains details about the `sympy.solvers` module. functions.

Contents

Diophantine

Diophantine equations

The word "Diophantine" comes with the name Diophantus, a mathematician lived in the great city of Alexandria sometime around 250 AD. Often referred to as the "father of Algebra", Diophantus in his famous work "Arithmetica" presented 150 problems that marked the early beginnings of number theory, the field of study about integers and their properties. Diophantine equations play a central and an important part in number theory.

We call a "Diophantine equation" to an equation of the form, $f(x_1, x_2, \ldots x_n) = 0$ where $n \geq 2$ and $x_1, x_2, \ldots x_n$ are integer variables. If we can find n integers $a_1, a_2, \ldots a_n$ such that $x_1 = a_1, x_2 = a_2, \ldots x_n = a_n$ satisfies the above equation, we say that the equation is solvable. You can read more about Diophantine equations in[1] and[2].

Currently, following five types of Diophantine equations can be solved using *diophantine()* (page 720) and other helper functions of the Diophantine module.

- Linear Diophantine equations: $a_1x_1 + a_2x_2 + \ldots + a_nx_n = b$.
- General binary quadratic equation: $ax^2 + bxy + cy^2 + dx + ey + f = 0$
- Homogeneous ternary quadratic equation: $ax^2 + by^2 + cz^2 + dxy + eyz + fzx = 0$
- Extended Pythagorean equation: $a_1x_1^2 + a_2x_2^2 + \ldots + a_nx_n^2 = a_{n+1}x_{n+1}^2$

[1] Andreescu, Titu. Andrica, Dorin. Cucurezeanu, Ion. An Introduction to Diophantine Equations

[2] Diophantine Equation, Wolfram Mathworld, [online]. Available: http://mathworld.wolfram.com/DiophantineEquation.html

- General sum of squares: $x_1^2 + x_2^2 + \ldots + x_n^2 = k$

Module structure

This module contains *diophantine()* (page 720) and helper functions that are needed to solve certain Diophantine equations. It's structured in the following manner.

- *diophantine()* (page 720)
  - *diop_solve()* (page 721)
    - *classify_diop()* (page 721)
    - *diop_linear()* (page 722)
    - *diop_quadratic()* (page 723)
    - *diop_ternary_quadratic()* (page 729)
    - *diop_ternary_quadratic_normal()* (page 741)
    - *diop_general_pythagorean()* (page 731)
    - *diop_general_sum_of_squares()* (page 731)
    - *diop_general_sum_of_even_powers()* (page 732)
  - *merge_solution()* (page 737)

When an equation is given to *diophantine()* (page 720), it factors the equation(if possible) and solves the equation given by each factor by calling *diop_solve()* (page 721) separately. Then all the results are combined using *merge_solution()* (page 737).

diop_solve() (page 721) internally uses *classify_diop()* (page 721) to find the type of the equation(and some other details) given to it and then calls the appropriate solver function based on the type returned. For example, if *classify_diop()* (page 721) returned "linear" as the type of the equation, then *diop_solve()* (page 721) calls *diop_linear()* (page 722) to solve the equation.

Each of the functions, *diop_linear()* (page 722), *diop_quadratic()* (page 723), *diop_ternary_quadratic()* (page 729), *diop_general_pythagorean()* (page 731) and *diop_general_sum_of_squares()* (page 731) solves a specific type of equations and the type can be easily guessed by it's name.

Apart from these functions, there are a considerable number of other functions in the "Diophantine Module" and all of them are listed under User functions and Internal functions.

Tutorial

First, let's import the highest API of the Diophantine module.

```
>>> from sympy.solvers.diophantine import diophantine
```

Before we start solving the equations, we need to define the variables.

```
>>> from sympy import symbols
>>> x, y, z = symbols("x, y, z", integer=True)
```

Let's start by solving the easiest type of Diophantine equations, i.e. linear Diophantine equations. Let's solve $2x + 3y = 5$. Note that although we write the equation in the above form, when we input the equation to any of the functions in Diophantine module, it needs to be in the form $eq = 0$.

```
>>> diophantine(2*x + 3*y - 5)
{(3*t_0 - 5, 5 - 2*t_0)}
```

Note that stepping one more level below the highest API, we can solve the very same equation by calling *diop_solve()* (page 721).

```
>>> from sympy.solvers.diophantine.diophantine import diop_solve
>>> diop_solve(2*x + 3*y - 5)
(3*t_0 - 5, 5 - 2*t_0)
```

Note that it returns a tuple rather than a set. *diophantine()* (page 720) always return a set of tuples. But *diop_solve()* (page 721) may return a single tuple or a set of tuples depending on the type of the equation given.

We can also solve this equation by calling *diop_linear()* (page 722), which is what *diop_solve()* (page 721) calls internally.

```
>>> from sympy.solvers.diophantine.diophantine import diop_linear
>>> diop_linear(2*x + 3*y - 5)
(3*t_0 - 5, 5 - 2*t_0)
```

If the given equation has no solutions then the outputs will look like below.

```
>>> diophantine(2*x + 4*y - 3)
set()
>>> diop_solve(2*x + 4*y - 3)
(None, None)
>>> diop_linear(2*x + 4*y - 3)
(None, None)
```

Note that except for the highest level API, in case of no solutions, a tuple of $None$ are returned. Size of the tuple is the same as the number of variables. Also, one can specifically set the parameter to be used in the solutions by passing a customized parameter. Consider the following example:

```
>>> m = symbols("m", integer=True)
>>> diop_solve(2*x + 3*y - 5, m)
(3*m_0 - 5, 5 - 2*m_0)
```

For linear Diophantine equations, the customized parameter is the prefix used for each free variable in the solution. Consider the following example:

```
>>> diop_solve(2*x + 3*y - 5*z + 7, m)
(m_0, m_0 + 5*m_1 - 14, m_0 + 3*m_1 - 7)
```

In the solution above, m_0 and m_1 are independent free variables.

Please note that for the moment, users can set the parameter only for linear Diophantine equations and binary quadratic equations.

Let's try solving a binary quadratic equation which is an equation with two variables and has a degree of two. Before trying to solve these equations, an idea about various cases