```
>>> list(ordered([y + 2, x + 2, x**2 + y + 3]))
...
[x + 2, y + 2, x**2 + y + 3]
```

Here, sequences are sorted by length, then sum:

```
>>> seq, keys = [[[1, 2, 1], [0, 3, 1], [1, 1, 3], [2], [1]], [
...     lambda x: len(x),
...     lambda x: sum(x)]]
...
>>> list(ordered(seq, keys, default=False, warn=False))
[[1], [2], [1, 2, 1], [0, 3, 1], [1, 1, 3]]
```

If warn is True, an error will be raised if there were not enough keys to break ties:

```
>>> list(ordered(seq, keys, default=False, warn=True))
Traceback (most recent call last):
...
ValueError: not enough keys to break ties
```

Notes

The decorated sort is one of the fastest ways to sort a sequence for which special item comparison is desired: the sequence is decorated, sorted on the basis of the decoration (e.g. making all letters lower case) and then undecorated. If one wants to break ties for items that have the same decorated value, a second key can be used. But if the second key is expensive to compute then it is inefficient to decorate all items with both keys: only those items having identical first key values need to be decorated. This function applies keys successively only when needed to break ties. By yielding an iterator, use of the tie-breaker is delayed as long as possible.

This function is best used in cases when use of the first key is expected to be a good hashing function; if there are no unique hashes from application of a key, then that key should not have been used. The exception, however, is that even if there are many collisions, if the first group is small and one does not need to process all items in the list then time will not be wasted sorting what one was not interested in. For example, if one were looking for the minimum in a list and there were several criteria used to define the sort order, then this function would be good at returning that quickly if the first group of candidates is small relative to the number of items being processed.

Random

When you need to use random numbers in SymPy library code, import from here so there is only one generator working for SymPy. Imports from here should behave the same as if they were being imported from Python's random module. But only the routines currently used in SymPy are included here. To use others import `rng` and access the method directly. For example, to capture the current state of the generator use `rng.getstate()`.

There is intentionally no Random to import from here. If you want to control the state of the generator, import `seed` and call it with or without an argument to set the state.

Examples

```
>>> from sympy.core.random import random, seed
>>> assert random() < 1
>>> seed(1); a = random()
>>> b = random()
>>> seed(1); c = random()
>>> assert a == c
>>> assert a != b  # remote possibility this will fail
```

random_complex_number

sympy.core.random.**random_complex_number**(*a=2, b=-1, c=3, d=1, rational=False, tolerance=None*)

Return a random complex number.

To reduce chance of hitting branch cuts or anything, we guarantee b <= Im z <= d, a <= Re z <= c

When rational is True, a rational approximation to a random number is obtained within specified tolerance, if any.

verify_numerically

sympy.core.random.**verify_numerically**(*f, g, z=None, tol=1e-06, a=2, b=-1, c=3, d=1*)

Test numerically that f and g agree when evaluated in the argument z.

If z is None, all symbols will be tested. This routine does not test whether there are Floats present with precision higher than 15 digits so if there are, your results may not be what you expect due to round- off errors.

Examples

```
>>> from sympy import sin, cos
>>> from sympy.abc import x
>>> from sympy.core.random import verify_numerically as tn
>>> tn(sin(x)**2 + cos(x)**2, 1, x)
True
```

test_derivative_numerically

sympy.core.random.**test_derivative_numerically**(*f, z, tol=1e-06, a=2, b=-1, c=3, d=1*)

Test numerically that the symbolically computed derivative of f with respect to z is correct.

This routine does not test whether there are Floats present with precision higher than 15 digits so if there are, your results may not be what you expect due to round-off errors.

Examples

```
>>> from sympy import sin
>>> from sympy.abc import x
>>> from sympy.core.random import test_derivative_numerically as td
>>> td(sin(x), x)
True
```

_randrange

sympy.core.random.**_randrange**(*seed=None*)

Return a randrange generator.

seed can be

- None - return randomly seeded generator
- int - return a generator seeded with the int
- list - the values to be returned will be taken from the list in the order given; the provided list is not modified.

Examples

```
>>> from sympy.core.random import _randrange
>>> rr = _randrange()
>>> rr(1000)
999
>>> rr = _randrange(3)
>>> rr(1000)
238
>>> rr = _randrange([0, 5, 1, 3, 4])
>>> rr(3), rr(3)
(0, 1)
```

_randint

sympy.core.random.**_randint**(*seed=None*)

Return a randint generator.

seed can be

- None - return randomly seeded generator
- int - return a generator seeded with the int
- list - the values to be returned will be taken from the list in the order given; the provided list is not modified.

Examples

```
>>> from sympy.core.random import _randint
>>> ri = _randint()
>>> ri(1, 1000)
999
>>> ri = _randint(3)
>>> ri(1, 1000)
238
>>> ri = _randint([0, 5, 1, 2, 4])
>>> ri(1, 3), ri(1, 3)
(1, 2)
```

Traversal

bottom_up

`sympy.core.traversal.`**`bottom_up`**(*rv, F, atoms=False, nonbasic=False*)

Apply `F` to all expressions in an expression tree from the bottom up. If `atoms` is True, apply `F` even if there are no args; if `nonbasic` is True, try to apply `F` to non-Basic objects.

postorder_traversal

`sympy.core.traversal.`**`postorder_traversal`**(*node, keys=None*)

Do a postorder traversal of a tree.

This generator recursively yields nodes that it has visited in a postorder fashion. That is, it descends through the tree depth-first to yield all of a node's children's postorder traversal before yielding the node itself.

Parameters

node : SymPy expression

The expression to traverse.

keys : (default None) sort key(s)

The key(s) used to sort args of Basic objects. When None, args of Basic objects are processed in arbitrary order. If key is defined, it will be passed along to ordered() as the only key(s) to use to sort the arguments; if `key` is simply True then the default keys of `ordered` will be used (node count and default_sort_key).

Yields

subtree : SymPy expression

All of the subtrees in the tree.

Examples

```
>>> from sympy import postorder_traversal
>>> from sympy.abc import w, x, y, z
```

The nodes are returned in the order that they are encountered unless key is given; simply passing key=True will guarantee that the traversal is unique.

```
>>> list(postorder_traversal(w + (x + y)*z))
[z, y, x, x + y, z*(x + y), w, w + z*(x + y)]
>>> list(postorder_traversal(w + (x + y)*z, keys=True))
[w, z, x, y, x + y, z*(x + y), w + z*(x + y)]
```

preorder_traversal

sympy.core.traversal.**preorder_traversal**(*node*, *keys=None*)

Do a pre-order traversal of a tree.

This iterator recursively yields nodes that it has visited in a pre-order fashion. That is, it yields the current node then descends through the tree breadth-first to yield all of a node's children's pre-order traversal.

For an expression, the order of the traversal depends on the order of .args, which in many cases can be arbitrary.

Parameters

node : SymPy expression

The expression to traverse.

keys : (default None) sort key(s)

The key(s) used to sort args of Basic objects. When None, args of Basic objects are processed in arbitrary order. If key is defined, it will be passed along to ordered() as the only key(s) to use to sort the arguments; if key is simply True then the default keys of ordered will be used.

Yields

subtree : SymPy expression

All of the subtrees in the tree.

Examples

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

The nodes are returned in the order that they are encountered unless key is given; simply passing key=True will guarantee that the traversal is unique.

```
>>> list(preorder_traversal((x + y)*z, keys=None))
[z*(x + y), z, x + y, y, x]
```

(continues on next page)

(continued from previous page)

```
>>> list(preorder_traversal((x + y)*z, keys=True))
[z*(x + y), z, x + y, x, y]
```

use

sympy.core.traversal.**use**(*expr, func, level=0, args=(), kwargs={}*)

Use func to transform expr at the given level.

Examples

```
>>> from sympy import use, expand
>>> from sympy.abc import x, y
```

```
>>> f = (x + y)**2*x + 1
```

```
>>> use(f, expand, level=2)
x*(x**2 + 2*x*y + y**2) + 1
>>> expand(f)
x**3 + 2*x**2*y + x*y**2 + 1
```

walk

sympy.core.traversal.**walk**(*e, *target*)

Iterate through the args that are the given types (target) and return a list of the args that were traversed; arguments that are not of the specified types are not traversed.

Examples

```
>>> from sympy.core.traversal import walk
>>> from sympy import Min, Max
>>> from sympy.abc import x, y, z
>>> list(walk(Min(x, Max(y, Min(1, z))), Min))
[Min(x, Max(y, Min(1, z)))]
>>> list(walk(Min(x, Max(y, Min(1, z))), Min, Max))
[Min(x, Max(y, Min(1, z))), Max(y, Min(1, z)), Min(1, z)]
```

See also:

bottom_up (page 1080)

Discrete

The `discrete` module in SymPy implements methods to compute discrete transforms and convolutions of finite sequences.

This module contains functions which operate on discrete sequences.

Transforms - `fft`, `ifft`, `ntt`, `intt`, `fwht`, `ifwht`,
`mobius_transform`, `inverse_mobius_transform`

Convolutions - `convolution`, `convolution_fft`, `convolution_ntt`,
`convolution_fwht`, `convolution_subset`, `covering_product`, `intersecting_product`

Since the discrete transforms can be used to reduce the computational complexity of the discrete convolutions, the `convolutions` module makes use of the `transforms` module for efficient computation (notable for long input sequences).

Transforms

This section lists the methods which implement the basic transforms for discrete sequences.

Fast Fourier Transform

`sympy.discrete.transforms.`**`fft`**`(`*seq, dps=None*`)`

Performs the Discrete Fourier Transform (**DFT**) in the complex domain.

The sequence is automatically padded to the right with zeros, as the *radix-2 FFT* requires the number of sample points to be a power of 2.

This method should be used with default arguments only for short sequences as the complexity of expressions increases with the size of the sequence.

Parameters

seq : iterable

The sequence on which **DFT** is to be applied.

dps : Integer

Specifies the number of decimal digits for precision.

Examples

```
>>> from sympy import fft, ifft
```

```
>>> fft([1, 2, 3, 4])
[10, -2 - 2*I, -2, -2 + 2*I]
>>> ifft(_)
[1, 2, 3, 4]
```

```
>>> ifft([1, 2, 3, 4])
[5/2, -1/2 + I/2, -1/2, -1/2 - I/2]
>>> fft(_)
[1, 2, 3, 4]
```

```
>>> ifft([1, 7, 3, 4], dps=15)
[3.75, -0.5 - 0.75*I, -1.75, -0.5 + 0.75*I]
>>> fft(_)
[1.0, 7.0, 3.0, 4.0]
```

References

[R164], [R165]

sympy.discrete.transforms.**ifft**(*seq*, *dps=None*)

Performs the Discrete Fourier Transform (**DFT**) in the complex domain.

The sequence is automatically padded to the right with zeros, as the *radix-2 FFT* requires the number of sample points to be a power of 2.

This method should be used with default arguments only for short sequences as the complexity of expressions increases with the size of the sequence.

Parameters

seq : iterable

The sequence on which **DFT** is to be applied.

dps : Integer

Specifies the number of decimal digits for precision.

Examples

```
>>> from sympy import fft, ifft
```

```
>>> fft([1, 2, 3, 4])
[10, -2 - 2*I, -2, -2 + 2*I]
>>> ifft(_)
[1, 2, 3, 4]
```

```
>>> ifft([1, 2, 3, 4])
[5/2, -1/2 + I/2, -1/2, -1/2 - I/2]
>>> fft(_)
[1, 2, 3, 4]
```

```
>>> ifft([1, 7, 3, 4], dps=15)
[3.75, -0.5 - 0.75*I, -1.75, -0.5 + 0.75*I]
>>> fft(_)
[1.0, 7.0, 3.0, 4.0]
```

References

[R166], [R167]

Number Theoretic Transform

sympy.discrete.transforms.**ntt**(*seq*, *prime*)

Performs the Number Theoretic Transform (**NTT**), which specializes the Discrete Fourier Transform (**DFT**) over quotient ring Z/pZ for prime p instead of complex numbers C.

The sequence is automatically padded to the right with zeros, as the *radix-2 NTT* requires the number of sample points to be a power of 2.

Parameters

seq : iterable

The sequence on which **DFT** is to be applied.

prime : Integer

Prime modulus of the form $(m2^k + 1)$ to be used for performing **NTT** on the sequence.

Examples

```
>>> from sympy import ntt, intt
>>> ntt([1, 2, 3, 4], prime=3*2**8 + 1)
[10, 643, 767, 122]
>>> intt(_, 3*2**8 + 1)
[1, 2, 3, 4]
>>> intt([1, 2, 3, 4], prime=3*2**8 + 1)
[387, 415, 384, 353]
>>> ntt(_, prime=3*2**8 + 1)
[1, 2, 3, 4]
```

References

[R168], [R169], [R170]

sympy.discrete.transforms.**intt**(*seq*, *prime*)

Performs the Number Theoretic Transform (**NTT**), which specializes the Discrete Fourier Transform (**DFT**) over quotient ring Z/pZ for prime p instead of complex numbers C.

The sequence is automatically padded to the right with zeros, as the *radix-2 NTT* requires the number of sample points to be a power of 2.

Parameters

seq : iterable

The sequence on which **DFT** is to be applied.

prime : Integer

Prime modulus of the form $(m2^k + 1)$ to be used for performing **NTT** on the sequence.

Examples

```
>>> from sympy import ntt, intt
>>> ntt([1, 2, 3, 4], prime=3*2**8 + 1)
[10, 643, 767, 122]
>>> intt(_, 3*2**8 + 1)
[1, 2, 3, 4]
>>> intt([1, 2, 3, 4], prime=3*2**8 + 1)
[387, 415, 384, 353]
>>> ntt(_, prime=3*2**8 + 1)
[1, 2, 3, 4]
```

References

[R171], [R172], [R173]

Fast Walsh Hadamard Transform

sympy.discrete.transforms.**fwht**(*seq*)

Performs the Walsh Hadamard Transform (**WHT**), and uses Hadamard ordering for the sequence.

The sequence is automatically padded to the right with zeros, as the *radix-2 FWHT* requires the number of sample points to be a power of 2.

Parameters

seq : iterable

The sequence on which WHT is to be applied.

Examples

```
>>> from sympy import fwht, ifwht
>>> fwht([4, 2, 2, 0, 0, 2, -2, 0])
[8, 0, 8, 0, 8, 8, 0, 0]
>>> ifwht(_)
[4, 2, 2, 0, 0, 2, -2, 0]
```

```
>>> ifwht([19, -1, 11, -9, -7, 13, -15, 5])
[2, 0, 4, 0, 3, 10, 0, 0]
>>> fwht(_)
[19, -1, 11, -9, -7, 13, -15, 5]
```

References

[R174], [R175]

`sympy.discrete.transforms.`**`ifwht`**`(`*`seq`*`)`

Performs the Walsh Hadamard Transform (**WHT**), and uses Hadamard ordering for the sequence.

The sequence is automatically padded to the right with zeros, as the *radix-2 FWHT* requires the number of sample points to be a power of 2.

Parameters

seq : iterable

The sequence on which WHT is to be applied.

Examples

```
>>> from sympy import fwht, ifwht
>>> fwht([4, 2, 2, 0, 0, 2, -2, 0])
[8, 0, 8, 0, 8, 8, 0, 0]
>>> ifwht(_)
[4, 2, 2, 0, 0, 2, -2, 0]
```

```
>>> ifwht([19, -1, 11, -9, -7, 13, -15, 5])
[2, 0, 4, 0, 3, 10, 0, 0]
>>> fwht(_)
[19, -1, 11, -9, -7, 13, -15, 5]
```

References

[R176], [R177]

Möbius Transform

`sympy.discrete.transforms.`**`mobius_transform`**`(`*`seq, subset=True`*`)`

Performs the Mobius Transform for subset lattice with indices of sequence as bitmasks.

The indices of each argument, considered as bit strings, correspond to subsets of a finite set.

The sequence is automatically padded to the right with zeros, as the definition of subset/superset based on bitmasks (indices) requires the size of sequence to be a power of 2.

Parameters

seq : iterable

The sequence on which Mobius Transform is to be applied.

subset : bool

Specifies if Mobius Transform is applied by enumerating subsets or supersets of the given set.

Examples

```
>>> from sympy import symbols
>>> from sympy import mobius_transform, inverse_mobius_transform
>>> x, y, z = symbols('x y z')
```

```
>>> mobius_transform([x, y, z])
[x, x + y, x + z, x + y + z]
>>> inverse_mobius_transform(_)
[x, y, z, 0]
```

```
>>> mobius_transform([x, y, z], subset=False)
[x + y + z, y, z, 0]
>>> inverse_mobius_transform(_, subset=False)
[x, y, z, 0]
```

```
>>> mobius_transform([1, 2, 3, 4])
[1, 3, 4, 10]
>>> inverse_mobius_transform(_)
[1, 2, 3, 4]
>>> mobius_transform([1, 2, 3, 4], subset=False)
[10, 6, 7, 4]
>>> inverse_mobius_transform(_, subset=False)
[1, 2, 3, 4]
```

References

[R178], [R179], [R180]

sympy.discrete.transforms.**inverse_mobius_transform**(*seq*, *subset=True*)

Performs the Mobius Transform for subset lattice with indices of sequence as bitmasks.

The indices of each argument, considered as bit strings, correspond to subsets of a finite set.

The sequence is automatically padded to the right with zeros, as the definition of subset/superset based on bitmasks (indices) requires the size of sequence to be a power of 2.

Parameters

seq : iterable

The sequence on which Mobius Transform is to be applied.

subset : bool

Specifies if Mobius Transform is applied by enumerating subsets or supersets of the given set.

Examples

```
>>> from sympy import symbols
>>> from sympy import mobius_transform, inverse_mobius_transform
>>> x, y, z = symbols('x y z')
```

```
>>> mobius_transform([x, y, z])
[x, x + y, x + z, x + y + z]
>>> inverse_mobius_transform(_)
[x, y, z, 0]
```

```
>>> mobius_transform([x, y, z], subset=False)
[x + y + z, y, z, 0]
>>> inverse_mobius_transform(_, subset=False)
[x, y, z, 0]
```

```
>>> mobius_transform([1, 2, 3, 4])
[1, 3, 4, 10]
>>> inverse_mobius_transform(_)
[1, 2, 3, 4]
>>> mobius_transform([1, 2, 3, 4], subset=False)
[10, 6, 7, 4]
>>> inverse_mobius_transform(_, subset=False)
[1, 2, 3, 4]
```

References

[R181], [R182], [R183]

Convolutions

This section lists the methods which implement the basic convolutions for discrete sequences.

Convolution

This is a general method for calculating the convolution of discrete sequences, which internally calls one of the methods `convolution_fft`, `convolution_ntt`, `convolution_fwht`, or `convolution_subset`.

`sympy.discrete.convolutions.`**`convolution`**(*a*, *b*, *cycle=0*, *dps=None*, *prime=None*, *dyadic=None*, *subset=None*)

Performs convolution by determining the type of desired convolution using hints.

Exactly one of `dps`, `prime`, `dyadic`, `subset` arguments should be specified explicitly for identifying the type of convolution, and the argument `cycle` can be specified optionally.

For the default arguments, linear convolution is performed using **FFT**.

Parameters

a, b : iterables

The sequences for which convolution is performed.

cycle : Integer

Specifies the length for doing cyclic convolution.

dps : Integer

Specifies the number of decimal digits for precision for performing **FFT** on the sequence.

prime : Integer

Prime modulus of the form $(m2^k + 1)$ to be used for performing **NTT** on the sequence.

dyadic : bool

Identifies the convolution type as dyadic (*bitwise-XOR*) convolution, which is performed using **FWHT**.

subset : bool

Identifies the convolution type as subset convolution.

Examples

```
>>> from sympy import convolution, symbols, S, I
>>> u, v, w, x, y, z = symbols('u v w x y z')
```

```
>>> convolution([1 + 2*I, 4 + 3*I], [S(5)/4, 6], dps=3)
[1.25 + 2.5*I, 11.0 + 15.8*I, 24.0 + 18.0*I]
>>> convolution([1, 2, 3], [4, 5, 6], cycle=3)
[31, 31, 28]
```

```
>>> convolution([111, 777], [888, 444], prime=19*2**10 + 1)
[1283, 19351, 14219]
>>> convolution([111, 777], [888, 444], prime=19*2**10 + 1, cycle=2)
[15502, 19351]
```

```
>>> convolution([u, v], [x, y, z], dyadic=True)
[u*x + v*y, u*y + v*x, u*z, v*z]
>>> convolution([u, v], [x, y, z], dyadic=True, cycle=2)
[u*x + u*z + v*y, u*y + v*x + v*z]
```

```
>>> convolution([u, v, w], [x, y, z], subset=True)
[u*x, u*y + v*x, u*z + w*x, v*z + w*y]
>>> convolution([u, v, w], [x, y, z], subset=True, cycle=3)
[u*x + v*z + w*y, u*y + v*x, u*z + w*x]
```

Convolution using Fast Fourier Transform

sympy.discrete.convolutions.**convolution_fft**(*a*, *b*, *dps=None*)

Performs linear convolution using Fast Fourier Transform.

Parameters

a, b : iterables

The sequences for which convolution is performed.

dps : Integer

Specifies the number of decimal digits for precision.

Examples

```
>>> from sympy import S, I
>>> from sympy.discrete.convolutions import convolution_fft
```

```
>>> convolution_fft([2, 3], [4, 5])
[8, 22, 15]
>>> convolution_fft([2, 5], [6, 7, 3])
[12, 44, 41, 15]
>>> convolution_fft([1 + 2*I, 4 + 3*I], [S(5)/4, 6])
[5/4 + 5*I/2, 11 + 63*I/4, 24 + 18*I]
```

References

[R184], [R185]

Convolution using Number Theoretic Transform

sympy.discrete.convolutions.**convolution_ntt**(*a*, *b*, *prime*)

Performs linear convolution using Number Theoretic Transform.

Parameters

a, b : iterables

The sequences for which convolution is performed.

prime : Integer

Prime modulus of the form $(m2^k + 1)$ to be used for performing **NTT** on the sequence.

Examples

```
>>> from sympy.discrete.convolutions import convolution_ntt
>>> convolution_ntt([2, 3], [4, 5], prime=19*2**10 + 1)
[8, 22, 15]
>>> convolution_ntt([2, 5], [6, 7, 3], prime=19*2**10 + 1)
[12, 44, 41, 15]
>>> convolution_ntt([333, 555], [222, 666], prime=19*2**10 + 1)
[15555, 14219, 19404]
```

References

[R186], [R187]

Convolution using Fast Walsh Hadamard Transform

`sympy.discrete.convolutions.`**`convolution_fwht`**`(`*a*, *b*`)`

Performs dyadic (*bitwise-XOR*) convolution using Fast Walsh Hadamard Transform.

The convolution is automatically padded to the right with zeros, as the *radix-2 FWHT* requires the number of sample points to be a power of 2.

Parameters

a, b : iterables

The sequences for which convolution is performed.

Examples

```
>>> from sympy import symbols, S, I
>>> from sympy.discrete.convolutions import convolution_fwht
```

```
>>> u, v, x, y = symbols('u v x y')
>>> convolution_fwht([u, v], [x, y])
[u*x + v*y, u*y + v*x]
```

```
>>> convolution_fwht([2, 3], [4, 5])
[23, 22]
>>> convolution_fwht([2, 5 + 4*I, 7], [6*I, 7, 3 + 4*I])
[56 + 68*I, -10 + 30*I, 6 + 50*I, 48 + 32*I]
```

```
>>> convolution_fwht([S(33)/7, S(55)/6, S(7)/4], [S(2)/3, 5])
[2057/42, 1870/63, 7/6, 35/4]
```

References

[R188], [R189]

Subset Convolution

`sympy.discrete.convolutions.`**`convolution_subset`**`(a, b)`

Performs Subset Convolution of given sequences.

The indices of each argument, considered as bit strings, correspond to subsets of a finite set.

The sequence is automatically padded to the right with zeros, as the definition of subset based on bitmasks (indices) requires the size of sequence to be a power of 2.

Parameters

a, b : iterables

The sequences for which convolution is performed.

Examples

```
>>> from sympy import symbols, S
>>> from sympy.discrete.convolutions import convolution_subset
>>> u, v, x, y, z = symbols('u v x y z')
```

```
>>> convolution_subset([u, v], [x, y])
[u*x, u*y + v*x]
>>> convolution_subset([u, v, x], [y, z])
[u*y, u*z + v*y, x*y, x*z]
```

```
>>> convolution_subset([1, S(2)/3], [3, 4])
[3, 6]
>>> convolution_subset([1, 3, S(5)/7], [7])
[7, 21, 5, 0]
```

References

[R190]

Covering Product

`sympy.discrete.convolutions.`**`covering_product`**`(a, b)`

Returns the covering product of given sequences.

The indices of each argument, considered as bit strings, correspond to subsets of a finite set.

The covering product of given sequences is a sequence which contains the sum of products of the elements of the given sequences grouped by the *bitwise-OR* of the corresponding indices.

The sequence is automatically padded to the right with zeros, as the definition of subset based on bitmasks (indices) requires the size of sequence to be a power of 2.

Parameters

a, b : iterables

The sequences for which covering product is to be obtained.

Examples

```
>>> from sympy import symbols, S, I, covering_product
>>> u, v, x, y, z = symbols('u v x y z')
```

```
>>> covering_product([u, v], [x, y])
[u*x, u*y + v*x + v*y]
>>> covering_product([u, v, x], [y, z])
[u*y, u*z + v*y + v*z, x*y, x*z]
```

```
>>> covering_product([1, S(2)/3], [3, 4 + 5*I])
[3, 26/3 + 25*I/3]
>>> covering_product([1, 3, S(5)/7], [7, 8])
[7, 53, 5, 40/7]
```

References

[R191]

Intersecting Product

sympy.discrete.convolutions.**intersecting_product**(*a*, *b*)

Returns the intersecting product of given sequences.

The indices of each argument, considered as bit strings, correspond to subsets of a finite set.

The intersecting product of given sequences is the sequence which contains the sum of products of the elements of the given sequences grouped by the *bitwise-AND* of the corresponding indices.

The sequence is automatically padded to the right with zeros, as the definition of subset based on bitmasks (indices) requires the size of sequence to be a power of 2.

Parameters

a, b : iterables

The sequences for which intersecting product is to be obtained.

Examples

```
>>> from sympy import symbols, S, I, intersecting_product
>>> u, v, x, y, z = symbols('u v x y z')
```

```
>>> intersecting_product([u, v], [x, y])
[u*x + u*y + v*x, v*y]
>>> intersecting_product([u, v, x], [y, z])
[u*y + u*z + v*y + x*y + x*z, v*z, 0, 0]
```

```
>>> intersecting_product([1, S(2)/3], [3, 4 + 5*I])
[9 + 5*I, 8/3 + 10*I/3]
>>> intersecting_product([1, 3, S(5)/7], [7, 8])
[327/7, 24, 0, 0]
```

References

[R192]

Numerical Evaluation

Basics

Exact SymPy expressions can be converted to floating-point approximations (decimal numbers) using either the `.evalf()` method or the `N()` function. `N(expr, <args>)` is equivalent to `sympify(expr).evalf(<args>)`.

```
>>> from sympy import *
>>> N(sqrt(2)*pi)
4.44288293815837
>>> (sqrt(2)*pi).evalf()
4.44288293815837
```

By default, numerical evaluation is performed to an accuracy of 15 decimal digits. You can optionally pass a desired accuracy (which should be a positive integer) as an argument to `evalf` or `N`:

```
>>> N(sqrt(2)*pi, 5)
4.4429
>>> N(sqrt(2)*pi, 50)
4.4428829381583662470158809900606936986146216893757
```

Complex numbers are supported:

```
>>> N(1/(pi + I), 20)
0.28902548222223624241 - 0.091999668350375232456*I
```

If the expression contains symbols or for some other reason cannot be evaluated numerically, calling `.evalf()` or `N()` returns the original expression, or in some cases a partially evaluated expression. For example, when the expression is a polynomial in expanded form, the coefficients are evaluated:

```
>>> x = Symbol('x')
>>> (pi*x**2 + x/3).evalf()
3.14159265358979*x**2 + 0.333333333333333*x
```

You can also use the standard Python functions `float()`, `complex()` to convert SymPy expressions to regular Python numbers:

```
>>> float(pi)
3.1415926535...
>>> complex(pi+E*I)
(3.1415926535...+2.7182818284...j)
```

If these functions are used, failure to evaluate the expression to an explicit number (for example if the expression contains symbols) will raise an exception.

There is essentially no upper precision limit. The following command, for example, computes the first 100,000 digits of π/e:

```
>>> N(pi/E, 100000)
...
```

This shows digits 999,951 through 1,000,000 of pi:

```
>>> str(N(pi, 10**6))[-50:]
'95678796130331164628399634646042209010610577945815'
```

High-precision calculations can be slow. It is recommended (but entirely optional) to install gmpy (https://code.google.com/p/gmpy/), which will significantly speed up computations such as the one above.

Floating-point numbers

Floating-point numbers in SymPy are instances of the class `Float`. A `Float` can be created with a custom precision as second argument:

```
>>> Float(0.1)
0.100000000000000
>>> Float(0.1, 10)
0.1000000000
>>> Float(0.125, 30)
0.125000000000000000000000000000
>>> Float(0.1, 30)
0.100000000000000005551115123126
```

As the last example shows, some Python floats are only accurate to about 15 digits as inputs, while others (those that have a denominator that is a power of 2, like 0.125 = 1/8) are exact. To create a `Float` from a high-precision decimal number, it is better to pass a string, `Rational`, or `evalf` a `Rational`:

```
>>> Float('0.1', 30)
0.100000000000000000000000000000
>>> Float(Rational(1, 10), 30)
0.100000000000000000000000000000
```

(continues on next page)

(continued from previous page)

```
>>> Rational(1, 10).evalf(30)
0.100000000000000000000000000000
```

The precision of a number determines 1) the precision to use when performing arithmetic with the number, and 2) the number of digits to display when printing the number. When two numbers with different precision are used together in an arithmetic operation, the higher of the precisions is used for the result. The product of 0.1 +/- 0.001 and 3.1415 +/- 0.0001 has an uncertainty of about 0.003 and yet 5 digits of precision are shown.

```
>>> Float(0.1, 3)*Float(3.1415, 5)
0.31417
```

So the displayed precision should not be used as a model of error propagation or significance arithmetic; rather, this scheme is employed to ensure stability of numerical algorithms.

`N` and `evalf` can be used to change the precision of existing floating-point numbers:

```
>>> N(3.5)
3.50000000000000
>>> N(3.5, 5)
3.5000
>>> N(3.5, 30)
3.50000000000000000000000000000
```

Accuracy and error handling

When the input to `N` or `evalf` is a complicated expression, numerical error propagation becomes a concern. As an example, consider the 100'th Fibonacci number and the excellent (but not exact) approximation $\varphi^{100}/\sqrt{5}$ where φ is the golden ratio. With ordinary floating-point arithmetic, subtracting these numbers from each other erroneously results in a complete cancellation:

```
>>> a, b = GoldenRatio**1000/sqrt(5), fibonacci(1000)
>>> float(a)
4.34665576869e+208
>>> float(b)
4.34665576869e+208
>>> float(a) - float(b)
0.0
```

`N` and `evalf` keep track of errors and automatically increase the precision used internally in order to obtain a correct result:

```
>>> N(fibonacci(100) - GoldenRatio**100/sqrt(5))
-5.64613129282185e-22
```

Unfortunately, numerical evaluation cannot tell an expression that is exactly zero apart from one that is merely very small. The working precision is therefore capped, by default to around 100 digits. If we try with the 1000'th Fibonacci number, the following happens:

```
>>> N(fibonacci(1000) - (GoldenRatio)**1000/sqrt(5))
0.e+85
```

The lack of digits in the returned number indicates that N failed to achieve full accuracy. The result indicates that the magnitude of the expression is something less than 10^84, but that is not a particularly good answer. To force a higher working precision, the maxn keyword argument can be used:

```
>>> N(fibonacci(1000) - (GoldenRatio)**1000/sqrt(5), maxn=500)
-4.60123853010113e-210
```

Normally, maxn can be set very high (thousands of digits), but be aware that this may cause significant slowdown in extreme cases. Alternatively, the strict=True option can be set to force an exception instead of silently returning a value with less than the requested accuracy:

```
>>> N(fibonacci(1000) - (GoldenRatio)**1000/sqrt(5), strict=True)
Traceback (most recent call last):
...
PrecisionExhausted: Failed to distinguish the expression:

-sqrt(5)*GoldenRatio**1000/5 +␣
↪43466557686937456435688527675040625802564660517371780402481729089536555417949051890403879

from zero. Try simplifying the input, using chop=True, or providing a higher␣
↪maxn for evalf
```

If we add a term so that the Fibonacci approximation becomes exact (the full form of Binet's formula), we get an expression that is exactly zero, but N does not know this:

```
>>> f = fibonacci(100) - (GoldenRatio**100 - (GoldenRatio-1)**100)/sqrt(5)
>>> N(f)
0.e-104
>>> N(f, maxn=1000)
0.e-1336
```

In situations where such cancellations are known to occur, the chop options is useful. This basically replaces very small numbers in the real or imaginary portions of a number with exact zeros:

```
>>> N(f, chop=True)
0
>>> N(3 + I*f, chop=True)
3.00000000000000
```

In situations where you wish to remove meaningless digits, re-evaluation or the use of the round method are useful:

```
>>> Float('.1', '')*Float('.12345', '')
0.012297
>>> ans = _
>>> N(ans, 1)
0.01
>>> ans.round(2)
0.01
```

If you are dealing with a numeric expression that contains no floats, it can be evaluated to arbitrary precision. To round the result relative to a given decimal, the round method is useful:

```
>>> v = 10*pi + cos(1)
>>> N(v)
31.9562288417661
>>> v.round(3)
31.956
```

Sums and integrals

Sums (in particular, infinite series) and integrals can be used like regular closed-form expressions, and support arbitrary-precision evaluation:

```
>>> var('n x')
(n, x)
>>> Sum(1/n**n, (n, 1, oo)).evalf()
1.29128599706266
>>> Integral(x**(-x), (x, 0, 1)).evalf()
1.29128599706266
>>> Sum(1/n**n, (n, 1, oo)).evalf(50)
1.2912859970626635404072825905956005414986193682745
>>> Integral(x**(-x), (x, 0, 1)).evalf(50)
1.2912859970626635404072825905956005414986193682745
>>> (Integral(exp(-x**2), (x, -oo, oo)) ** 2).evalf(30)
3.14159265358979323846264338328
```

By default, the tanh-sinh quadrature algorithm is used to evaluate integrals. This algorithm is very efficient and robust for smooth integrands (and even integrals with endpoint singularities), but may struggle with integrals that are highly oscillatory or have mid-interval discontinuities. In many cases, `evalf/N` will correctly estimate the error. With the following integral, the result is accurate but only good to four digits:

```
>>> f = abs(sin(x))
>>> Integral(abs(sin(x)), (x, 0, 4)).evalf()
2.346
```

It is better to split this integral into two pieces:

```
>>> (Integral(f, (x, 0, pi)) + Integral(f, (x, pi, 4))).evalf()
2.34635637913639
```

A similar example is the following oscillatory integral:

```
>>> Integral(sin(x)/x**2, (x, 1, oo)).evalf(maxn=20)
0.5
```

It can be dealt with much more efficiently by telling `evalf` or `N` to use an oscillatory quadrature algorithm:

```
>>> Integral(sin(x)/x**2, (x, 1, oo)).evalf(quad='osc')
0.504067061906928
>>> Integral(sin(x)/x**2, (x, 1, oo)).evalf(20, quad='osc')
0.50406706190692837199
```

Oscillatory quadrature requires an integrand containing a factor cos(ax+b) or sin(ax+b). Note that many other oscillatory integrals can be transformed to this form with a change of variables:

```
>>> init_printing(use_unicode=False, wrap_line=False)
>>> intgrl = Integral(sin(1/x), (x, 0, 1)).transform(x, 1/x)
>>> intgrl
 oo
  /
 |
 |  sin(x)
 |  ------ dx
 |     2
 |    x
 |
/
1
>>> N(intgrl, quad='osc')
0.504067061906928
```

Infinite series use direct summation if the series converges quickly enough. Otherwise, extrapolation methods (generally the Euler-Maclaurin formula but also Richardson extrapolation) are used to speed up convergence. This allows high-precision evaluation of slowly convergent series:

```
>>> var('k')
k
>>> Sum(1/k**2, (k, 1, oo)).evalf()
1.64493406684823
>>> zeta(2).evalf()
1.64493406684823
>>> Sum(1/k-log(1+1/k), (k, 1, oo)).evalf()
0.577215664901533
>>> Sum(1/k-log(1+1/k), (k, 1, oo)).evalf(50)
0.57721566490153286060651209008240243104215933593992
>>> EulerGamma.evalf(50)
0.57721566490153286060651209008240243104215933593992
```

The Euler-Maclaurin formula is also used for finite series, allowing them to be approximated quickly without evaluating all terms:

```
>>> Sum(1/k, (k, 10000000, 20000000)).evalf()
0.693147255559946
```

Note that `evalf` makes some assumptions that are not always optimal. For fine-tuned control over numerical summation, it might be worthwhile to manually use the method `Sum.euler_maclaurin`.

Special optimizations are used for rational hypergeometric series (where the term is a product of polynomials, powers, factorials, binomial coefficients and the like). `N/evalf` sum series of this type very rapidly to high precision. For example, this Ramanujan formula for pi can be summed to 10,000 digits in a fraction of a second with a simple command:

```
>>> f = factorial
>>> n = Symbol('n', integer=True)
```

(continues on next page)

(continued from previous page)

```
>>> R = 9801/sqrt(8)/Sum(f(4*n)*(1103+26390*n)/f(n)**4/396**(4*n),
...                             (n, 0, oo))
>>> N(R, 10000)
3.
↪1415926535897932384626433832795028841971693993751058209749445923078164062862089
9862803482534211706798214808651328230664709384460955058223172535940812848111745
02841027019385211055596446229489549303819644288109756659334461284756482337867831
...
```

Numerical simplification

The function nsimplify attempts to find a formula that is numerically equal to the given input. This feature can be used to guess an exact formula for an approximate floating-point input, or to guess a simpler formula for a complicated symbolic input. The algorithm used by nsimplify is capable of identifying simple fractions, simple algebraic expressions, linear combinations of given constants, and certain elementary functional transformations of any of the preceding.

Optionally, nsimplify can be passed a list of constants to include (e.g. pi) and a minimum numerical tolerance. Here are some elementary examples:

```
>>> nsimplify(0.1)
1/10
>>> nsimplify(6.28, [pi], tolerance=0.01)
2*pi
>>> nsimplify(pi, tolerance=0.01)
22/7
>>> nsimplify(pi, tolerance=0.001)
355
---
113
>>> nsimplify(0.33333, tolerance=1e-4)
1/3
>>> nsimplify(2.0**(1/3.), tolerance=0.001)
635
---
504
>>> nsimplify(2.0**(1/3.), tolerance=0.001, full=True)
3 ___
\/ 2
```

Here are several more advanced examples:

```
>>> nsimplify(Float('0.130198866629986772369127970337',30), [pi, E])
    1
----------
5*pi
---- + 2*e
 7
>>> nsimplify(cos(atan('1/3')))
    ____
```

(continues on next page)

(continued from previous page)

```
3*\/ 10
--------
   10
>>> nsimplify(4/(1+sqrt(5)), [GoldenRatio])
-2 + 2*GoldenRatio
>>> nsimplify(2 + exp(2*atan('1/4')*I))
49   8*I
-- + ---
17    17
>>> nsimplify((1/(exp(3*pi*I/5)+1)))
           ___________
          /   ___
1        /  \/ 5    1
- - I*  /   ----- + -
2     \/      10    4
>>> nsimplify(I**I, [pi])
 -pi
 ----
  2
e
>>> n = Symbol('n')
>>> nsimplify(Sum(1/n**2, (n, 1, oo)), [pi])
  2
pi
---
 6
>>> nsimplify(gamma('1/4')*gamma('3/4'), [pi])
  ___
\/ 2 *pi
```

Numeric Computation

Symbolic computer algebra systems like SymPy facilitate the construction and manipulation of mathematical expressions. Unfortunately when it comes time to evaluate these expressions on numerical data, symbolic systems often have poor performance.

Fortunately SymPy offers a number of easy-to-use hooks into other numeric systems, allowing you to create mathematical expressions in SymPy and then ship them off to the numeric system of your choice. This page documents many of the options available including the `math` library, the popular array computing package `numpy`, code generation in `Fortran` or `C`, and the use of the array compiler `Aesara`.

Subs/evalf

Subs is the slowest but simplest option. It runs at SymPy speeds. The `.subs(...).evalf()` method can substitute a numeric value for a symbolic one and then evaluate the result within SymPy.

```
>>> from sympy import *
>>> from sympy.abc import x
>>> expr = sin(x)/x
>>> expr.evalf(subs={x: 3.14})
0.000507214304613640
```

This method is slow. You should use this method production only if performance is not an issue. You can expect `.subs` to take tens of microseconds. It can be useful while prototyping or if you just want to see a value once.

Lambdify

The `lambdify` function translates SymPy expressions into Python functions, leveraging a variety of numerical libraries. It is used as follows:

```
>>> from sympy import *
>>> from sympy.abc import x
>>> expr = sin(x)/x
>>> f = lambdify(x, expr)
>>> f(3.14)
0.000507214304614
```

Here lambdify makes a function that computes `f(x) = sin(x)/x`. By default lambdify relies on implementations in the `math` standard library. This numerical evaluation takes on the order of hundreds of nanoseconds, roughly two orders of magnitude faster than the `.subs` method. This is the speed difference between SymPy and raw Python.

Lambdify can leverage a variety of numerical backends. By default it uses the `math` library. However it also supports `mpmath` and most notably, `numpy`. Using the `numpy` library gives the generated function access to powerful vectorized ufuncs that are backed by compiled C code.

```
>>> from sympy import *
>>> from sympy.abc import x
>>> expr = sin(x)/x
>>> f = lambdify(x, expr, "numpy")
```

```
>>> import numpy
>>> data = numpy.linspace(1, 10, 10000)
>>> f(data)
[ 0.84147098  0.84119981  0.84092844 ... -0.05426074 -0.05433146
 -0.05440211]
```

If you have array-based data this can confer a considerable speedup, on the order of 10 nanoseconds per element. Unfortunately numpy incurs some start-up time and introduces an overhead of a few microseconds.

CuPy is a NumPy-compatible array library that mainly runs on CUDA, but has increasing support for other GPU manufacturers. It can in many cases be used as a drop-in replacement for numpy.

```
>>> f = lambdify(x, expr, "cupy")
>>> import cupy as cp
>>> data = cp.linspace(1, 10, 10000)
>>> y = f(data) # perform the computation
>>> cp.asnumpy(y) # explicitly copy from GPU to CPU / numpy array
[ 0.84147098  0.84119981  0.84092844 ... -0.05426074 -0.05433146
 -0.05440211]
```

JAX is a similar alternative to CuPy that provides GPU and TPU acceleration via just-in-time compliation to XLA. It too, can in some cases, be used as a drop-in replacement for numpy.

```
>>> f = lambdify(x, expr, "jax")
>>> import jax.numpy as jnp
>>> data = jnp.linspace(1, 10, 10000)
>>> y = f(data) # perform the computation
>>> numpy.asarray(y) # explicitly copy to CPU / numpy array
array([ 0.84147096,  0.8411998 ,  0.84092844, ..., -0.05426079,
   -0.05433151, -0.05440211], dtype=float32)
```

uFuncify

The `autowrap` module contains methods that help in efficient computation.

- *autowrap* (page 1114) method for compiling code generated by the *codegen* (page 1108) module, and wrap the binary for use in python.
- *binary_function* (page 1116) method automates the steps needed to autowrap the SymPy expression and attaching it to a `Function` object with `implemented_function()`.
- *ufuncify* (page 1116) generates a binary function that supports broadcasting on numpy arrays using different backends that are faster as compared to `subs/evalf` and `lambdify`.

The API reference of all the above is listed here: *`sympy.utilities.autowrap()`* (page 2039).

Aesara

SymPy has a strong connection with Aesara, a mathematical array compiler. SymPy expressions can be easily translated to Aesara graphs and then compiled using the Aesara compiler chain.

```
>>> from sympy import *
>>> from sympy.abc import x
>>> expr = sin(x)/x
```

```
>>> from sympy.printing.aesaracode import aesara_function
>>> f = aesara_function([x], [expr])
```

If array broadcasting or types are desired then Aesara requires this extra information

```
>>> f = aesara_function([x], [expr], dims={x: 1}, dtypes={x: 'float64'})
```

Aesara has a more sophisticated code generation system than SymPy's C/Fortran code printers. Among other things it handles common sub-expressions and compilation onto the GPU. Aesara also supports SymPy Matrix and Matrix Expression objects.

So Which Should I Use?

The options here were listed in order from slowest and least dependencies to fastest and most dependencies. For example, if you have Aesara installed then that will often be the best choice. If you don't have Aesara but do have `f2py` then you should use `ufuncify`. If you have been comfortable using lambdify with the numpy module, but have a GPU, CuPy and JAX can provide substantial speedups with little effort.

Tool	Speed	Qualities	Dependencies
subs/evalf	50us	Simple	None
lambdify	1us	Scalar functions	math
lambdify-numpy	10ns	Vector functions	numpy
ufuncify	10ns	Complex vector expressions	f2py, Cython
lambdify-cupy	10ns	Vector functions on GPUs	cupy
lambdify-jax	10ns	Vector functions on CPUs, GPUs and TPUs	jax
Aesara	10ns	Many outputs, CSE, GPUs	Aesara

Term Rewriting

Term rewriting is a very general class of functionalities which are used to convert expressions of one type in terms of expressions of different kind. For example expanding, combining and converting expressions apply to term rewriting, and also simplification routines can be included here. Currently SymPy has several functions and basic built-in methods for performing various types of rewriting.

Expanding

The simplest rewrite rule is expanding expressions into a _sparse_ form. Expanding has several flavors and include expanding complex valued expressions, arithmetic expand of products and powers but also expanding functions in terms of more general functions is possible. Below are listed all currently available expand rules.

Expanding of arithmetic expressions involving products and powers:

```
>>> from sympy import *
>>> x, y, z = symbols('x,y,z')
>>> ((x + y)*(x - y)).expand(basic=True)
x**2 - y**2
>>> ((x + y + z)**2).expand(basic=True)
x**2 + 2*x*y + 2*x*z + y**2 + 2*y*z + z**2
```

Arithmetic expand is done by default in `expand()` so the keyword `basic` can be omitted. However you can set `basic=False` to avoid this type of expand if you use rules described below. This give complete control on what is done with the expression.

Another type of expand rule is expanding complex valued expressions and putting them into a normal form. For this complex keyword is used. Note that it will always perform arithmetic expand to obtain the desired normal form:

```
>>> (x + I*y).expand(complex=True)
re(x) + I*re(y) + I*im(x) - im(y)
```

```
>>> sin(x + I*y).expand(complex=True)
sin(re(x) - im(y))*cosh(re(y) + im(x)) + I*cos(re(x) - im(y))*sinh(re(y) +␣
↪im(x))
```

Note also that the same behavior can be obtained by using as_real_imag() method. However it will return a tuple containing the real part in the first place and the imaginary part in the other. This can be also done in a two step process by using collect function:

```
>>> (x + I*y).as_real_imag()
(re(x) - im(y), re(y) + im(x))
```

```
>>> collect((x + I*y).expand(complex=True), I, evaluate=False)
{1: re(x) - im(y), I: re(y) + im(x)}
```

There is also possibility for expanding expressions in terms of expressions of different kind. This is very general type of expanding and usually you would use rewrite() to do specific type of rewrite:

```
>>> GoldenRatio.expand(func=True)
1/2 + sqrt(5)/2
```

Common Subexpression Detection and Collection

Before evaluating a large expression, it is often useful to identify common subexpressions, collect them and evaluate them at once. This is implemented in the cse function. Examples:

```
>>> from sympy import cse, sqrt, sin, pprint
>>> from sympy.abc import x

>>> pprint(cse(sqrt(sin(x))), use_unicode=True)
⎛    ⎡  ________⎤⎞
⎝[], ⎣╲╱ sin(x) ⎦⎠

>>> pprint(cse(sqrt(sin(x)+5)*sqrt(sin(x)+4)), use_unicode=True)
⎛                ⎡  ________   ________⎤⎞
⎝[(x₀, sin(x))], ⎣╲╱ x₀ + 4 ⋅╲╱ x₀ + 5 ⎦⎠

>>> pprint(cse(sqrt(sin(x+1) + 5 + cos(y))*sqrt(sin(x+1) + 4 + cos(y))),
...     use_unicode=True)
⎛                             ⎡  ________   ________⎤⎞
⎝[(x₀, sin(x + 1) + cos(y))], ⎣╲╱ x₀ + 4 ⋅╲╱ x₀ + 5 ⎦⎠

>>> pprint(cse((x-y)*(z-y) + sqrt((x-y)*(z-y))), use_unicode=True)
⎛                          ⎡  ____     ⎤⎞
⎝[(x₀, (x - y)⋅(-y + z))], ⎣╲╱ x₀  + x₀⎦⎠
```

Optimizations to be performed before and after common subexpressions elimination can be passed in the``optimizations`` optional argument. A set of predefined basic optimizations can be applied by passing `optimizations='basic'`:

```
>>> pprint(cse((x-y)*(z-y) + sqrt((x-y)*(z-y)), optimizations='basic'),
...     use_unicode=True)
⎛                                  ⎡  ____     ⎤⎞
⎝[(x₀, -(x - y)⋅(y - z))], ⎣╲╱ x₀  + x₀⎦⎠
```

However, these optimizations can be very slow for large expressions. Moreover, if speed is a concern, one can pass the option `order='none'`. Order of terms will then be dependent on hashing algorithm implementation, but speed will be greatly improved.

More information:

`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)
```

5.8.2 Code Generation

Contents

Several submodules in SymPy allow one to generate directly compilable and executable code in a variety of different programming languages from SymPy expressions. In addition, there are functions that generate Python importable objects that can evaluate SymPy expressions very efficiently.

We will start with a brief introduction to the components that make up the code generation capabilities of SymPy.

Introduction

There are four main levels of abstractions:

```
expression
    |
code printers
    |
code generators
    |
autowrap
```

sympy.utilities.autowrap (page 2039) uses codegen, and codegen uses the code printers. *sympy.utilities.autowrap* (page 2039) does everything: it lets you go from SymPy expression to numerical function in the same Python process in one step. Codegen is actual code generation, i.e., to compile and use later, or to include in some larger project.

The code printers translate the SymPy objects into actual code, like `abs(x) -> fabs(x)` (for C).

The code printers don't print optimal code in many cases. An example of this is powers in C. `x**2` prints as `pow(x, 2)` instead of `x*x`. Other optimizations (like mathematical simplifications) should happen before the code printers.

Currently, *sympy.simplify.cse_main.cse()* (page 685) is not applied automatically anywhere in this chain. It ideally happens at the codegen level, or somewhere above it.

We will iterate through the levels below.

The following three lines will be used to setup each example:

```
>>> from sympy import *
>>> init_printing(use_unicode=True)
>>> from sympy.abc import a, e, k, n, r, t, x, y, z, T, Z
>>> from sympy.abc import beta, omega, tau
>>> f, g = symbols('f, g', cls=Function)
```

Code printers (sympy.printing)

This is where the meat of code generation is; the translation of SymPy actually more like a lightweight version of codegen for Python, and Python (*sympy.printing.pycode.pycode()* (page 2176)), and *sympy.printing.lambdarepr.lambdarepr()* (page 2170), which supports many libraries (like NumPy), and Aesara (*sympy.printing.aesaracode.aesara_function()* (page 2167)). The code printers are special cases of the other prints in SymPy (str printer, pretty printer, etc.).

An important distinction is that the code printer has to deal with assignments (using the *sympy.codegen.ast.Assignment* (page 1126) object). This serves as building blocks for the code printers and hence the `codegen` module. An example that shows the use of `Assignment` in C code:

```
>>> from sympy.codegen.ast import Assignment
>>> print(ccode(Assignment(x, y + 1)))
x = y + 1;
```

Here is another simple example of printing a C version of a SymPy expression:

```
>>> expr = (Rational(-1, 2) * Z * k * (e**2) / r)
>>> expr
    2
-Z⋅e ⋅k
────────
  2⋅r
>>> ccode(expr)
-1.0/2.0*Z*pow(e, 2)*k/r
>>> from sympy.codegen.ast import real, float80
>>> ccode(expr, assign_to="E", type_aliases={real: float80})
E = -1.0L/2.0L*Z*powl(e, 2)*k/r;
```

To generate code with some math functions provided by e.g. the C99 standard we need to import functions from *sympy.codegen.cfunctions* (page 1143):

```
>>> from sympy.codegen.cfunctions import expm1
>>> ccode(expm1(x), standard='C99')
expm1(x)
```

`Piecewise` expressions are converted into conditionals. If an `assign_to` variable is provided an if statement is created, otherwise the ternary operator is used. Note that if the `Piecewise` lacks a default term, represented by `(expr, True)` then an error will be thrown. This is to prevent generating an expression that may not evaluate to anything. A use case for `Piecewise`:

```
>>> expr = Piecewise((x + 1, x > 0), (x, True))
>>> print(fcode(expr, tau))
      if (x > 0) then
         tau = x + 1
      else
         tau = x
      end if
```

The various printers also tend to support `Indexed` objects well. With `contract=True` these expressions will be turned into loops, whereas `contract=False` will just print the assignment expression that should be looped over:

```
>>> len_y = 5
>>> mat_1 = IndexedBase('mat_1', shape=(len_y,))
>>> mat_2 = IndexedBase('mat_2', shape=(len_y,))
>>> Dy = IndexedBase('Dy', shape=(len_y-1,))
>>> i = Idx('i', len_y-1)
>>> eq = Eq(Dy[i], (mat_1[i+1] - mat_1[i]) / (mat_2[i+1] - mat_2[i]))
>>> print(jscode(eq.rhs, assign_to=eq.lhs, contract=False))
Dy[i] = (mat_1[i + 1] - mat_1[i])/(mat_2[i + 1] - mat_2[i]);
>>> Res = IndexedBase('Res', shape=(len_y,))
>>> j = Idx('j', len_y)
>>> eq = Eq(Res[j], mat_1[j]*mat_2[j])
>>> print(jscode(eq.rhs, assign_to=eq.lhs, contract=True))
for (var j=0; j<5; j++){
   Res[j] = 0;
}
for (var j=0; j<5; j++){
   for (var j=0; j<5; j++){
```

(continues on next page)

(continued from previous page)

```
      Res[j] = Res[j] + mat_1[j]*mat_2[j];
   }
}
>>> print(jscode(eq.rhs, assign_to=eq.lhs, contract=False))
Res[j] = mat_1[j]*mat_2[j];
```

Custom printing can be defined for certain types by passing a dictionary of "type" : "function" to the user_functions kwarg. Alternatively, the dictionary value can be a list of tuples i.e., [(argument_test, cfunction_string)]. This can be used to call a custom Octave function:

```
>>> custom_functions = {
...   "f": "existing_octave_fcn",
...   "g": [(lambda x: x.is_Matrix, "my_mat_fcn"),
...         (lambda x: not x.is_Matrix, "my_fcn")]
... }
>>> mat = Matrix([[1, x]])
>>> octave_code(f(x) + g(x) + g(mat), user_functions=custom_functions)
existing_octave_fcn(x) + my_fcn(x) + my_mat_fcn([1 x])
```

An example of Mathematica code printer:

```
>>> x_ = Function('x')
>>> expr = x_(n*T) * sin((t - n*T) / T)
>>> expr = expr / ((-T*n + t) / T)
>>> expr
            ⎛-T⋅n + t⎞
T⋅x(T⋅n)⋅sin⎜────────⎟
            ⎝   T    ⎠
──────────────────────
       -T⋅n + t

>>> expr = summation(expr, (n, -1, 1))
>>> mathematica_code(expr)
T*(x[-T]*Sin[(T + t)/T]/(T + t) + x[T]*Sin[(-T + t)/T]/(-T + t) + x[0]*Sin[t/T
]/t)
```

We can go through a common expression in different languages we support and see how it works:

```
>>> k, g1, g2, r, I, S = symbols("k, gamma_1, gamma_2, r, I, S")
>>> expr = k * g1 * g2 / (r**3)
>>> expr = expr * 2 * I * S * (3 * (cos(beta))**2 - 1) / 2
>>> expr
            ⎛     2       ⎞
I⋅S⋅γ₁⋅γ₂⋅k⋅⎝3⋅cos (β) - 1⎠
───────────────────────────
              3
             r
>>> print(jscode(expr, assign_to="H_is"))
H_is = I*S*gamma_1*gamma_2*k*(3*Math.pow(Math.cos(beta), 2) - 1)/Math.pow(r,␣
↪3);
>>> print(ccode(expr, assign_to="H_is", standard='C89'))
```

(continues on next page)

(continued from previous page)

```
H_is = I*S*gamma_1*gamma_2*k*(3*pow(cos(beta), 2) - 1)/pow(r, 3);
>>> print(fcode(expr, assign_to="H_is"))
      H_is = I*S*gamma_1*gamma_2*k*(3*cos(beta)**2 - 1)/r**3
>>> print(julia_code(expr, assign_to="H_is"))
H_is = I .* S .* gamma_1 .* gamma_2 .* k .* (3 * cos(beta) .^ 2 - 1) ./ r .^ 3
>>> print(octave_code(expr, assign_to="H_is"))
H_is = I.*S.*gamma_1.*gamma_2.*k.*(3*cos(beta).^2 - 1)./r.^3;
>>> print(rust_code(expr, assign_to="H_is"))
H_is = I*S*gamma_1*gamma_2*k*(3*beta.cos().powi(2) - 1)/r.powi(3);
>>> print(mathematica_code(expr))
I*S*gamma_1*gamma_2*k*(3*Cos[beta]^2 - 1)/r^3
```

Codegen (sympy.utilities.codegen)

This module deals with creating compilable code from SymPy expressions. This is lower level than autowrap, as it doesn't actually attempt to compile the code, but higher level than the printers, as it generates compilable files (including header files), rather than just code snippets.

The user friendly functions, here, are `codegen` and `make_routine`. `codegen` takes a list of `(variable, expression)` pairs and a language (C, F95, and Octave/Matlab are supported). It returns, as strings, a code file and a header file (for relevant languages). The variables are created as functions that return the value of the expression as output.

Note: The codegen callable is not in the sympy namespace automatically, to use it you must first import codegen from `sympy.utilities.codegen`

For instance:

```
>>> from sympy.utilities.codegen import codegen
>>> length, breadth, height = symbols('length, breadth, height')
>>> [(c_name, c_code), (h_name, c_header)] = \
... codegen(('volume', length*breadth*height), "C99", "test",
...         header=False, empty=False)
>>> print(c_name)
test.c
>>> print(c_code)
#include "test.h"
#include <math.h>
double volume(double breadth, double height, double length) {
   double volume_result;
   volume_result = breadth*height*length;
   return volume_result;
}
>>> print(h_name)
test.h
>>> print(c_header)
#ifndef PROJECT__TEST__H
#define PROJECT__TEST__H
```

(continues on next page)

(continued from previous page)

```
double volume(double breadth, double height, double length);
#endif
```

Various flags to codegen let you modify things. The project name for preprocessor instructions can be varied using project. Variables listed as global variables in arg global_vars will not show up as function arguments.

language is a case-insensitive string that indicates the source code language. Currently, C, F95 and Octave are supported. Octave generates code compatible with both Octave and Matlab.

header when True, a header is written on top of each source file. empty when True, empty lines are used to structure the code. With argument_sequence a sequence of arguments for the routine can be defined in a preferred order.

prefix defines a prefix for the names of the files that contain the source code. If omitted, the name of the first name_expr tuple is used.

to_files when True, the code will be written to one or more files with the given prefix.

Here is an example:

```
>>> [(f_name, f_code), header] = codegen(("volume", length*breadth*height),
...     "F95", header=False, empty=False, argument_sequence=(breadth, length),
...     global_vars=(height,))
>>> print(f_code)
REAL*8 function volume(breadth, length)
implicit none
REAL*8, intent(in) :: breadth
REAL*8, intent(in) :: length
volume = breadth*height*length
end function
```

The method make_routine creates a Routine object, which represents an evaluation routine for a set of expressions. This is only good for internal use by the CodeGen objects, as an intermediate representation from SymPy expression to generated code. It is not recommended to make a Routine object yourself. You should instead use make_routine method. make_routine in turn calls the routine method of the CodeGen object depending upon the language of choice. This creates the internal objects representing assignments and so on, and creates the Routine class with them.

The various codegen objects such as Routine and Variable aren't SymPy objects (they don't subclass from Basic).

For example:

```
>>> from sympy.utilities.codegen import make_routine
>>> from sympy.physics.hydrogen import R_nl
>>> expr = R_nl(3, y, x, 6)
>>> routine = make_routine('my_routine', expr)
>>> [arg.result_var for arg in routine.results]
[result₅₁₄₂₃₄₁₆₈₁₃₉₇₇₁₉₄₂₈]
>>> [arg.expr for arg in routine.results]
⎡                 __________                                                    ⎤
⎢          y     ╱ (2 - y)!   -2⋅x                                              ⎥
```

(continues on next page)

(continued from previous page)

```
│4⋅√6⋅(4⋅x)  ⋅  ╱   ──────── ⋅e     ⋅assoc_laguerre(2 - y, 2⋅y + 1, 4⋅x)│
│             ╲╱   (y + 3)!                                            │
│──────────────────────────────────────────────────────────────────────│
⎣                                3                                     ⎦
>>> [arg.name for arg in routine.arguments]
[x, y]
```

Another more complicated example with a mixture of specified and automatically-assigned names. Also has Matrix output:

```
>>> routine = make_routine('fcn', [x*y, Eq(a, 1), Eq(r, x + r), Matrix([[x,␣
↪2]])])
>>> [arg.result_var for arg in routine.results]
[result_5397460570204848505]
>>> [arg.expr for arg in routine.results]
[x⋅y]
>>> [arg.name for arg in routine.arguments]
[x, y, a, r, out_8598435338387848786]
```

We can examine the various arguments more closely:

```
>>> from sympy.utilities.codegen import (InputArgument, OutputArgument,
...                                      InOutArgument)
>>> [a.name for a in routine.arguments if isinstance(a, InputArgument)]
[x, y]

>>> [a.name for a in routine.arguments if isinstance(a, OutputArgument)]
[a, out_8598435338387848786]
>>> [a.expr for a in routine.arguments if isinstance(a, OutputArgument)]
[1, [x  2]]

>>> [a.name for a in routine.arguments if isinstance(a, InOutArgument)]
[r]
>>> [a.expr for a in routine.arguments if isinstance(a, InOutArgument)]
[r + x]
```

The full API reference can be viewed *here* (page 2045).

Autowrap

Autowrap automatically generates code, writes it to disk, compiles it, and imports it into the current session. Main functions of this module are `autowrap`, `binary_function`, and `ufuncify`.

It also automatically converts expressions containing `Indexed` objects into summations. The classes IndexedBase, Indexed and Idx represent a matrix element M[i, j]. See *Tensor* (page 1387) for more on this.

`autowrap` creates a wrapper using f2py or Cython and creates a numerical function.

Note: The `autowrap` callable is not in the sympy namespace automatically, to use it you must first import autowrap from sympy.utilities.autowrap

The callable returned from autowrap() is a binary Python function, not a SymPy object. For example:

```
>>> from sympy.utilities.autowrap import autowrap
>>> expr = ((x - y + z)**(13)).expand()
>>> binary_func = autowrap(expr)
>>> binary_func(1, 4, 2)
-1.0
```

The various flags available with autowrap() help to modify the services provided by the method. The argument tempdir tells autowrap to compile the code in a specific directory, and leave the files intact when finished. For instance:

```
>>> from sympy.utilities.autowrap import autowrap
>>> from sympy.physics.qho_1d import psi_n
>>> x_ = IndexedBase('x')
>>> y_ = IndexedBase('y')
>>> m = symbols('m', integer=True)
>>> i = Idx('i', m)
>>> qho = autowrap(Eq(y_[i], psi_n(0, x_[i], m, omega)), tempdir='/tmp')
```

Checking the Fortran source code in the directory specified reveals this:

```
subroutine autofunc(m, omega, x, y)
implicit none
INTEGER*4, intent(in) :: m
REAL*8, intent(in) :: omega
REAL*8, intent(in), dimension(1:m) :: x
REAL*8, intent(out), dimension(1:m) :: y
INTEGER*4 :: i

REAL*8, parameter :: hbar = 1.05457162d-34
REAL*8, parameter :: pi = 3.14159265358979d0
do i = 1, m
   y(i) = (m*omega)**(1.0d0/4.0d0)*exp(-4.74126166983329d+33*m*omega*x(i &
         )**2)/(hbar**(1.0d0/4.0d0)*pi**(1.0d0/4.0d0))
end do

end subroutine
```

Using the argument args along with it changes argument sequence:

```
>>> eq = Eq(y_[i], psi_n(0, x_[i], m, omega))
>>> qho = autowrap(eq, tempdir='/tmp', args=[y, x, m, omega])
```

yields:

```
subroutine autofunc(y, x, m, omega)
implicit none
INTEGER*4, intent(in) :: m
REAL*8, intent(in) :: omega
REAL*8, intent(out), dimension(1:m) :: y
```

(continues on next page)

(continued from previous page)

```
REAL*8, intent(in), dimension(1:m) :: x
INTEGER*4 :: i

REAL*8, parameter :: hbar = 1.05457162d-34
REAL*8, parameter :: pi = 3.14159265358979d0
do i = 1, m
   y(i) = (m*omega)**(1.0d0/4.0d0)*exp(-4.74126166983329d+33*m*omega*x(i &
         )**2)/(hbar**(1.0d0/4.0d0)*pi**(1.0d0/4.0d0))
end do

end subroutine
```

The argument `verbose` is boolean, optional and if True, autowrap will not mute the command line backends. This can be helpful for debugging.

The argument `language` and `backend` are used to change defaults: `Fortran` and `f2py` to `C` and `Cython`. The argument helpers is used to define auxiliary expressions needed for the main expression. If the main expression needs to call a specialized function it should be put in the `helpers` iterable. Autowrap will then make sure that the compiled main expression can link to the helper routine. Items should be tuples with `(<function_name>, <sympy_expression>, <arguments>)`. It is mandatory to supply an argument sequence to helper routines.

Another method available at the `autowrap` level is `binary_function`. It returns a sympy function. The advantage is that we can have very fast functions as compared to SymPy speeds. This is because we will be using compiled functions with SymPy attributes and methods. An illustration:

```
>>> from sympy.utilities.autowrap import binary_function
>>> from sympy.physics.hydrogen import R_nl
>>> psi_nl = R_nl(1, 0, a, r)
>>> f = binary_function('f', psi_nl)
>>> f(a, r).evalf(3, subs={a: 1, r: 2})
0.766
```

While NumPy operations are very efficient for vectorized data but they sometimes incur unnecessary costs when chained together. Consider the following operation

```
>>> x = get_numpy_array(...)
>>> y = sin(x) / x
```

The operators `sin` and `/` call routines that execute tight for loops in `C`. The resulting computation looks something like this

```
for(int i = 0; i < n; i++)
{
    temp[i] = sin(x[i]);
}
for(int i = i; i < n; i++)
{
    y[i] = temp[i] / x[i];
}
```

This is slightly sub-optimal because

1. We allocate an extra `temp` array