Examples

```
>>> from sympy.combinatorics import SymmetricGroup
>>> from sympy.combinatorics.util import _base_ordering
>>> S = SymmetricGroup(4)
>>> S.schreier_sims()
>>> _base_ordering(S.base, S.degree)
[0, 1, 2, 3]
```

Notes

This is used in backtrack searches, when we define a relation $\ll$ on the underlying set for a permutation group of degree n, $\{0, 1, \ldots, n-1\}$, so that if $(b_1, b_2, \ldots, b_k)$ is a base we have $b_i \ll b_j$ whenever $i < j$ and $b_i \ll a$ for all $i \in \{1, 2, \ldots, k\}$ and a is not in the base. The idea is developed and applied to backtracking algorithms in [1], pp.108-132. The points that are not in the base are taken in increasing order.

References

[R84]

sympy.combinatorics.util.**_check_cycles_alt_sym**(*perm*)

Checks for cycles of prime length p with n/2 < p < n-2.

Explanation

Here n is the degree of the permutation. This is a helper function for the function is_alt_sym from sympy.combinatorics.perm_groups.

Examples

```
>>> from sympy.combinatorics.util import _check_cycles_alt_sym
>>> from sympy.combinatorics import Permutation
>>> a = Permutation([[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10], [11, 12]])
>>> _check_cycles_alt_sym(a)
False
>>> b = Permutation([[0, 1, 2, 3, 4, 5, 6], [7, 8, 9, 10]])
>>> _check_cycles_alt_sym(b)
True
```

See also:

sympy.combinatorics.perm_groups.PermutationGroup.is_alt_sym (page 310)

sympy.combinatorics.util.**_distribute_gens_by_base**(*base*, *gens*)

Distribute the group elements gens by membership in basic stabilizers.

Parameters

``**base**`` : a sequence of points in $\{0, 1, \ldots, n-1\}$

``**gens**`` : a list of elements of a permutation group of degree n.

Returns

List of length k, where k is the length of `base`. The i-th entry contains those elements in `gens` which fix the first i elements of `base` (so that the 0-th entry is equal to `gens` itself). If no element fixes the first i elements of `base`, the i-th element is set to a list containing the identity element.

Explanation

Notice that for a base $(b_1, b_2, \ldots, b_k)$, the basic stabilizers are defined as $G^{(i)} = G_{b_1,\ldots,b_{i-1}}$ for $i \in \{1, 2, \ldots, k\}$.

Examples

```
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> from sympy.combinatorics.util import _distribute_gens_by_base
>>> D = DihedralGroup(3)
>>> D.schreier_sims()
>>> D.strong_gens
[(0 1 2), (0 2), (1 2)]
>>> D.base
[0, 1]
>>> _distribute_gens_by_base(D.base, D.strong_gens)
[[(0 1 2), (0 2), (1 2)],
 [(1 2)]]
```

See also:

_strong_gens_from_distr (page 362), *_orbits_transversals_from_bsgs* (page 359), *_handle_precomputed_bsgs* (page 358)

sympy.combinatorics.util.**_handle_precomputed_bsgs**(*base*, *strong_gens*, *transversals=None*, *basic_orbits=None*, *strong_gens_distr=None*)

Calculate BSGS-related structures from those present.

Parameters

``base`` - the base

``strong_gens`` - the strong generators

``transversals`` - basic transversals

``basic_orbits`` - basic orbits

``strong_gens_distr`` - strong generators distributed by membership in basic

stabilizers

Returns

`(transversals, basic_orbits, strong_gens_distr)` where `transversals` are the basic transversals, `basic_orbits` are the basic orbits, and `strong_gens_distr` are the strong generators distributed by membership in basic stabilizers.

Explanation

The base and strong generating set must be provided; if any of the transversals, basic orbits or distributed strong generators are not provided, they will be calculated from the base and strong generating set.

Examples

```
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> from sympy.combinatorics.util import _handle_precomputed_bsgs
>>> D = DihedralGroup(3)
>>> D.schreier_sims()
>>> _handle_precomputed_bsgs(D.base, D.strong_gens,
... basic_orbits=D.basic_orbits)
([{0: (2), 1: (0 1 2), 2: (0 2)}, {1: (2), 2: (1 2)}], [[0, 1, 2], [1,␣
↪2]], [[(0 1 2), (0 2), (1 2)], [(1 2)]])
```

See also:

_orbits_transversals_from_bsgs (page 359), *_distribute_gens_by_base* (page 357)

sympy.combinatorics.util.**_orbits_transversals_from_bsgs**(*base*, *strong_gens_distr*, *transversals_only=False*, *slp=False*)

Compute basic orbits and transversals from a base and strong generating set.

Parameters

``base`` - The base.

``strong_gens_distr`` - Strong generators distributed by membership in basic

stabilizers.

``transversals_only`` - bool

A flag switching between returning only the transversals and both orbits and transversals.

``slp`` -

If `True`, return a list of dictionaries containing the generator presentations of the elements of the transversals, i.e. the list of indices of generators from `strong_gens_distr[i]` such that their product is the relevant transversal element.

Explanation

The generators are provided as distributed across the basic stabilizers. If the optional argument `transversals_only` is set to True, only the transversals are returned.

Examples

```
>>> from sympy.combinatorics import SymmetricGroup
>>> from sympy.combinatorics.util import _distribute_gens_by_base
>>> S = SymmetricGroup(3)
>>> S.schreier_sims()
>>> strong_gens_distr = _distribute_gens_by_base(S.base, S.strong_gens)
>>> (S.base, strong_gens_distr)
([0, 1], [[(0 1 2), (2)(0 1), (1 2)], [(1 2)]])
```

See also:

_distribute_gens_by_base (page 357), *_handle_precomputed_bsgs* (page 358)

sympy.combinatorics.util.**_remove_gens**(*base*, *strong_gens*, *basic_orbits=None*, *strong_gens_distr=None*)

Remove redundant generators from a strong generating set.

Parameters

``base`` - a base

``strong_gens`` - a strong generating set relative to ``base``

``basic_orbits`` - basic orbits

``strong_gens_distr`` - strong generators distributed by membership in basic

stabilizers

Returns

A strong generating set with respect to `base` which is a subset of

`strong_gens`.

Examples

```
>>> from sympy.combinatorics import SymmetricGroup
>>> from sympy.combinatorics.util import _remove_gens
>>> from sympy.combinatorics.testutil import _verify_bsgs
>>> S = SymmetricGroup(15)
>>> base, strong_gens = S.schreier_sims_incremental()
>>> new_gens = _remove_gens(base, strong_gens)
>>> len(new_gens)
14
>>> _verify_bsgs(S, base, new_gens)
True
```

Notes

This procedure is outlined in [1],p.95.

References

[R85]

sympy.combinatorics.util.**_strip**(*g*, *base*, *orbits*, *transversals*)

Attempt to decompose a permutation using a (possibly partial) BSGS structure.

Parameters

``g`` - permutation to be decomposed

``base`` - sequence of points

``orbits`` - a list in which the ``i``-th entry is an orbit of ``base[i]``

under some subgroup of the pointwise stabilizer of `

`base[0], base[1], ..., base[i - 1]``. The groups themselves are implicit

in this function since the only information we need is encoded in the orbits

and transversals

``transversals`` - a list of orbit transversals associated with the orbits

``orbits``.

Explanation

This is done by treating the sequence `base` as an actual base, and the orbits `orbits` and transversals `transversals` as basic orbits and transversals relative to it.

This process is called “sifting”. A sift is unsuccessful when a certain orbit element is not found or when after the sift the decomposition does not end with the identity element.

The argument `transversals` is a list of dictionaries that provides transversal elements for the orbits `orbits`.

Examples

```
>>> from sympy.combinatorics import Permutation, SymmetricGroup
>>> from sympy.combinatorics.util import _strip
>>> S = SymmetricGroup(5)
>>> S.schreier_sims()
>>> g = Permutation([0, 2, 3, 1, 4])
>>> _strip(g, S.base, S.basic_orbits, S.basic_transversals)
((4), 5)
```

Notes

The algorithm is described in [1],pp.89-90. The reason for returning both the current state of the element being decomposed and the level at which the sifting ends is that they provide important information for the randomized version of the Schreier-Sims algorithm.

See also:

sympy.combinatorics.perm_groups.PermutationGroup.schreier_sims (page 324), *sympy.combinatorics.perm_groups.PermutationGroup.schreier_sims_random* (page 326)

References

[R86]

sympy.combinatorics.util.**_strong_gens_from_distr**(*strong_gens_distr*)

Retrieve strong generating set from generators of basic stabilizers.

This is just the union of the generators of the first and second basic stabilizers.

Parameters

``strong_gens_distr`` - strong generators distributed by membership in basic

stabilizers

Examples

```
>>> from sympy.combinatorics import SymmetricGroup
>>> from sympy.combinatorics.util import (_strong_gens_from_distr,
... _distribute_gens_by_base)
>>> S = SymmetricGroup(3)
>>> S.schreier_sims()
>>> S.strong_gens
[(0 1 2), (2)(0 1), (1 2)]
>>> strong_gens_distr = _distribute_gens_by_base(S.base, S.strong_gens)
>>> _strong_gens_from_distr(strong_gens_distr)
[(0 1 2), (2)(0 1), (1 2)]
```

See also:

_distribute_gens_by_base (page 357)

Group constructors

sympy.combinatorics.group_constructs.**DirectProduct**(**groups*)

Returns the direct product of several groups as a permutation group.

Explanation

This is implemented much like the __mul__ procedure for taking the direct product of two permutation groups, but the idea of shifting the generators is realized in the case of an arbitrary number of groups. A call to DirectProduct(G1, G2, ..., Gn) is generally expected to be faster than a call to G1*G2*...*Gn (and thus the need for this algorithm).

Examples

```
>>> from sympy.combinatorics.group_constructs import DirectProduct
>>> from sympy.combinatorics.named_groups import CyclicGroup
>>> C = CyclicGroup(4)
>>> G = DirectProduct(C, C, C)
>>> G.order()
64
```

See also:

sympy.combinatorics.perm_groups.PermutationGroup.__mul__ (page 289)

Test Utilities

sympy.combinatorics.testutil.**_cmp_perm_lists**(*first*, *second*)

Compare two lists of permutations as sets.

Explanation

This is used for testing purposes. Since the array form of a permutation is currently a list, Permutation is not hashable and cannot be put into a set.

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.combinatorics.testutil import _cmp_perm_lists
>>> a = Permutation([0, 2, 3, 4, 1])
>>> b = Permutation([1, 2, 0, 4, 3])
>>> c = Permutation([3, 4, 0, 1, 2])
>>> ls1 = [a, b, c]
>>> ls2 = [b, c, a]
>>> _cmp_perm_lists(ls1, ls2)
True
```

sympy.combinatorics.testutil.**_naive_list_centralizer**(*self*, *other*, *af=False*)

sympy.combinatorics.testutil.**_verify_bsgs**(*group*, *base*, *gens*)

Verify the correctness of a base and strong generating set.

Explanation

This is a naive implementation using the definition of a base and a strong generating set relative to it. There are other procedures for verifying a base and strong generating set, but this one will serve for more robust testing.

Examples

```
>>> from sympy.combinatorics.named_groups import AlternatingGroup
>>> from sympy.combinatorics.testutil import _verify_bsgs
>>> A = AlternatingGroup(4)
>>> A.schreier_sims()
>>> _verify_bsgs(A, A.base, A.strong_gens)
True
```

See also:

sympy.combinatorics.perm_groups.PermutationGroup.schreier_sims (page 324)

sympy.combinatorics.testutil.**_verify_centralizer**(*group*, *arg*, *centr=None*)

Verify the centralizer of a group/set/element inside another group.

This is used for testing `.centralizer()` from `sympy.combinatorics.perm_groups`

Examples

```
>>> from sympy.combinatorics.named_groups import (SymmetricGroup,
... AlternatingGroup)
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.combinatorics.testutil import _verify_centralizer
>>> S = SymmetricGroup(5)
>>> A = AlternatingGroup(5)
>>> centr = PermutationGroup([Permutation([0, 1, 2, 3, 4])])
>>> _verify_centralizer(S, A, centr)
True
```

See also:

_naive_list_centralizer (page 363), *sympy.combinatorics.perm_groups.PermutationGroup.centralizer* (page 299), *_cmp_perm_lists* (page 363)

sympy.combinatorics.testutil.**_verify_normal_closure**(*group*, *arg*, *closure=None*)

Tensor Canonicalization

sympy.combinatorics.tensor_can.**canonicalize**(*g*, *dummies*, *msym*, **v*)

canonicalize tensor formed by tensors

Parameters

g : permutation representing the tensor

dummies : list representing the dummy indices

it can be a list of dummy indices of the same type or a list of lists of dummy indices, one list for each type of index; the dummy indices must come after the free indices, and put in order contravariant, covariant [d0, -d0, d1,-d1,...]

msym : symmetry of the metric(s)

it can be an integer or a list; in the first case it is the symmetry of the dummy index metric; in the second case it is the list of the symmetries of the index metric for each type

v : list, (base_i, gens_i, n_i, sym_i) for tensors of type i

base_i, gens_i : BSGS for tensors of this type.

The BSGS should have minimal base under lexicographic ordering; if not, an attempt is made do get the minimal BSGS; in case of failure, canonicalize_naive is used, which is much slower.

n_i : number of tensors of type i.

sym_i : symmetry under exchange of component tensors of type i.

Both for msym and sym_i the cases are

- None no symmetry
- 0 commuting
- 1 anticommuting

Returns

0 if the tensor is zero, else return the array form of

the permutation representing the canonical form of the tensor.

Algorithm

First one uses canonical_free to get the minimum tensor under lexicographic order, using only the slot symmetries. If the component tensors have not minimal BSGS, it is attempted to find it; if the attempt fails canonicalize_naive is used instead.

Compute the residual slot symmetry keeping fixed the free indices using tensor_gens(base, gens, list_free_indices, sym).

Reduce the problem eliminating the free indices.

Then use double_coset_can_rep and lift back the result reintroducing the free indices.

Examples

one type of index with commuting metric;

A_{ab} and B_{ab} antisymmetric and commuting

$T = A_{d0d1} * B^{d0}{}_{d2} * B^{d2d1}$

$ord = [d0, -d0, d1, -d1, d2, -d2]$ order of the indices

g = [1, 3, 0, 5, 4, 2, 6, 7]

$T_c = 0$

```
>>> from sympy.combinatorics.tensor_can import get_symmetric_group_sgs,␣
↪canonicalize, bsgs_direct_product
>>> from sympy.combinatorics import Permutation
>>> base2a, gens2a = get_symmetric_group_sgs(2, 1)
>>> t0 = (base2a, gens2a, 1, 0)
>>> t1 = (base2a, gens2a, 2, 0)
>>> g = Permutation([1, 3, 0, 5, 4, 2, 6, 7])
>>> canonicalize(g, range(6), 0, t0, t1)
0
```

same as above, but with B_{ab} anticommuting

$T_c = -A^{d0d1} * B_{d0}{}^{d2} * B_{d1d2}$

can = [0,2,1,4,3,5,7,6]

```
>>> t1 = (base2a, gens2a, 2, 1)
>>> canonicalize(g, range(6), 0, t0, t1)
[0, 2, 1, 4, 3, 5, 7, 6]
```

two types of indices $[a, b, c, d, e, f]$ and $[m, n]$, in this order, both with commuting metric

f^{abc} antisymmetric, commuting

A_{ma} no symmetry, commuting

$T = f^c{}_{da} * f^f{}_{eb} * A_m{}^d * A^{mb} * A_n{}^a * A^{ne}$

ord = [c,f,a,-a,b,-b,d,-d,e,-e,m,-m,n,-n]

g = [0,7,3, 1,9,5, 11,6, 10,4, 13,2, 12,8, 14,15]

The canonical tensor is $T_c = -f^{cab} * f^{fde} * A^m{}_a * A_{md} * A^n{}_b * A_{ne}$

can = [0,2,4, 1,6,8, 10,3, 11,7, 12,5, 13,9, 15,14]

```
>>> base_f, gens_f = get_symmetric_group_sgs(3, 1)
>>> base1, gens1 = get_symmetric_group_sgs(1)
>>> base_A, gens_A = bsgs_direct_product(base1, gens1, base1, gens1)
>>> t0 = (base_f, gens_f, 2, 0)
>>> t1 = (base_A, gens_A, 4, 0)
>>> dummies = [range(2, 10), range(10, 14)]
>>> g = Permutation([0, 7, 3, 1, 9, 5, 11, 6, 10, 4, 13, 2, 12, 8, 14,␣
↪15])
>>> canonicalize(g, dummies, [0, 0], t0, t1)
[0, 2, 4, 1, 6, 8, 10, 3, 11, 7, 12, 5, 13, 9, 15, 14]
```

sympy.combinatorics.tensor_can.**double_coset_can_rep**(*dummies*, *sym*, *b_S*, *sgens*, *S_transversals*, *g*)

Butler-Portugal algorithm for tensor canonicalization with dummy indices.

Parameters

dummies

list of lists of dummy indices, one list for each type of index; the dummy indices are put in order contravariant, covariant [d0, -d0, d1, -d1, ...].

sym

list of the symmetries of the index metric for each type.

possible symmetries of the metrics

- 0 symmetric
- 1 antisymmetric
- None no symmetry

b_S

base of a minimal slot symmetry BSGS.

sgens

generators of the slot symmetry BSGS.

S_transversals

transversals for the slot BSGS.

g

permutation representing the tensor.

Returns

Return 0 if the tensor is zero, else return the array form of the permutation representing the canonical form of the tensor.

Notes

A tensor with dummy indices can be represented in a number of equivalent ways which typically grows exponentially with the number of indices. To be able to establish if two tensors with many indices are equal becomes computationally very slow in absence of an efficient algorithm.

The Butler-Portugal algorithm [3] is an efficient algorithm to put tensors in canonical form, solving the above problem.

Portugal observed that a tensor can be represented by a permutation, and that the class of tensors equivalent to it under slot and dummy symmetries is equivalent to the double coset $D * g * S$ (Note: in this documentation we use the conventions for multiplication of permutations p, q with (p*q)(i) = p[q[i]] which is opposite to the one used in the Permutation class)

Using the algorithm by Butler to find a representative of the double coset one can find a canonical form for the tensor.

To see this correspondence, let g be a permutation in array form; a tensor with indices ind (the indices including both the contravariant and the covariant ones) can be written as

$t = T(ind[g[0]], \ldots, ind[g[n-1]])$,

where $n = len(ind)$; g has size $n+2$, the last two indices for the sign of the tensor (trick introduced in [4]).

A slot symmetry transformation s is a permutation acting on the slots $t \to T(ind[(g*s)[0]], \ldots, ind[(g*s)[n-1]])$

A dummy symmetry transformation acts on ind $t \to T(ind[(d*g)[0]], \ldots, ind[(d*g)[n-1]])$

Being interested only in the transformations of the tensor under these symmetries, one can represent the tensor by g, which transforms as

$g- > d*g*s$, so it belongs to the coset $D*g*S$, or in other words to the set of all permutations allowed by the slot and dummy symmetries.

Let us explain the conventions by an example.

Given a tensor $T^{d3d2d1}{}_{d1d2d3}$ with the slot symmetries

$T^{a0a1a2a3a4a5} = -T^{a2a1a0a3a4a5}$

$T^{a0a1a2a3a4a5} = -T^{a4a1a2a3a0a5}$

and symmetric metric, find the tensor equivalent to it which is the lowest under the ordering of indices: lexicographic ordering $d1, d2, d3$ and then contravariant before covariant index; that is the canonical form of the tensor.

The canonical form is $-T^{d1d2d3}{}_{d1d2d3}$ obtained using $T^{a0a1a2a3a4a5} = -T^{a2a1a0a3a4a5}$.

To convert this problem in the input for this function, use the following ordering of the index names (- for covariant for short) $d1, -d1, d2, -d2, d3, -d3$

$T^{d3d2d1}{}_{d1d2d3}$ corresponds to $g = [4, 2, 0, 1, 3, 5, 6, 7]$ where the last two indices are for the sign

$sgens = [Permutation(0, 2)(6, 7), Permutation(0, 4)(6, 7)]$

sgens[0] is the slot symmetry $-(0, 2)$ $T^{a0a1a2a3a4a5} = -T^{a2a1a0a3a4a5}$

sgens[1] is the slot symmetry $-(0, 4)$ $T^{a0a1a2a3a4a5} = -T^{a4a1a2a3a0a5}$

The dummy symmetry group D is generated by the strong base generators $[(0, 1), (2, 3), (4, 5), (0, 2)(1, 3), (0, 4)(1, 5)]$ where the first three interchange covariant and contravariant positions of the same index (d1 <-> -d1) and the last two interchange the dummy indices themselves (d1 <-> d2).

The dummy symmetry acts from the left $d = [1, 0, 2, 3, 4, 5, 6, 7]$ exchange $d1 \leftrightarrow -d1$ $T^{d3d2d1}{}_{d1d2d3} == T^{d3d2}{}_{d1}{}^{d1}{}_{d2d3}$

$g = [4, 2, 0, 1, 3, 5, 6, 7]- > [4, 2, 1, 0, 3, 5, 6, 7] =_a f_r mul(d, g)$ which differs from ${}_a f_r mul(g, d)$.

The slot symmetry acts from the right $s = [2, 1, 0, 3, 4, 5, 7, 6]$ exchanges slots 0 and 2 and changes sign $T^{d3d2d1}{}_{d1d2d3} == -T^{d1d2d3}{}_{d1d2d3}$

$g = [4, 2, 0, 1, 3, 5, 6, 7]- > [0, 2, 4, 1, 3, 5, 7, 6] =_a f_r mul(g, s)$

Example in which the tensor is zero, same slot symmetries as above: $T^{d2}{}_{d1d3}{}^{d1d3}{}_{d2}$

$= -T^{d3}{}_{d1d3}{}^{d1d2}{}_{d2}$ under slot symmetry $-(0, 4)$;

$= T_{d3d1}{}^{d3d1d2}{}_{d2}$ under slot symmetry $-(0, 2)$;

$= T^{d3}{}_{d1d3}{}^{d1d2}{}_{d2}$ symmetric metric;

$= 0$ since two of these lines have tensors differ only for the sign.

The double coset D*g*S consists of permutations $h = d * g * s$ corresponding to equivalent tensors; if there are two h which are the same apart from the sign, return zero; otherwise choose as representative the tensor with indices ordered lexicographically according to $[d1, -d1, d2, -d2, d3, -d3]$ that is `rep = min(D*g*S) = min([d*g*s for d in D for s in S])`

The indices are fixed one by one; first choose the lowest index for slot 0, then the lowest remaining index for slot 1, etc. Doing this one obtains a chain of stabilizers

$S \to S_{b0} \to S_{b0,b1} \to \dots$ and $D \to D_{p0} \to D_{p0,p1} \to \dots$

where `[b0, b1, ...] = range(b)` is a base of the symmetric group; the strong base b_S of S is an ordered sublist of it; therefore it is sufficient to compute once the strong base generators of S using the Schreier-Sims algorithm; the stabilizers of the strong base generators are the strong base generators of the stabilizer subgroup.

`dbase = [p0, p1, ...]` is not in general in lexicographic order, so that one must recompute the strong base generators each time; however this is trivial, there is no need to use the Schreier-Sims algorithm for D.

The algorithm keeps a TAB of elements (s_i, d_i, h_i) where $h_i = d_i \times g \times s_i$ satisfying $h_i[j] = p_j$ for $0 \leq j < i$ starting from $s_0 = id, d_0 = id, h_0 = g$.

The equations $h_0[0] = p_0, h_1[1] = p_1, \dots$ are solved in this order, choosing each time the lowest possible value of p_i

For $j < i$ $d_i * g * s_i * S_{b_0,\dots,b_{i-1}} * b_j = D_{p_0,\dots,p_{i-1}} * p_j$ so that for dx in $D_{p_0,\dots,p_{i-1}}$ and sx in $S_{base[0],\dots,base[i-1]}$ one has $dx * d_i * g * s_i * sx * b_j = p_j$

Search for dx, sx such that this equation holds for $j = i$; it can be written as $s_i * sx * b_j = J, dx * d_i * g * J = p_j$ $sx * b_j = s_i * * -1 * J$; $sx = trace(s_i * * -1, S_{b_0,\dots,b_{i-1}})$ $dx * * -1 * p_j = d_i * g * J$; $dx = trace(d_i * g * J, D_{p_0,\dots,p_{i-1}})$

$s_{i+1} = s_i * trace(s_i * * -1 * J, S_{b_0,\dots,b_{i-1}})$ $d_{i+1} = trace(d_i * g * J, D_{p_0,\dots,p_{i-1}}) * * -1 * d_i$ $h_{i+1} * b_i = d_{i+1} * g * s_{i+1} * b_i = p_i$

$h_n * b_j = p_j$ for all j, so that h_n is the solution.

Add the found (s, d, h) to TAB1.

At the end of the iteration sort TAB1 with respect to the h; if there are two consecutive h in TAB1 which differ only for the sign, the tensor is zero, so return 0; if there are two consecutive h which are equal, keep only one.

Then stabilize the slot generators under i and the dummy generators under p_i.

Assign $TAB = TAB1$ at the end of the iteration step.

At the end TAB contains a unique (s, d, h), since all the slots of the tensor h have been fixed to have the minimum value according to the symmetries. The algorithm returns h.

It is important that the slot BSGS has lexicographic minimal base, otherwise there is an i which does not belong to the slot base for which p_i is fixed by the dummy symmetry only, while i is not invariant from the slot stabilizer, so p_i is not in general the minimal value.

This algorithm differs slightly from the original algorithm [3]:

the canonical form is minimal lexicographically, and the BSGS has minimal base under lexicographic order. Equal tensors h are eliminated from TAB.

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.combinatorics.tensor_can import double_coset_can_rep, get_
↪transversals
>>> gens = [Permutation(x) for x in [[2, 1, 0, 3, 4, 5, 7, 6], [4, 1, 2,␣
↪3, 0, 5, 7, 6]]]
>>> base = [0, 2]
>>> g = Permutation([4, 2, 0, 1, 3, 5, 6, 7])
>>> transversals = get_transversals(base, gens)
>>> double_coset_can_rep([list(range(6))], [0], base, gens, transversals,
↪ g)
[0, 1, 2, 3, 4, 5, 7, 6]
```

```
>>> g = Permutation([4, 1, 3, 0, 5, 2, 6, 7])
>>> double_coset_can_rep([list(range(6))], [0], base, gens, transversals,
↪ g)
0
```

sympy.combinatorics.tensor_can.**get_symmetric_group_sgs**(*n*, *antisym=False*)

Return base, gens of the minimal BSGS for (anti)symmetric tensor

Parameters

``n``: rank of the tensor

``antisym`` : bool

`antisym = False` symmetric tensor `antisym = True` antisymmetric tensor

Examples

```
>>> from sympy.combinatorics.tensor_can import get_symmetric_group_sgs
>>> get_symmetric_group_sgs(3)
([0, 1], [(4)(0 1), (4)(1 2)])
```

sympy.combinatorics.tensor_can.**bsgs_direct_product**(*base1*, *gens1*, *base2*, *gens2*, *signed=True*)

Direct product of two BSGS.

Parameters

base1 : base of the first BSGS.

gens1 : strong generating sequence of the first BSGS.

base2, gens2 : similarly for the second BSGS.

signed : flag for signed permutations.

Examples

```
>>> from sympy.combinatorics.tensor_can import (get_symmetric_group_sgs,␣
↪bsgs_direct_product)
>>> base1, gens1 = get_symmetric_group_sgs(1)
>>> base2, gens2 = get_symmetric_group_sgs(2)
>>> bsgs_direct_product(base1, gens1, base2, gens2)
([1], [(4)(1 2)])
```

Finitely Presented Groups

Introduction

This module presents the functionality designed for computing with finitely- presented groups (fp-groups for short). The name of the corresponding SymPy object is `FpGroup`. The functions or classes described here are studied under **computational group theory**. All code examples assume:

```
>>> from sympy.combinatorics.free_groups import free_group, vfree_group,␣
↪xfree_group
>>> from sympy.combinatorics.fp_groups import FpGroup, CosetTable, coset_
↪enumeration_r
```

Overview of Facilities

The facilities provided for fp-groups fall into a number of natural groupings

- The construction of fp-groups using a free group and a list of words in generators of that free group.
- Index determination using the famous Todd-Coxeter procedure.
- The construction of all subgroups having index less than some (small) specified positive integer, using the *Low-Index Subgroups* algorithm.
- Algorithms for computing presentations of a subgroup of finite index in a group defined by finite presentation.

For a description of fundamental algorithms of finitely presented groups we often make use of *Handbook of Computational Group Theory*.

The Construction of Finitely Presented Groups

Finitely presented groups are constructed by factoring a free group by a set of relators. The set of relators is taken in as a list of words in generators of free group in SymPy, using a list provides ordering to the relators. If the list of relators is empty, the associated free group is returned.

Example of construction of a finitely-presented group. The symmetric group of degree 4 may be represented as a two generator group with presentation $\langle a, b \mid a^2, b^3, (ab)^4 \rangle$. Giving the relations as a list of relators, group in SymPy would be specified as:

```
>>> F, a, b = free_group("a, b")
>>> G = FpGroup(F, [a**2, b**3, (a*b)**4])
>>> G
<fp group on the generators (a, b)>
```

Currently groups with relators having presentation like $\langle r, s, t \mid r^2, s^2, t^2, rst = str = trs \rangle$ will have to be specified as:

```
>>> F, r, s, t = free_group("r, s, t")
>>> G = FpGroup(F, [r**2, s**2, t**2, r*s*t*r**-1*t**-1*s**-1, s*t*r*s**-
↪1*r**-1*t**-1])
```

Obviously this is not a unique way to make that particular group, but the point is that in case of equality with non-identity the user has to manually do that.

Free Groups and Words

Construction of a Free Group

`free_group("gen0, gen1, ..., gen_(n-1)")` constructs a free group `F` on `n` generators, where `n` is a positive integer. The i-th generator of F may be obtained using the method `.generators[i]`, $i = 0, \ldots n-1$.

```
>>> F, x, y = free_group("x, y")
```

creates a free group `F` of rank 2 and assigns the variables `x` and `y` to the two generators.

```
>>> F = vfree_group("x, y")
>>> F
<free group on the generators (x, y)>
```

creates a free group `F` of rank 2, with tuple of generators `F.generators`, and inserts `x` and `y` as generators into the global namespace.

```
>>> F = xfree_group("x, y")
>>> F
(<free group on the generators (x, y)>, (x, y))
>>> x**2
x**2
```

creates a free groups `F[0]` of rank 2, with tuple of generators `F[1]`.

Construction of words

This section is applicable to words of `FreeGroup` as well as `FpGroup`. When we say *word* in SymPy, it actually means a reduced word , since the words are automatically reduced. Given a group `G` defined on n generators $x_1, x_2, x_3, \ldots, x_n$, a word is constructed as $s_1^{r_1} s_2^{r_2} \cdots s_k^{r_k}$ where $s_i \in \{x_1, x_2, \ldots, x_n\}$, $r_i \in \mathbb{Z}$ for all k.

Each word can be constructed in a variety of ways, since after reduction they may be equivalent.

Coset Enumeration: The Todd-Coxeter Algorithm

This section describes the use of coset enumeration techniques in SymPy. The algorithm used for coset enumeration procedure is Todd-Coxeter algorithm and is developed in SymPy using [Ho05] and [CDHW73]. The reader should consult [CDHW73] and [Hav91] for a general description of the algorithm.

We have two strategies of coset enumeration *relator-based* and *coset-table based* and the two have been implemented as `coset_enumeration_r`, `coset_enumeration_c` respectively. The two strategies differ in the way they make new definitions for the cosets.

Though from the user point of view it is suggested to rather use the `.coset_enumeration` method of `FpGroup` and specify the `strategy` argument.

`strategy`:
(default="relator_based") specifies the strategy of coset enumeration to be used, possible values are *"relator_based"* or *"coset_table_based"*.

CosetTable

Class used to manipulate the information regarding the coset enumeration of the finitely presented group `G` on the cosets of the subgroup `H`.

Basically a *coset table* `CosetTable(G,H)`, is the permutation representation of the finitely presented group on the cosets of a subgroup. Most of the set theoretic and group functions use the regular representation of `G`, i.e., the coset table of `G` over the trivial subgroup.

The actual mathematical coset table is obtained using `.table` attribute and is a list of lists. For each generator `g` of `G` it contains a column and the next column corresponds to `g**-1` and so on for other generators, so in total it has `2*G.rank()` columns. Each column is simply a list of integers. If `l` is the generator list for the generator g and if `l[i] = j` then generator `g` takes the coset i to the coset j by multiplication from the right.

For finitely presented groups, a coset table is computed by a Todd-Coxeter coset enumeration. Note that you may influence the performance of that enumeration by changing the values of the variable `CosetTable.coset_table_max_limit`.

Attributes of CosetTable

For `CosetTable(G, H)` where `G` is the group and `H` is the subgroup.

- `n`: A non-negative integer, non-mutable attribute, dependently calculated as the maximum among the live-cosets (i.e. Ω).
- `table`: A list of lists, mutable attribute, mathematically represents the coset table.
- `omega`: A list, dependent on the internal attribute `p`. Ω represents the list of live-cosets. A *standard* coset-table has its $\Omega = \{0, 1, \ldots, index - 1\}$ where $index$ is the index of subgroup H in G.

For experienced users we have a number of parameters that can be used to manipulate the algorithm, like

- `coset_table_max_limit` (default value = 4096000): manipulate the maximum number of cosets allowed in coset enumeration, i.e. the number of rows allowed in coset table. A coset enumeration will not finish if the subgroup does not have finite index, and even if it has it may take many more intermediate cosets than the actual index of the subgroup

is. To avoid a coset enumeration "running away" therefore SymPy has a "safety stop" built-in. This is controlled by this variable. To change it, use max_cosets keyword. For example:

```
>>> F, a, b = free_group("a, b")
>>> Cox = FpGroup(F, [a**6, b**6, (a*b)**2, (a**2*b**2)**2,␣
↪(a**3*b**3)**5])
>>> C_r = coset_enumeration_r(Cox, [a], max_cosets=50)
Traceback (most recent call last):
  ...
ValueError: the coset enumeration has defined more than 50 cosets
```

- `max_stack_size` (default value = 500): manipulate the maximum size of `deduction_stack` above or equal to which the stack is emptied.

Compression and Standardization

For any two entries i, j with $i < j$ in coset table, the first occurrence of i in a coset table precedes the first occurrence of j with respect to the usual row-wise ordering of the table entries. We call such a table a standard coset table. To standardize a `CosetTable` we use the `.standardize` method.

Note the method alters the given table, it does not create a copy.

Subgroups of Finite Index

The functionality in this section are concerned with the construction of subgroups of finite index. We describe a method for computing all subgroups whose index does not exceed some (modest) integer bound.

Low Index Subgroups

`low_index_subgroups(G, N)`: Given a finitely presented group $G = \langle X \mid R \rangle$ (can be a free group), and `N` a positive integer, determine the conjugacy classes of subgroups of `G` whose indices is less than or equal to `N`.

For example to find all subgroups of $G = \langle a, b \mid a^2 = b^3 = (ab)^4 = 1 \rangle$ having index ≤ 4, can be found as follows:

```
>>> from sympy.combinatorics.fp_groups import low_index_subgroups
>>> F, a, b = free_group("a, b")
>>> G = FpGroup(F, [a**2, b**3, (a*b)**4])
>>> l = low_index_subgroups(G, 4)
>>> for coset_table in l:
...     print(coset_table.table)
...
[[0, 0, 0, 0]]
[[0, 0, 1, 2], [1, 1, 2, 0], [3, 3, 0, 1], [2, 2, 3, 3]]
[[0, 0, 1, 2], [2, 2, 2, 0], [1, 1, 0, 1]]
[[1, 1, 0, 0], [0, 0, 1, 1]]
```

This returns the coset tables of subgroups of satisfying the property that index, $index$, of subgroup in group is $\leq n$.

Constructing a presentation for a subgroup

In this section we discuss finding the presentation of a subgroup in a finitely presentation group. While the *subgroup* is currently allowed as input only in the form of a list of generators for the subgroup, you can expect the functionality of a *coset table* as input for subgroup in the group in near future.

There are two ways to construct a set of defining relations for subgroup from those of `G`. First is on a set of Schreier generators, known generally as Reidemeister-Schreier algorithm or on the given list of generators of `H`.

Reidemeister Schreier algorithm

called using `reidemeister_presentation(G, Y)` where `G` is the group and `Y` is a list of generators for subgroup `H` whose presentation we want to find.

```
>>> from sympy.combinatorics.fp_groups import reidemeister_presentation
>>> F, x, y = free_group("x, y")
>>> f = FpGroup(F, [x**3, y**5, (x*y)**2])
>>> H = [x*y, x**-1*y**-1*x*y*x]
>>> p1 = reidemeister_presentation(f, H)
>>> p1
((y_1, y_2), (y_1**2, y_2**3, y_2*y_1*y_2*y_1*y_2*y_1))
```


Bibliography

Polycyclic Groups

Introduction

This module presents the functionality designed for computing with polycyclic groups(PcGroup for short). The name of the corresponding SymPy object is `PolycyclicGroup`. The functions or classes described here are studied under **Computational Group Theory**.

Overview of functionalities

- The construction of PolycyclicGroup from a given PermutationGroup.
- Computation of polycyclic generating sequence(pcgs for short) and polycyclic series(pc_series).
- Computation of relative order for polycyclic series.
- Implementation of class Collector which can be treated as a base for polycylic groups.
- Implementation of polycyclic group presentation(pc_presentation for short).

- Computation of exponent vector, depth and leading exponent for a given element of a polycyclic group.

For a description of fundamental algorithms of polycyclic groups, we often make use of *Handbook of Computational Group Theory*.

The Construction of Polycyclic Groups

Given a Permutation Group, A Polycyclic Group is constructed by computing the corresponding polycylic generating sequence, polycyclic series and it's relative order.

Attributes of PolycyclicGroup

- `pc_sequence` : Polycyclic sequence is formed by collecting all the missing generators between the adjacent groups in the derived series of given permutation group.
- `pc_series` : Polycyclic series is formed by adding all the missing generators of `der[i+1]` in `der[i]`, where `der` represents derived series.
- `relative_order` : A list, computed by the ratio of adjacent groups in pc_series.
- `collector` : By default, it is None. Collector class provides the polycyclic presentation.

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> G = SymmetricGroup(4)
>>> PcGroup = G.polycyclic_group()
>>> len(PcGroup.pcgs)
4
>>> pc_series = PcGroup.pc_series
>>> pc_series[0].equals(G)  # use equals, not literal `==`
True
>>> gen = pc_series[len(pc_series) - 1].generators[0]
>>> gen.is_identity
True
>>> PcGroup.relative_order
[2, 3, 2, 2]
```

The Construction of Collector

Collector is one of the attributes of class PolycyclicGroup.

Attributes of Collector

Collector posses all the attributes of PolycyclicGroup, In addition there are few more attributes which are defined below:

- `free_group` : free_group provides the mapping of polycyclic generating sequence with the free group elements.
- `pc_presentation` : Provides the presentation of polycyclic groups with the help of power and conjugate relators.

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> G = SymmetricGroup(3)
>>> PcGroup = G.polycyclic_group()
>>> Collector = PcGroup.collector
>>> Collector.free_group
<free group on the generators (x0, x1)>
>>> Collector.pc_presentation
{x0**2: (), x1**3: (), x0**-1*x1*x0: x1**2}
```

Computation of Minimal Uncollected Subword

A word V defined on generators in the free_group of pc_group is a minimal uncollected subword of the word W if V is a subword of W and it has one of the following form:

- $v = {x_{i+1}}^{a_j} x_i$
- $v = {x_{i+1}}^{a_j} {x_i}^{-1}$
- $v = {x_i}^{a_j}$

$a_j \notin \{0, \ldots \text{relative\_order}[j] - 1\}$.

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.free_groups import free_group
>>> G = SymmetricGroup(4)
>>> PcGroup = G.polycyclic_group()
>>> collector = PcGroup.collector
>>> F, x1, x2 = free_group("x1, x2")
>>> word = x2**2*x1**7
>>> collector.minimal_uncollected_subword(word)
((x2, 2),)
```

Computation of Subword Index

For a given word and it's subword, subword_index computes the starting and ending index of the subword in the word.

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.free_groups import free_group
>>> G = SymmetricGroup(4)
>>> PcGroup = G.polycyclic_group()
>>> collector = PcGroup.collector
>>> F, x1, x2 = free_group("x1, x2")
>>> word = x2**2*x1**7
>>> w = x2**2*x1
>>> collector.subword_index(word, w)
(0, 3)
>>> w = x1**7
>>> collector.subword_index(word, w)
(2, 9)
```

Computation of Collected Word

A word W is called collected, if $W = x_{i_1}{}^{a_1} \dots x_{i_r}{}^{a_r}$ with $i_1 < i_2 < \dots < i_r$ and a_j is in $\{1 \dots s_{j-1}\}$, where s_j represents the respective relative order.

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.free_groups import free_group
>>> G = SymmetricGroup(4)
>>> PcGroup = G.polycyclic_group()
>>> collector = PcGroup.collector
>>> F, x0, x1, x2, x3 = free_group("x0, x1, x2, x3")
>>> word = x3*x2*x1*x0
>>> collected_word = collector.collected_word(word)
>>> free_to_perm = {}
>>> free_group = collector.free_group
>>> for sym, gen in zip(free_group.symbols, collector.pcgs):
...     free_to_perm[sym] = gen
>>> G1 = PermutationGroup()
>>> for w in word:
...     sym = w[0]
...     perm = free_to_perm[sym]
...     G1 = PermutationGroup([perm] + G1.generators)
>>> G2 = PermutationGroup()
>>> for w in collected_word:
...     sym = w[0]
...     perm = free_to_perm[sym]
...     G2 = PermutationGroup([perm] + G2.generators)
```

The two are not identical but they are equivalent:

```
>>> G1 == G2
False
>>> G1.equals(G2)
True
```

Computation of Polycyclic Presentation

The computation of presentation starts from the bottom of the pcgs and polycyclic series. Storing all the previous generators from pcgs and then taking the last generator as the generator which acts as a conjugator and conjugates all the previous generators in the list.

To get a clear picture, start with an example of SymmetricGroup(4). For S(4) there are 4 generators in pcgs say $[x_0, x_1, x_2, x_3]$ and the relative_order vector is [2, 3, 2, 2]. Starting from bottom of this sequence the presentation is computed in order as below.

using only $[x_3]$ from `pcgs` and `pc_series[4]` compute:

- x_3^2

using only $[x_3]$ from `pcgs` and `pc_series[3]` compute:

- x_2^2
- $x_2^{-1} x_3 x_2$

using $[x_3, x_2]$ from pcgs and pc_series[2] compute:

- x_1^3
- $x_1^{-1}x_3x_1$
- $x_1^{-1}x_2x_1$

using $[x_3, x_2, x_1]$ from pcgs and pc_series[1] compute:

- x_0^2
- $x_0^{-1}x_3x_0$
- $x_0^{-1}x_2x_0$
- $x_0^{-1}x_1x_0$

One thing to note is same group can have different pcgs due to variying derived_series which, results in different polycyclic presentations.

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.permutations import Permutation
>>> G = SymmetricGroup(4)
>>> PcGroup = G.polycyclic_group()
>>> collector = PcGroup.collector
>>> pcgs = PcGroup.pcgs
>>> len(pcgs)
4
>>> free_group = collector.free_group
>>> pc_resentation = collector.pc_presentation
>>> free_to_perm = {}
>>> for s, g in zip(free_group.symbols, pcgs):
...     free_to_perm[s] = g
>>> for k, v in pc_resentation.items():
...     k_array = k.array_form
...     if v != ():
...        v_array = v.array_form
...     lhs = Permutation()
...     for gen in k_array:
...         s = gen[0]
...         e = gen[1]
...         lhs = lhs*free_to_perm[s]**e
...     if v == ():
...         assert lhs.is_identity
...         continue
...     rhs = Permutation()
...     for gen in v_array:
...         s = gen[0]
...         e = gen[1]
...         rhs = rhs*free_to_perm[s]**e
...     assert lhs == rhs
```

Computation of Exponent Vector

Any generator of the polycyclic group can be represented with the help of it's polycyclic generating sequence. Hence, the length of exponent vector is equal to the length of the pcgs.

A given generator g of the polycyclic group, can be represented as $g = x_1^{e_1} \dots x_n^{e_n}$, where x_i represents polycyclic generators and n is the number of generators in the free_group equal to the length of pcgs.

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.permutations import Permutation
>>> G = SymmetricGroup(4)
>>> PcGroup = G.polycyclic_group()
>>> collector = PcGroup.collector
>>> pcgs = PcGroup.pcgs
>>> collector.exponent_vector(G[0])
[1, 0, 0, 0]
>>> exp = collector.exponent_vector(G[1])
>>> g = Permutation()
>>> for i in range(len(exp)):
...     g = g*pcgs[i]**exp[i] if exp[i] else g
>>> assert g == G[1]
```

Depth of Polycyclic generator

Depth of a given polycyclic generator is defined as the index of the first non-zero entry in the exponent vector.

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> G = SymmetricGroup(3)
>>> PcGroup = G.polycyclic_group()
>>> collector = PcGroup.collector
>>> collector.depth(G[0])
2
>>> collector.depth(G[1])
1
```

Computation of Leading Exponent

Leading exponent represents the exponent of polycyclic generator at the above depth.

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> G = SymmetricGroup(3)
>>> PcGroup = G.polycyclic_group()
>>> collector = PcGroup.collector
>>> collector.leading_exponent(G[1])
1
```

Bibliography

Functions

All functions support the methods documented below, inherited from *sympy.core.function.Function* (page 1050).

class sympy.core.function.**Function**(**args*)

Base class for applied mathematical functions.

It also serves as a constructor for undefined function classes.

See the *Writing Custom Functions* (page 102) guide for details on how to subclass `Function` and what methods can be defined.

Examples

Undefined Functions

To create an undefined function, pass a string of the function name to `Function`.

```
>>> from sympy import Function, Symbol
>>> x = Symbol('x')
>>> f = Function('f')
>>> g = Function('g')(x)
>>> f
f
>>> f(x)
f(x)
>>> g
g(x)
>>> f(x).diff(x)
Derivative(f(x), x)
>>> g.diff(x)
Derivative(g(x), x)
```

Assumptions can be passed to `Function` the same as with a *Symbol* (page 976). Alternatively, you can use a `Symbol` with assumptions for the function name and the function will inherit the name and assumptions associated with the `Symbol`:

```
>>> f_real = Function('f', real=True)
>>> f_real(x).is_real
True
>>> f_real_inherit = Function(Symbol('f', real=True))
>>> f_real_inherit(x).is_real
True
```

Note that assumptions on a function are unrelated to the assumptions on the variables it is called on. If you want to add a relationship, subclass `Function` and define custom assumptions handler methods. See the *Assumptions* (page 111) section of the *Writing Custom Functions* (page 102) guide for more details.

Custom Function Subclasses

The *Writing Custom Functions* (page 102) guide has several *Complete Examples* (page 122) of how to subclass `Function` to create a custom function.

as_base_exp()
Returns the method as the 2-tuple (base, exponent).

fdiff(*argindex=1*)
Returns the first derivative of the function.

classmethod is_singular(*a*)
Tests whether the argument is an essential singularity or a branch point, or the functions is non-holomorphic.

Contents

Elementary

This module implements elementary functions such as trigonometric, hyperbolic, and sqrt, as well as functions like `Abs`, `Max`, `Min` etc.

sympy.functions.elementary.complexes

re

class sympy.functions.elementary.complexes.**re**(*arg*)
Returns real part of expression. This function performs only elementary analysis and so it will fail to decompose properly more complicated expressions. If completely simplified result is needed then use `Basic.as_real_imag()` or perform complex expansion on instance of this function.

Parameters
arg : Expr
Real or complex expression.

Returns
expr : Expr
Real part of expression.

Examples

```
>>> from sympy import re, im, I, E, symbols
>>> x, y = symbols('x y', real=True)
>>> re(2*E)
2*E
>>> re(2*I + 17)
17
>>> re(2*I)
0
>>> re(im(x) + x*I + 2)
2
>>> re(5 + I + 2)
7
```

See also:

im (page 383)

as_real_imag(*deep=True*, ***hints*)

Returns the real number with a zero imaginary part.

im

class sympy.functions.elementary.complexes.**im**(*arg*)

Returns imaginary part of expression. This function performs only elementary analysis and so it will fail to decompose properly more complicated expressions. If completely simplified result is needed then use `Basic.as_real_imag()` or perform complex expansion on instance of this function.

Parameters

arg : Expr

Real or complex expression.

Returns

expr : Expr

Imaginary part of expression.

Examples

```
>>> from sympy import re, im, E, I
>>> from sympy.abc import x, y
>>> im(2*E)
0
>>> im(2*I + 17)
2
>>> im(x*I)
re(x)
>>> im(re(x) + y)
im(y)
>>> im(2 + 3*I)
3
```

See also:

re (page 382)

as_real_imag(*deep=True*, ***hints*)

Return the imaginary part with a zero real part.

sign

class sympy.functions.elementary.complexes.**sign**(*arg*)

Returns the complex sign of an expression:

Parameters
arg : Expr

Real or imaginary expression.

Returns
expr : Expr

Complex sign of expression.

Explanation

If the expression is real the sign will be:

- 1 if expression is positive
- 0 if expression is equal to zero
- -1 if expression is negative

If the expression is imaginary the sign will be:

- I if im(expression) is positive
- $-I$ if im(expression) is negative

Otherwise an unevaluated expression will be returned. When evaluated, the result (in general) will be `cos(arg(expr)) + I*sin(arg(expr))`.

Examples

```
>>> from sympy import sign, I
```

```
>>> sign(-1)
-1
>>> sign(0)
0
>>> sign(-3*I)
-I
>>> sign(1 + I)
sign(1 + I)
>>> _.evalf()
0.707106781186548 + 0.707106781186548*I
```

See also:

Abs (page 385), *conjugate* (page 386)

Abs

class sympy.functions.elementary.complexes.**Abs**(*arg*)

Return the absolute value of the argument.

Parameters

arg : Expr

Real or complex expression.

Returns

expr : Expr

Absolute value returned can be an expression or integer depending on input arg.

Explanation

This is an extension of the built-in function `abs()` to accept symbolic values. If you pass a SymPy expression to the built-in `abs()`, it will pass it automatically to `Abs()`.

Examples

```
>>> from sympy import Abs, Symbol, S, I
>>> Abs(-1)
1
>>> x = Symbol('x', real=True)
>>> Abs(-x)
Abs(x)
>>> Abs(x**2)
x**2
>>> abs(-x) # The Python built-in
Abs(x)
>>> Abs(3*x + 2*I)
sqrt(9*x**2 + 4)
>>> Abs(8*I)
8
```

Note that the Python built-in will return either an Expr or int depending on the argument:

```
>>> type(abs(-1))
<... 'int'>
>>> type(abs(S.NegativeOne))
<class 'sympy.core.numbers.One'>
```

Abs will always return a SymPy object.

See also:

sign (page 384), *conjugate* (page 386)

fdiff(*argindex=1*)

Get the first derivative of the argument to Abs().

arg

class sympy.functions.elementary.complexes.**arg**(*arg*)

Returns the argument (in radians) of a complex number. The argument is evaluated in consistent convention with `atan2` where the branch-cut is taken along the negative real axis and `arg(z)` is in the interval $(-\pi, \pi]$. For a positive number, the argument is always 0; the argument of a negative number is π; and the argument of 0 is undefined and returns `nan`. So the `arg` function will never nest greater than 3 levels since at the 4th application, the result must be nan; for a real number, nan is returned on the 3rd application.

Parameters
arg : Expr

Real or complex expression.

Returns
value : Expr

Returns arc tangent of arg measured in radians.

Examples

```
>>> from sympy import arg, I, sqrt, Dummy
>>> from sympy.abc import x
>>> arg(2.0)
0
>>> arg(I)
pi/2
>>> arg(sqrt(2) + I*sqrt(2))
pi/4
>>> arg(sqrt(3)/2 + I/2)
pi/6
>>> arg(4 + 3*I)
atan(3/4)
>>> arg(0.8 + 0.6*I)
0.643501108793284
>>> arg(arg(arg(arg(x))))
nan
>>> real = Dummy(real=True)
>>> arg(arg(arg(real)))
nan
```

conjugate

class sympy.functions.elementary.complexes.**conjugate**(*arg*)

Returns the *complex conjugate* [R238] of an argument. In mathematics, the complex conjugate of a complex number is given by changing the sign of the imaginary part.

Thus, the conjugate of the complex number $a + ib$ (where a and b are real numbers) is $a - ib$

Parameters
arg : Expr

Real or complex expression.

Returns

arg : Expr

Complex conjugate of arg as real, imaginary or mixed expression.

Examples

```
>>> from sympy import conjugate, I
>>> conjugate(2)
2
>>> conjugate(I)
-I
>>> conjugate(3 + 2*I)
3 - 2*I
>>> conjugate(5 - I)
5 + I
```

See also:

sign (page 384), *Abs* (page 385)

References

[R238]

polar_lift

class sympy.functions.elementary.complexes.**polar_lift**(*arg*)

Lift argument to the Riemann surface of the logarithm, using the standard branch.

Parameters

arg : Expr

Real or complex expression.

Examples

```
>>> from sympy import Symbol, polar_lift, I
>>> p = Symbol('p', polar=True)
>>> x = Symbol('x')
>>> polar_lift(4)
4*exp_polar(0)
>>> polar_lift(-4)
4*exp_polar(I*pi)
>>> polar_lift(-I)
exp_polar(-I*pi/2)
>>> polar_lift(I + 2)
polar_lift(2 + I)
```

```
>>> polar_lift(4*x)
4*polar_lift(x)
>>> polar_lift(4*p)
4*p
```

See also:

sympy.functions.elementary.exponential.exp_polar (page 414), *periodic_argument* (page 388)

periodic_argument

class sympy.functions.elementary.complexes.**periodic_argument**(*ar*, *period*)

Represent the argument on a quotient of the Riemann surface of the logarithm. That is, given a period P, always return a value in $(-P/2, P/2]$, by using $\exp(PI) = 1$.

Parameters

ar : Expr

A polar number.

period : Expr

The period P.

Examples

```
>>> from sympy import exp_polar, periodic_argument
>>> from sympy import I, pi
>>> periodic_argument(exp_polar(10*I*pi), 2*pi)
0
>>> periodic_argument(exp_polar(5*I*pi), 4*pi)
pi
>>> from sympy import exp_polar, periodic_argument
>>> from sympy import I, pi
>>> periodic_argument(exp_polar(5*I*pi), 2*pi)
pi
>>> periodic_argument(exp_polar(5*I*pi), 3*pi)
-pi
>>> periodic_argument(exp_polar(5*I*pi), pi)
0
```

See also:

sympy.functions.elementary.exponential.exp_polar (page 414)

***polar_lift* (page 387)**
Lift argument to the Riemann surface of the logarithm

principal_branch (page 389)

principal_branch

class sympy.functions.elementary.complexes.**principal_branch**(*x*, *period*)

Represent a polar number reduced to its principal branch on a quotient of the Riemann surface of the logarithm.

Parameters

x : Expr

A polar number.

period : Expr

Positive real number or infinity.

Explanation

This is a function of two arguments. The first argument is a polar number z, and the second one a positive real number or infinity, p. The result is `z mod exp_polar(I*p)`.

Examples

```
>>> from sympy import exp_polar, principal_branch, oo, I, pi
>>> from sympy.abc import z
>>> principal_branch(z, oo)
z
>>> principal_branch(exp_polar(2*pi*I)*3, 2*pi)
3*exp_polar(0)
>>> principal_branch(exp_polar(2*pi*I)*3*z, 2*pi)
3*principal_branch(z, 2*pi)
```

See also:

sympy.functions.elementary.exponential.exp_polar (page 414)

***polar_lift* (page 387)**
Lift argument to the Riemann surface of the logarithm

periodic_argument (page 388)

sympy.functions.elementary.trigonometric

Trigonometric Functions

sin

class sympy.functions.elementary.trigonometric.**sin**(*arg*)

The sine function.

Returns the sine of x (measured in radians).

Explanation

This function will evaluate automatically in the case x/π is some rational number [R242]. For example, if x is a multiple of π, $\pi/2$, $\pi/3$, $\pi/4$, and $\pi/6$.

Examples

```
>>> from sympy import sin, pi
>>> from sympy.abc import x
>>> sin(x**2).diff(x)
2*x*cos(x**2)
>>> sin(1).diff(x)
0
>>> sin(pi)
0
>>> sin(pi/2)
1
>>> sin(pi/6)
1/2
>>> sin(pi/12)
-sqrt(2)/4 + sqrt(6)/4
```

See also:

csc (page 393), *cos* (page 390), *sec* (page 393), *tan* (page 391), *cot* (page 392), *asin* (page 395), *acsc* (page 399), *acos* (page 396), *asec* (page 398), *atan* (page 397), *acot* (page 397), *atan2* (page 400)

References

[R239], [R240], [R241], [R242]

cos

class sympy.functions.elementary.trigonometric.**cos**(*arg*)

The cosine function.

Returns the cosine of x (measured in radians).

Explanation

See *sin()* (page 389) for notes about automatic evaluation.

Examples

```
>>> from sympy import cos, pi
>>> from sympy.abc import x
>>> cos(x**2).diff(x)
-2*x*sin(x**2)
>>> cos(1).diff(x)
0
>>> cos(pi)
-1
>>> cos(pi/2)
0
>>> cos(2*pi/3)
-1/2
>>> cos(pi/12)
sqrt(2)/4 + sqrt(6)/4
```

See also:

sin (page 389), *csc* (page 393), *sec* (page 393), *tan* (page 391), *cot* (page 392), *asin* (page 395), *acsc* (page 399), *acos* (page 396), *asec* (page 398), *atan* (page 397), *acot* (page 397), *atan2* (page 400)

References

[R243], [R244], [R245]

tan

class sympy.functions.elementary.trigonometric.**tan**(*arg*)

The tangent function.

Returns the tangent of x (measured in radians).

Explanation

See *sin* (page 389) for notes about automatic evaluation.

Examples

```
>>> from sympy import tan, pi
>>> from sympy.abc import x
>>> tan(x**2).diff(x)
2*x*(tan(x**2)**2 + 1)
>>> tan(1).diff(x)
0
>>> tan(pi/8).expand()
-1 + sqrt(2)
```

See also:

sin (page 389), *csc* (page 393), *cos* (page 390), *sec* (page 393), *cot* (page 392), *asin* (page 395), *acsc* (page 399), *acos* (page 396), *asec* (page 398), *atan* (page 397), *acot* (page 397), *atan2* (page 400)

References

[R246], [R247], [R248]

inverse(*argindex=1*)

Returns the inverse of this function.

cot

class sympy.functions.elementary.trigonometric.**cot**(*arg*)

The cotangent function.

Returns the cotangent of x (measured in radians).

Explanation

See *sin* (page 389) for notes about automatic evaluation.

Examples

```
>>> from sympy import cot, pi
>>> from sympy.abc import x
>>> cot(x**2).diff(x)
2*x*(-cot(x**2)**2 - 1)
>>> cot(1).diff(x)
0
>>> cot(pi/12)
sqrt(3) + 2
```

See also:

sin (page 389), *csc* (page 393), *cos* (page 390), *sec* (page 393), *tan* (page 391), *asin* (page 395), *acsc* (page 399), *acos* (page 396), *asec* (page 398), *atan* (page 397), *acot* (page 397), *atan2* (page 400)

References

[R249], [R250], [R251]

inverse(*argindex=1*)

Returns the inverse of this function.

sec

class sympy.functions.elementary.trigonometric.**sec**(*arg*)

The secant function.

Returns the secant of x (measured in radians).

Explanation

See *sin* (page 389) for notes about automatic evaluation.

Examples

```
>>> from sympy import sec
>>> from sympy.abc import x
>>> sec(x**2).diff(x)
2*x*tan(x**2)*sec(x**2)
>>> sec(1).diff(x)
0
```


See also:

sin (page 389), *csc* (page 393), *cos* (page 390), *tan* (page 391), *cot* (page 392), *asin* (page 395), *acsc* (page 399), *acos* (page 396), *asec* (page 398), *atan* (page 397), *acot* (page 397), *atan2* (page 400)

References

[R252], [R253], [R254]

csc

class sympy.functions.elementary.trigonometric.**csc**(*arg*)

The cosecant function.

Returns the cosecant of x (measured in radians).

Explanation

See *sin()* (page 389) for notes about automatic evaluation.

Examples

```
>>> from sympy import csc
>>> from sympy.abc import x
>>> csc(x**2).diff(x)
-2*x*cot(x**2)*csc(x**2)
>>> csc(1).diff(x)
0
```

See also:

sin (page 389), *cos* (page 390), *sec* (page 393), *tan* (page 391), *cot* (page 392), *asin* (page 395), *acsc* (page 399), *acos* (page 396), *asec* (page 398), *atan* (page 397), *acot* (page 397), *atan2* (page 400)

References

[R255], [R256], [R257]

sinc

class sympy.functions.elementary.trigonometric.**sinc**(*arg*)

Represents an unnormalized sinc function:

$$\operatorname{sinc}(x) = \begin{cases} \frac{\sin x}{x} & x \neq 0 \\ 1 & x = 0 \end{cases}$$

Examples

```
>>> from sympy import sinc, oo, jn
>>> from sympy.abc import x
>>> sinc(x)
sinc(x)
```

- Automated Evaluation

```
>>> sinc(0)
1
>>> sinc(oo)
0
```

- Differentiation

```
>>> sinc(x).diff()
cos(x)/x - sin(x)/x**2
```

- Series Expansion

```
>>> sinc(x).series()
1 - x**2/6 + x**4/120 + O(x**6)
```

- As zero'th order spherical Bessel Function

```
>>> sinc(x).rewrite(jn)
jn(0, x)
```

See also:

sin (page 389)

References

[R258]

Trigonometric Inverses

asin

class sympy.functions.elementary.trigonometric.**asin**(*arg*)

The inverse sine function.

Returns the arcsine of x in radians.

Explanation

`asin(x)` will evaluate automatically in the cases $x \in \{\infty, -\infty, 0, 1, -1\}$ and for some instances when the result is a rational multiple of π (see the `eval` class method).

A purely imaginary argument will lead to an asinh expression.

Examples

```
>>> from sympy import asin, oo
>>> asin(1)
pi/2
>>> asin(-1)
-pi/2
>>> asin(-oo)
oo*I
>>> asin(oo)
-oo*I
```

See also:

sin (page 389), *csc* (page 393), *cos* (page 390), *sec* (page 393), *tan* (page 391), *cot* (page 392), *acsc* (page 399), *acos* (page 396), *asec* (page 398), *atan* (page 397), *acot* (page 397), *atan2* (page 400)

References

[R259], [R260], [R261]

inverse(*argindex=1*)

Returns the inverse of this function.

acos

class sympy.functions.elementary.trigonometric.**acos**(*arg*)

The inverse cosine function.

Returns the arc cosine of x (measured in radians).

Examples

```
>>> from sympy import acos, oo
>>> acos(1)
0
>>> acos(0)
pi/2
>>> acos(oo)
oo*I
```

See also:

sin (page 389), *csc* (page 393), *cos* (page 390), *sec* (page 393), *tan* (page 391), *cot* (page 392), *asin* (page 395), *acsc* (page 399), *asec* (page 398), *atan* (page 397), *acot* (page 397), *atan2* (page 400)

References

[R262], [R263], [R264]

inverse(*argindex=1*)

Returns the inverse of this function.