References

- "Frobenius Pseudoprimes", Jon Grantham, 2000. http://www.ams.org/journals/mcom/2001-70-234/S0025-5718-00-01197-2/
- OEIS A217719: Extra Strong Lucas Pseudoprimes https://oeis.org/A217719
- https://en.wikipedia.org/wiki/Lucas_pseudoprime

`sympy.ntheory.primetest.`**`isprime`**`(`*`n`*`)`

Test if n is a prime number (True) or not (False). For n < 2^64 the answer is definitive; larger n values have a small probability of actually being pseudoprimes.

Negative numbers (e.g. -2) are not considered prime.

The first step is looking for trivial factors, which if found enables a quick return. Next, if the sieve is large enough, use bisection search on the sieve. For small numbers, a set of deterministic Miller-Rabin tests are performed with bases that are known to have no counterexamples in their range. Finally if the number is larger than 2^64, a strong BPSW test is performed. While this is a probable prime test and we believe counterexamples exist, there are no known counterexamples.

Examples

```
>>> from sympy.ntheory import isprime
>>> isprime(13)
True
>>> isprime(13.0)  # limited precision
False
>>> isprime(15)
False
```

Notes

This routine is intended only for integer input, not numerical expressions which may represent numbers. Floats are also rejected as input because they represent numbers of limited precision. While it is tempting to permit 7.0 to represent an integer there are errors that may "pass silently" if this is allowed:

```
>>> from sympy import Float, S
>>> int(1e3) == 1e3 == 10**3
True
>>> int(1e23) == 1e23
True
>>> int(1e23) == 10**23
False
```

```
>>> near_int = 1 + S(1)/10**19
>>> near_int == int(near_int)
False
>>> n = Float(near_int, 10)  # truncated by precision
>>> n == int(n)
```

(continues on next page)

(continued from previous page)

```
True
>>> n = Float(near_int, 20)
>>> n == int(n)
False
```

See also:

sympy.ntheory.generate.primerange **(page 1481)**
Generates all primes in a given range

sympy.ntheory.generate.primepi **(page 1479)**
Return the number of primes less than or equal to n

sympy.ntheory.generate.prime **(page 1479)**
Return the nth prime

References

- https://en.wikipedia.org/wiki/Strong_pseudoprime
- "Lucas Pseudoprimes", Baillie and Wagstaff, 1980. http://mpqs.free.fr/LucasPseudoprimes.pdf
- https://en.wikipedia.org/wiki/Baillie-PSW_primality_test

sympy.ntheory.primetest.**is_gaussian_prime**(*num*)

Test if num is a Gaussian prime number.

References

[R633]

sympy.ntheory.residue_ntheory.**n_order**(*a*, *n*)

Returns the order of a modulo n.

The order of a modulo n is the smallest integer k such that a**k leaves a remainder of 1 with n.

Examples

```
>>> from sympy.ntheory import n_order
>>> n_order(3, 7)
6
>>> n_order(4, 7)
3
```

sympy.ntheory.residue_ntheory.**is_primitive_root**(*a*, *p*)

Returns True if a is a primitive root of p

a is said to be the primitive root of p if gcd(a, p) == 1 and totient(p) is the smallest positive number s.t.

a**totient(p) cong 1 mod(p)

Examples

```
>>> from sympy.ntheory import is_primitive_root, n_order, totient
>>> is_primitive_root(3, 10)
True
>>> is_primitive_root(9, 10)
False
>>> n_order(3, 10) == totient(10)
True
>>> n_order(9, 10) == totient(10)
False
```

`sympy.ntheory.residue_ntheory.`**`primitive_root`**(*p*)

Returns the smallest primitive root or None

Parameters

p : positive integer

Examples

```
>>> from sympy.ntheory.residue_ntheory import primitive_root
>>> primitive_root(19)
2
```

References

[R634], [R635]

`sympy.ntheory.residue_ntheory.`**`sqrt_mod`**(*a, p, all_roots=False*)

Find a root of `x**2 = a mod p`

Parameters

a : integer

p : positive integer

all_roots : if True the list of roots is returned or None

Notes

If there is no root it is returned None; else the returned root is less or equal to `p // 2`; in general is not the smallest one. It is returned `p // 2` only if it is the only root.

Use `all_roots` only when it is expected that all the roots fit in memory; otherwise use `sqrt_mod_iter`.

Examples

```
>>> from sympy.ntheory import sqrt_mod
>>> sqrt_mod(11, 43)
21
>>> sqrt_mod(17, 32, True)
[7, 9, 23, 25]
```

`sympy.ntheory.residue_ntheory.`**`sqrt_mod_iter`**(*a, p, domain=<class 'int'>*)

Iterate over solutions to `x**2 = a mod p`

Parameters

a : integer

p : positive integer

domain : integer domain, `int`, `ZZ` or `Integer`

Examples

```
>>> from sympy.ntheory.residue_ntheory import sqrt_mod_iter
>>> list(sqrt_mod_iter(11, 43))
[21, 22]
```

`sympy.ntheory.residue_ntheory.`**`quadratic_residues`**(*p*) → list[int]

Returns the list of quadratic residues.

Examples

```
>>> from sympy.ntheory.residue_ntheory import quadratic_residues
>>> quadratic_residues(7)
[0, 1, 2, 4]
```

`sympy.ntheory.residue_ntheory.`**`nthroot_mod`**(*a, n, p, all_roots=False*)

Find the solutions to `x**n = a mod p`

Parameters

a : integer

n : positive integer

p : positive integer

all_roots : if False returns the smallest root, else the list of roots

Examples

```
>>> from sympy.ntheory.residue_ntheory import nthroot_mod
>>> nthroot_mod(11, 4, 19)
8
>>> nthroot_mod(11, 4, 19, True)
[8, 11]
>>> nthroot_mod(68, 3, 109)
23
```

sympy.ntheory.residue_ntheory.**is_nthpow_residue**(*a, n, m*)

Returns True if `x**n == a (mod m)` has solutions.

References

[R636]

sympy.ntheory.residue_ntheory.**is_quad_residue**(*a, p*)

Returns True if a (mod p) is in the set of squares mod p, i.e a % p in set([i**2 % p for i in range(p)]). If p is an odd prime, an iterative method is used to make the determination:

```
>>> from sympy.ntheory import is_quad_residue
>>> sorted(set([i**2 % 7 for i in range(7)]))
[0, 1, 2, 4]
>>> [j for j in range(7) if is_quad_residue(j, 7)]
[0, 1, 2, 4]
```

See also:

legendre_symbol (page 1521), *jacobi_symbol* (page 1522)

sympy.ntheory.residue_ntheory.**legendre_symbol**(*a, p*)

Returns the Legendre symbol (a/p).

For an integer a and an odd prime p, the Legendre symbol is defined as

$$\left(\frac{a}{p}\right) = \begin{cases} 0 & \text{if } p \text{ divides } a \\ 1 & \text{if } a \text{ is a quadratic residue modulo } p \\ -1 & \text{if } a \text{ is a quadratic nonresidue modulo } p \end{cases}$$

Parameters

a : integer

p : odd prime

Examples

```
>>> from sympy.ntheory import legendre_symbol
>>> [legendre_symbol(i, 7) for i in range(7)]
[0, 1, 1, -1, 1, -1, -1]
>>> sorted(set([i**2 % 7 for i in range(7)]))
[0, 1, 2, 4]
```

See also:

is_quad_residue (page 1521), *jacobi_symbol* (page 1522)

sympy.ntheory.residue_ntheory.**jacobi_symbol**(*m*, *n*)

Returns the Jacobi symbol (m/n).

For any integer m and any positive odd integer n the Jacobi symbol is defined as the product of the Legendre symbols corresponding to the prime factors of n:

$$\left(\frac{m}{n}\right) = \left(\frac{m}{p^1}\right)^{\alpha_1} \left(\frac{m}{p^2}\right)^{\alpha_2} ... \left(\frac{m}{p^k}\right)^{\alpha_k} \text{ where } n = p_1^{\alpha_1} p_2^{\alpha_2} ... p_k^{\alpha_k}$$

Like the Legendre symbol, if the Jacobi symbol $\left(\frac{m}{n}\right) = -1$ then m is a quadratic nonresidue modulo n.

But, unlike the Legendre symbol, if the Jacobi symbol $\left(\frac{m}{n}\right) = 1$ then m may or may not be a quadratic residue modulo n.

Parameters

m : integer

n : odd positive integer

Examples

```
>>> from sympy.ntheory import jacobi_symbol, legendre_symbol
>>> from sympy import S
>>> jacobi_symbol(45, 77)
-1
>>> jacobi_symbol(60, 121)
1
```

The relationship between the jacobi_symbol and legendre_symbol can be demonstrated as follows:

```
>>> L = legendre_symbol
>>> S(45).factors()
{3: 2, 5: 1}
>>> jacobi_symbol(7, 45) == L(7, 3)**2 * L(7, 5)**1
True
```

See also:

is_quad_residue (page 1521), *legendre_symbol* (page 1521) `sympy.ntheory.residue_ntheory.`**`discrete_log`**(*n*, *a*, *b*, *order=None*, *prime_order=None*)

Compute the discrete logarithm of a to the base b modulo n.

This is a recursive function to reduce the discrete logarithm problem in cyclic groups of composite order to the problem in cyclic groups of prime order.

It employs different algorithms depending on the problem (subgroup order size, prime order or not):

- Trial multiplication
- Baby-step giant-step
- Pollard's Rho
- Pohlig-Hellman

Examples

```
>>> from sympy.ntheory import discrete_log
>>> discrete_log(41, 15, 7)
3
```

References

[R637], [R638]

`sympy.ntheory.continued_fraction.`**`continued_fraction`**(*a*) → list

Return the continued fraction representation of a Rational or quadratic irrational.

Examples

```
>>> from sympy.ntheory.continued_fraction import continued_fraction
>>> from sympy import sqrt
>>> continued_fraction((1 + 2*sqrt(3))/5)
[0, 1, [8, 3, 34, 3]]
```

See also:

continued_fraction_periodic (page 1525), *continued_fraction_reduce* (page 1526), *continued_fraction_convergents* (page 1523)

`sympy.ntheory.continued_fraction.`**`continued_fraction_convergents`**(*cf*)

Return an iterator over the convergents of a continued fraction (cf).

The parameter should be an iterable returning successive partial quotients of the continued fraction, such as might be returned by continued_fraction_iterator. In computing the convergents, the continued fraction need not be strictly in canonical form (all integers, all but the first positive). Rational and negative elements may be present in the expansion.

Examples

```
>>> from sympy.core import pi
>>> from sympy import S
>>> from sympy.ntheory.continued_fraction import          continued_
↪fraction_convergents, continued_fraction_iterator
```

```
>>> list(continued_fraction_convergents([0, 2, 1, 2]))
[0, 1/2, 1/3, 3/8]
```

```
>>> list(continued_fraction_convergents([1, S('1/2'), -7, S('1/4')]))
[1, 3, 19/5, 7]
```

```
>>> it = continued_fraction_convergents(continued_fraction_iterator(pi))
>>> for n in range(7):
...     print(next(it))
3
22/7
333/106
355/113
103993/33102
104348/33215
208341/66317
```

See also:

continued_fraction_iterator (page 1524)

sympy.ntheory.continued_fraction.**continued_fraction_iterator**(*x*)

Return continued fraction expansion of x as iterator.

Examples

```
>>> from sympy import Rational, pi
>>> from sympy.ntheory.continued_fraction import continued_fraction_
↪iterator
```

```
>>> list(continued_fraction_iterator(Rational(3, 8)))
[0, 2, 1, 2]
>>> list(continued_fraction_iterator(Rational(-3, 8)))
[-1, 1, 1, 1, 2]
```

```
>>> for i, v in enumerate(continued_fraction_iterator(pi)):
...     if i > 7:
...         break
...     print(v)
3
7
15
1
292
```

(continues on next page)

(continued from previous page)

```
1
1
1
```

References

[R639]

`sympy.ntheory.continued_fraction.`**`continued_fraction_periodic`**(*p*, *q*, *d=0*, *s=1*) → list

Find the periodic continued fraction expansion of a quadratic irrational.

Compute the continued fraction expansion of a rational or a quadratic irrational number, i.e. $\frac{p+s\sqrt{d}}{q}$, where p, $q \ne 0$ and $d \geq 0$ are integers.

Returns the continued fraction representation (canonical form) as a list of integers, optionally ending (for quadratic irrationals) with list of integers representing the repeating digits.

Parameters

p : int

the rational part of the number's numerator

q : int

the denominator of the number

d : int, optional

the irrational part (discriminator) of the number's numerator

s : int, optional

the coefficient of the irrational part

Examples

```
>>> from sympy.ntheory.continued_fraction import continued_fraction_
↪periodic
>>> continued_fraction_periodic(3, 2, 7)
[2, [1, 4, 1, 1]]
```

Golden ratio has the simplest continued fraction expansion:

```
>>> continued_fraction_periodic(1, 2, 5)
[[1]]
```

If the discriminator is zero or a perfect square then the number will be a rational number:

```
>>> continued_fraction_periodic(4, 3, 0)
[1, 3]
>>> continued_fraction_periodic(4, 3, 49)
[3, 1, 2]
```

See also:

continued_fraction_iterator (page 1524), *continued_fraction_reduce* (page 1526)

References

[R640], [R641]

sympy.ntheory.continued_fraction.**continued_fraction_reduce**(*cf*)

Reduce a continued fraction to a rational or quadratic irrational.

Compute the rational or quadratic irrational number from its terminating or periodic continued fraction expansion. The continued fraction expansion (cf) should be supplied as a terminating iterator supplying the terms of the expansion. For terminating continued fractions, this is equivalent to `list(continued_fraction_convergents(cf))[-1]`, only a little more efficient. If the expansion has a repeating part, a list of the repeating terms should be returned as the last element from the iterator. This is the format returned by continued_fraction_periodic.

For quadratic irrationals, returns the largest solution found, which is generally the one sought, if the fraction is in canonical form (all terms positive except possibly the first).

Examples

```
>>> from sympy.ntheory.continued_fraction import continued_fraction_
↪reduce
>>> continued_fraction_reduce([1, 2, 3, 4, 5])
225/157
>>> continued_fraction_reduce([-2, 1, 9, 7, 1, 2])
-256/233
>>> continued_fraction_reduce([2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8]).n(10)
2.718281835
>>> continued_fraction_reduce([1, 4, 2, [3, 1]])
(sqrt(21) + 287)/238
>>> continued_fraction_reduce([[1]])
(1 + sqrt(5))/2
>>> from sympy.ntheory.continued_fraction import continued_fraction_
↪periodic
>>> continued_fraction_reduce(continued_fraction_periodic(8, 5, 13))
(sqrt(13) + 8)/5
```

See also:

continued_fraction_periodic (page 1525)

sympy.ntheory.digits.**count_digits**(*n*, *b=10*)

Return a dictionary whose keys are the digits of `n` in the given base, `b`, with keys indicating the digits appearing in the number and values indicating how many times that digit appeared.

Examples

```
>>> from sympy.ntheory import count_digits
```

```
>>> count_digits(1111339)
{1: 4, 3: 2, 9: 1}
```

The digits returned are always represented in base-10 but the number itself can be entered in any format that is understood by Python; the base of the number can also be given if it is different than 10:

```
>>> n = 0xFA; n
250
>>> count_digits(_)
{0: 1, 2: 1, 5: 1}
>>> count_digits(n, 16)
{10: 1, 15: 1}
```

The default dictionary will return a 0 for any digit that did not appear in the number. For example, which digits appear 7 times in 77!:

```
>>> from sympy import factorial
>>> c77 = count_digits(factorial(77))
>>> [i for i in range(10) if c77[i] == 7]
[1, 3, 7, 9]
```

`sympy.ntheory.digits.`**`digits`**`(`*n, b=10, digits=None*`)`

Return a list of the digits of `n` in base `b`. The first element in the list is `b` (or `-b` if `n` is negative).

Parameters

n: integer

The number whose digits are returned.

b: integer

The base in which digits are computed.

digits: integer (or None for all digits)

The number of digits to be returned (padded with zeros, if necessary).

Examples

```
>>> from sympy.ntheory.digits import digits
>>> digits(35)
[10, 3, 5]
```

If the number is negative, the negative sign will be placed on the base (which is the first element in the returned list):

```
>>> digits(-35)
[-10, 3, 5]
```

Bases other than 10 (and greater than 1) can be selected with b:

```
>>> digits(27, b=2)
[2, 1, 1, 0, 1, 1]
```

Use the digits keyword if a certain number of digits is desired:

```
>>> digits(35, digits=4)
[10, 0, 0, 3, 5]
```

sympy.ntheory.digits.**is_palindromic**(*n*, *b=10*)

return True if n is the same when read from left to right or right to left in the given base, b.

Examples

```
>>> from sympy.ntheory import is_palindromic
```

```
>>> all(is_palindromic(i) for i in (-11, 1, 22, 121))
True
```

The second argument allows you to test numbers in other bases. For example, 88 is palindromic in base-10 but not in base-8:

```
>>> is_palindromic(88, 8)
False
```

On the other hand, a number can be palindromic in base-8 but not in base-10:

```
>>> 0o121, is_palindromic(0o121)
(81, False)
```

Or it might be palindromic in both bases:

```
>>> oct(121), is_palindromic(121, 8) and is_palindromic(121)
('0o171', True)
```

class sympy.ntheory.**mobius**(*n*)

Mobius function maps natural number to {-1, 0, 1}

It is defined as follows:

1) 1 if $n = 1$.
2) 0 if n has a squared prime factor.
3) $(-1)^k$ if n is a square-free positive integer with k number of prime factors.

It is an important multiplicative function in number theory and combinatorics. It has applications in mathematical series, algebraic number theory and also physics (Fermion operator has very concrete realization with Mobius Function model).

Parameters

n : positive integer

Examples

```
>>> from sympy.ntheory import mobius
>>> mobius(13*7)
1
>>> mobius(1)
1
>>> mobius(13*7*5)
-1
>>> mobius(13**2)
0
```

References

[R642], [R643]

`sympy.ntheory.egyptian_fraction.`**`egyptian_fraction`**(*r*, *algorithm='Greedy'*)

Return the list of denominators of an Egyptian fraction expansion [R644] of the said rational r.

Parameters

r : Rational or (p, q)

a positive rational number, `p/q`.

algorithm : { "Greedy", "Graham Jewett", "Takenouchi", "Golomb" }, optional

Denotes the algorithm to be used (the default is "Greedy").

Examples

```
>>> from sympy import Rational
>>> from sympy.ntheory.egyptian_fraction import egyptian_fraction
>>> egyptian_fraction(Rational(3, 7))
[3, 11, 231]
>>> egyptian_fraction((3, 7), "Graham Jewett")
[7, 8, 9, 56, 57, 72, 3192]
>>> egyptian_fraction((3, 7), "Takenouchi")
[4, 7, 28]
>>> egyptian_fraction((3, 7), "Golomb")
[3, 15, 35]
>>> egyptian_fraction((11, 5), "Golomb")
[1, 2, 3, 4, 9, 234, 1118, 2580]
```

Notes

Currently the following algorithms are supported:

1) Greedy Algorithm

   Also called the Fibonacci-Sylvester algorithm [R645]. At each step, extract the largest unit fraction less than the target and replace the target with the remainder.

   It has some distinct properties:

   a) Given p/q in lowest terms, generates an expansion of maximum length p. Even as the numerators get large, the number of terms is seldom more than a handful.

   b) Uses minimal memory.

   c) The terms can blow up (standard examples of this are 5/121 and 31/311). The denominator is at most squared at each step (doubly-exponential growth) and typically exhibits singly-exponential growth.

2) Graham Jewett Algorithm

   The algorithm suggested by the result of Graham and Jewett. Note that this has a tendency to blow up: the length of the resulting expansion is always `2**(x/gcd(x, y)) - 1`. See [R646].

3) Takenouchi Algorithm

   The algorithm suggested by Takenouchi (1921). Differs from the Graham-Jewett algorithm only in the handling of duplicates. See [R646].

4) Golomb's Algorithm

   A method given by Golumb (1962), using modular arithmetic and inverses. It yields the same results as a method using continued fractions proposed by Bleicher (1972). See [R647].

If the given rational is greater than or equal to 1, a greedy algorithm of summing the harmonic sequence 1/1 + 1/2 + 1/3 + … is used, taking all the unit fractions of this sequence until adding one more would be greater than the given number. This list of denominators is prefixed to the result from the requested algorithm used on the remainder. For example, if r is 8/3, using the Greedy algorithm, we get [1, 2, 3, 4, 5, 6, 7, 14, 420], where the beginning of the sequence, [1, 2, 3, 4, 5, 6, 7] is part of the harmonic sequence summing to 363/140, leaving a remainder of 31/420, which yields [14, 420] by the Greedy algorithm. The result of egyptian_fraction(Rational(8, 3), "Golomb") is [1, 2, 3, 4, 5, 6, 7, 14, 574, 2788, 6460, 11590, 33062, 113820], and so on.

See also:

sympy.core.numbers.Rational (page 985)

References

[R644], [R645], [R646], [R647]

sympy.ntheory.bbp_pi.**pi_hex_digits**(*n*, *prec=14*)

Returns a string containing prec (default 14) digits starting at the nth digit of pi in hex. Counting of digits starts at 0 and the decimal is not counted, so for n = 0 the returned value starts with 3; n = 1 corresponds to the first digit past the decimal point (which in hex is 2).

Examples

```
>>> from sympy.ntheory.bbp_pi import pi_hex_digits
>>> pi_hex_digits(0)
'3243f6a8885a30'
>>> pi_hex_digits(0, 3)
'324'
```

References

[R648]

ECM function

The ecm function is a subexponential factoring algorithm capable of factoring numbers of around ~35 digits comfortably within few seconds. The time complexity of ecm is dependent on the smallest proper factor of the number. So even if the number is really large but its factors are comparatively smaller then ecm can easily factor them. For example we take N with 15 digit factors 15154262241479, 15423094826093, 799333555511111, 809709509409109, 888888877777777, 914148152112161. Now N is a 87 digit number. ECM takes under around 47s to factorise this.

sympy.ntheory.ecm.**ecm**(*n*, *B1=10000*, *B2=100000*, *max_curve=200*, *seed=1234*)

Performs factorization using Lenstra's Elliptic curve method.

This function repeatedly calls ecm_one_factor to compute the factors of n. First all the small factors are taken out using trial division. Then ecm_one_factor is used to compute one factor at a time.

Parameters

n : Number to be Factored

B1 : Stage 1 Bound

B2 : Stage 2 Bound

max_curve : Maximum number of curves generated

seed : Initialize pseudorandom generator

Examples

```
>>> from sympy.ntheory import ecm
>>> ecm(25645121643901801)
{5394769, 4753701529}
>>> ecm(9804659461513846513)
{4641991, 2112166839943}
```

Examples

```
>>> from sympy.ntheory import ecm
>>> ecm(7060005655815754299976961394452809, B1=100000, B2=1000000)
{6988699669998001, 1010203040506070809}
>>>␣
↪ecm(122921448543883967430908091422761898618349713604256384403202282756086473494959648313
↪ B1=100000, B2=1000000)
{15154262241479,
15423094826093,
799333555511111,
809709509409109,
888888877777777,
914148152112161}
```

QS function

The qs function is a subexponential factoring algorithm, the fastest factoring algorithm for numbers within 100 digits. The time complexity of qs is dependent on the size of the number so it is used if the number contains large factors. Due to this while factoring numbers first ecm is used to get smaller factors of around ~15 digits then qs is used to get larger factors.

For factoring $2709077133180915240135586837960864768806330782747$ which is a semi-prime number with two 25 digit factors. qs is able to factorize this in around 248s.

`sympy.ntheory.qs.`**`qs`**(*N, prime_bound, M, ERROR_TERM=25, seed=1234*)

Performs factorization using Self-Initializing Quadratic Sieve. In SIQS, let N be a number to be factored, and this N should not be a perfect power. If we find two integers such that `X**2 = Y**2 modN` and `X != +-Y modN`, then $gcd(X + Y, N)$ will reveal a proper factor of N. In order to find these integers X and Y we try to find relations of form t**2 = u modN where u is a product of small primes. If we have enough of these relations then we can form `(t1*t2...ti)**2 = u1*u2...ui modN` such that the right hand side is a square, thus we found a relation of `X**2 = Y**2 modN`.

Here, several optimizations are done like using muliple polynomials for sieving, fast changing between polynomials and using partial relations. The use of partial relations can speeds up the factoring by 2 times.

Parameters

N : Number to be Factored

prime_bound : upper bound for primes in the factor base

M : Sieve Interval

ERROR_TERM : Error term for checking smoothness

threshold : Extra smooth relations for factorization

seed : generate pseudo prime numbers

Examples

```
>>> from sympy.ntheory import qs
>>> qs(25645121643901801, 2000, 10000)
{5394769, 4753701529}
>>> qs(9804659461513846513, 2000, 10000)
{4641991, 2112166839943}
```

References

[R649], [R650]

Examples

```
>>> from sympy.ntheory import qs
>>> qs(5915587277*3267000013, 1000, 10000)
{3267000013, 5915587277}
```

5.8.6 Physics

A module that helps solving problems in physics.

Contents

Hydrogen Wavefunctions

sympy.physics.hydrogen.**E_nl**(*n*, *Z=1*)

Returns the energy of the state (n, l) in Hartree atomic units.

The energy does not depend on “l”.

Parameters

n : integer

Principal Quantum Number which is an integer with possible values as 1, 2, 3, 4,…

Z :

Atomic number (1 for Hydrogen, 2 for Helium, …)

Examples

```
>>> from sympy.physics.hydrogen import E_nl
>>> from sympy.abc import n, Z
>>> E_nl(n, Z)
-Z**2/(2*n**2)
>>> E_nl(1)
-1/2
>>> E_nl(2)
-1/8
>>> E_nl(3)
-1/18
>>> E_nl(3, 47)
-2209/18
```

sympy.physics.hydrogen.**E_nl_dirac**(*n*, *l*, *spin_up=True*, *Z=1*, *c=137.035999037000*)

Returns the relativistic energy of the state (n, l, spin) in Hartree atomic units.

The energy is calculated from the Dirac equation. The rest mass energy is *not* included.

Parameters

n : integer

Principal Quantum Number which is an integer with possible values as 1, 2, 3, 4,...

l : integer

`l` is the Angular Momentum Quantum Number with values ranging from 0 to `n-1`.

spin_up :

True if the electron spin is up (default), otherwise down

Z :

Atomic number (1 for Hydrogen, 2 for Helium, ...)

c :

Speed of light in atomic units. Default value is 137.035999037, taken from http://arxiv.org/abs/1012.3627

Examples

```
>>> from sympy.physics.hydrogen import E_nl_dirac
>>> E_nl_dirac(1, 0)
-0.500006656595360
```

```
>>> E_nl_dirac(2, 0)
-0.125002080189006
>>> E_nl_dirac(2, 1)
-0.125000416028342
>>> E_nl_dirac(2, 1, False)
-0.125002080189006
```

```
>>> E_nl_dirac(3, 0)
-0.0555562951740285
>>> E_nl_dirac(3, 1)
-0.0555558020932949
>>> E_nl_dirac(3, 1, False)
-0.0555562951740285
>>> E_nl_dirac(3, 2)
-0.0555556377366884
>>> E_nl_dirac(3, 2, False)
-0.0555558020932949
```

sympy.physics.hydrogen.**Psi_nlm**(*n*, *l*, *m*, *r*, *phi*, *theta*, *Z=1*)

Returns the Hydrogen wave function psi_{nlm}. It's the product of the radial wavefunction R_{nl} and the spherical harmonic Y_{l}^{m}.

Parameters

n : integer

Principal Quantum Number which is an integer with possible values as 1, 2, 3, 4,...

l : integer

`l` is the Angular Momentum Quantum Number with values ranging from 0 to `n-1`.

m : integer

`m` is the Magnetic Quantum Number with values ranging from `-l` to `l`.

r :

radial coordinate

phi :

azimuthal angle

theta :

polar angle

Z :

atomic number (1 for Hydrogen, 2 for Helium, ...)

Everything is in Hartree atomic units.

Examples

```
>>> from sympy.physics.hydrogen import Psi_nlm
>>> from sympy import Symbol
>>> r=Symbol("r", positive=True)
>>> phi=Symbol("phi", real=True)
>>> theta=Symbol("theta", real=True)
>>> Z=Symbol("Z", positive=True, integer=True, nonzero=True)
>>> Psi_nlm(1,0,0,r,phi,theta,Z)
```

(continues on next page)

(continued from previous page)

```
Z**(3/2)*exp(-Z*r)/sqrt(pi)
>>> Psi_nlm(2,1,1,r,phi,theta,Z)
-Z**(5/2)*r*exp(I*phi)*exp(-Z*r/2)*sin(theta)/(8*sqrt(pi))
```

Integrating the absolute square of a hydrogen wavefunction psi_{nlm} over the whole space leads 1.

The normalization of the hydrogen wavefunctions Psi_nlm is:

```
>>> from sympy import integrate, conjugate, pi, oo, sin
>>> wf=Psi_nlm(2,1,1,r,phi,theta,Z)
>>> abs_sqrd=wf*conjugate(wf)
>>> jacobi=r**2*sin(theta)
>>> integrate(abs_sqrd*jacobi, (r,0,oo), (phi,0,2*pi), (theta,0,pi))
1
```

sympy.physics.hydrogen.**R_nl**(*n, l, r, Z=1*)

Returns the Hydrogen radial wavefunction R_{nl}.

Parameters

n : integer

Principal Quantum Number which is an integer with possible values as 1, 2, 3, 4,...

l : integer

`l` is the Angular Momentum Quantum Number with values ranging from 0 to `n-1`.

r :

Radial coordinate.

Z :

Atomic number (1 for Hydrogen, 2 for Helium, ...)

Everything is in Hartree atomic units.

Examples

```
>>> from sympy.physics.hydrogen import R_nl
>>> from sympy.abc import r, Z
>>> R_nl(1, 0, r, Z)
2*sqrt(Z**3)*exp(-Z*r)
>>> R_nl(2, 0, r, Z)
sqrt(2)*(-Z*r + 2)*sqrt(Z**3)*exp(-Z*r/2)/4
>>> R_nl(2, 1, r, Z)
sqrt(6)*Z*r*sqrt(Z**3)*exp(-Z*r/2)/12
```

For Hydrogen atom, you can just use the default value of Z=1:

```
>>> R_nl(1, 0, r)
2*exp(-r)
>>> R_nl(2, 0, r)
```

(continues on next page)

(continued from previous page)

```
sqrt(2)*(2 - r)*exp(-r/2)/4
>>> R_nl(3, 0, r)
2*sqrt(3)*(2*r**2/9 - 2*r + 3)*exp(-r/3)/27
```

For Silver atom, you would use Z=47:

```
>>> R_nl(1, 0, r, Z=47)
94*sqrt(47)*exp(-47*r)
>>> R_nl(2, 0, r, Z=47)
47*sqrt(94)*(2 - 47*r)*exp(-47*r/2)/4
>>> R_nl(3, 0, r, Z=47)
94*sqrt(141)*(4418*r**2/9 - 94*r + 3)*exp(-47*r/3)/27
```

The normalization of the radial wavefunction is:

```
>>> from sympy import integrate, oo
>>> integrate(R_nl(1, 0, r)**2 * r**2, (r, 0, oo))
1
>>> integrate(R_nl(2, 0, r)**2 * r**2, (r, 0, oo))
1
>>> integrate(R_nl(2, 1, r)**2 * r**2, (r, 0, oo))
1
```

It holds for any atomic number:

```
>>> integrate(R_nl(1, 0, r, Z=2)**2 * r**2, (r, 0, oo))
1
>>> integrate(R_nl(2, 0, r, Z=3)**2 * r**2, (r, 0, oo))
1
>>> integrate(R_nl(2, 1, r, Z=4)**2 * r**2, (r, 0, oo))
1
```

Matrices

Known matrices related to physics

`sympy.physics.matrices.`**`mdft`**`(`*n*`)`

Deprecated since version 1.9: Use DFT from sympy.matrices.expressions.fourier instead.

To get identical behavior to `mdft(n)`, use `DFT(n).as_explicit()`.

`sympy.physics.matrices.`**`mgamma`**`(`*mu, lower=False*`)`

Returns a Dirac gamma matrix γ^μ in the standard (Dirac) representation.

Explanation

If you want γ_μ, use gamma(mu, True).

We use a convention:

$\gamma^5 = i \cdot \gamma^0 \cdot \gamma^1 \cdot \gamma^2 \cdot \gamma^3$

$\gamma_5 = i \cdot \gamma_0 \cdot \gamma_1 \cdot \gamma_2 \cdot \gamma_3 = -\gamma^5$

Examples

```
>>> from sympy.physics.matrices import mgamma
>>> mgamma(1)
Matrix([
[ 0,  0, 0, 1],
[ 0,  0, 1, 0],
[ 0, -1, 0, 0],
[-1,  0, 0, 0]])
```

References

[R658]

sympy.physics.matrices.**msigma**(*i*)

Returns a Pauli matrix σ_i with $i = 1, 2, 3$.

Examples

```
>>> from sympy.physics.matrices import msigma
>>> msigma(1)
Matrix([
[0, 1],
[1, 0]])
```

References

[R659]

sympy.physics.matrices.**pat_matrix**(*m*, *dx*, *dy*, *dz*)

Returns the Parallel Axis Theorem matrix to translate the inertia matrix a distance of (dx, dy, dz) for a body of mass m.

Examples

To translate a body having a mass of 2 units a distance of 1 unit along the x-axis we get:

```
>>> from sympy.physics.matrices import pat_matrix
>>> pat_matrix(2, 1, 0, 0)
Matrix([
[0, 0, 0],
[0, 2, 0],
[0, 0, 2]])
```

Pauli Algebra

This module implements Pauli algebra by subclassing Symbol. Only algebraic properties of Pauli matrices are used (we do not use the Matrix class).

See the documentation to the class Pauli for examples.

References

sympy.physics.paulialgebra.**evaluate_pauli_product**(*arg*)

Help function to evaluate Pauli matrices product with symbolic objects.

Parameters

arg: symbolic expression that contains Paulimatrices

Examples

```
>>> from sympy.physics.paulialgebra import Pauli, evaluate_pauli_product
>>> from sympy import I
>>> evaluate_pauli_product(I*Pauli(1)*Pauli(2))
-sigma3
```

```
>>> from sympy.abc import x
>>> evaluate_pauli_product(x**2*Pauli(2)*Pauli(1))
-I*x**2*sigma3
```

Quantum Harmonic Oscillator in 1-D

sympy.physics.qho_1d.**E_n**(*n*, *omega*)

Returns the Energy of the One-dimensional harmonic oscillator.

Parameters

``n`` :

The "nodal" quantum number.

``omega`` :

The harmonic oscillator angular frequency.

Notes

The unit of the returned value matches the unit of hw, since the energy is calculated as:

E_n = hbar * omega*(n + 1/2)

Examples

```
>>> from sympy.physics.qho_1d import E_n
>>> from sympy.abc import x, omega
>>> E_n(x, omega)
hbar*omega*(x + 1/2)
```

sympy.physics.qho_1d.**coherent_state**(*n*, *alpha*)

Returns <n|alpha> for the coherent states of 1D harmonic oscillator. See https://en.wikipedia.org/wiki/Coherent_states

Parameters

``n`` :

The "nodal" quantum number.

``alpha`` :

The eigen value of annihilation operator.

sympy.physics.qho_1d.**psi_n**(*n*, *x*, *m*, *omega*)

Returns the wavefunction psi_{n} for the One-dimensional harmonic oscillator.

Parameters

``n`` :

the "nodal" quantum number. Corresponds to the number of nodes in the wavefunction. `n >= 0`

``x`` :

x coordinate.

``m`` :

Mass of the particle.

``omega`` :

Angular frequency of the oscillator.

Examples

```
>>> from sympy.physics.qho_1d import psi_n
>>> from sympy.abc import m, x, omega
>>> psi_n(0, x, m, omega)
(m*omega)**(1/4)*exp(-m*omega*x**2/(2*hbar))/(hbar**(1/4)*pi**(1/4))
```

Quantum Harmonic Oscillator in 3-D

`sympy.physics.sho.`**`E_nl`**`(`*n, l, hw*`)`

Returns the Energy of an isotropic harmonic oscillator.

Parameters

``n`` :

The "nodal" quantum number.

``l`` :

The orbital angular momentum.

``hw`` :

The harmonic oscillator parameter.

Notes

The unit of the returned value matches the unit of hw, since the energy is calculated as:

E_nl = (2*n + l + 3/2)*hw

Examples

```
>>> from sympy.physics.sho import E_nl
>>> from sympy import symbols
>>> x, y, z = symbols('x, y, z')
>>> E_nl(x, y, z)
z*(2*x + y + 3/2)
```

`sympy.physics.sho.`**`R_nl`**`(`*n, l, nu, r*`)`

Returns the radial wavefunction R_{nl} for a 3d isotropic harmonic oscillator.

Parameters

``n`` :

The "nodal" quantum number. Corresponds to the number of nodes in the wavefunction. `n >= 0`

``l`` :

The quantum number for orbital angular momentum.

``nu`` :

mass-scaled frequency: nu = m*omega/(2*hbar) where m is the mass and $omega$ the frequency of the oscillator. (in atomic units `nu == omega/2`)

``r`` :

Radial coordinate.

Examples

```
>>> from sympy.physics.sho import R_nl
>>> from sympy.abc import r, nu, l
>>> R_nl(0, 0, 1, r)
2*2**(3/4)*exp(-r**2)/pi**(1/4)
>>> R_nl(1, 0, 1, r)
4*2**(1/4)*sqrt(3)*(3/2 - 2*r**2)*exp(-r**2)/(3*pi**(1/4))
```

l, nu and r may be symbolic:

```
>>> R_nl(0, 0, nu, r)
2*2**(3/4)*sqrt(nu**(3/2))*exp(-nu*r**2)/pi**(1/4)
>>> R_nl(0, l, 1, r)
r**l*sqrt(2**(l + 3/2)*2**(l + 2)/factorial2(2*l + 1))*exp(-r**2)/pi**(1/
↪4)
```

The normalization of the radial wavefunction is:

```
>>> from sympy import Integral, oo
>>> Integral(R_nl(0, 0, 1, r)**2*r**2, (r, 0, oo)).n()
1.00000000000000
>>> Integral(R_nl(1, 0, 1, r)**2*r**2, (r, 0, oo)).n()
1.00000000000000
>>> Integral(R_nl(1, 1, 1, r)**2*r**2, (r, 0, oo)).n()
1.00000000000000
```

Second Quantization

Second quantization operators and states for bosons.

This follow the formulation of Fetter and Welecka, “Quantum Theory of Many-Particle Systems.”

class sympy.physics.secondquant.**AnnihilateBoson**(*k*)

Bosonic annihilation operator.

Examples

```
>>> from sympy.physics.secondquant import B
>>> from sympy.abc import x
>>> B(x)
AnnihilateBoson(x)
```

apply_operator(*state*)

Apply state to self if self is not symbolic and state is a FockStateKet, else multiply self by state.

Examples

```
>>> from sympy.physics.secondquant import B, BKet
>>> from sympy.abc import x, y, n
>>> B(x).apply_operator(y)
y*AnnihilateBoson(x)
>>> B(0).apply_operator(BKet((n,)))
sqrt(n)*FockStateBosonKet((n - 1,))
```

class sympy.physics.secondquant.**AnnihilateFermion**(*k*)

Fermionic annihilation operator.

apply_operator(*state*)

Apply state to self if self is not symbolic and state is a FockStateKet, else multiply self by state.

Examples

```
>>> from sympy.physics.secondquant import B, Dagger, BKet
>>> from sympy.abc import x, y, n
>>> Dagger(B(x)).apply_operator(y)
y*CreateBoson(x)
>>> B(0).apply_operator(BKet((n,)))
sqrt(n)*FockStateBosonKet((n - 1,))
```

property is_only_q_annihilator

Always destroy a quasi-particle? (annihilate hole or annihilate particle)

Examples

```
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import F
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
```

```
>>> F(a).is_only_q_annihilator
True
>>> F(i).is_only_q_annihilator
False
>>> F(p).is_only_q_annihilator
False
```

property is_only_q_creator

Always create a quasi-particle? (create hole or create particle)

Examples

```
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import F
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
```

```
>>> F(a).is_only_q_creator
False
>>> F(i).is_only_q_creator
True
>>> F(p).is_only_q_creator
False
```

property is_q_annihilator

Can we destroy a quasi-particle? (annihilate hole or annihilate particle) If so, would that be above or below the fermi surface?

Examples

```
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import F
>>> a = Symbol('a', above_fermi=1)
>>> i = Symbol('i', below_fermi=1)
>>> p = Symbol('p')
```

```
>>> F(a).is_q_annihilator
1
>>> F(i).is_q_annihilator
0
>>> F(p).is_q_annihilator
1
```

property is_q_creator

Can we create a quasi-particle? (create hole or create particle) If so, would that be above or below the fermi surface?

Examples

```
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import F
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
```

```
>>> F(a).is_q_creator
0
>>> F(i).is_q_creator
```

(continues on next page)

(continued from previous page)

```
-1
>>> F(p).is_q_creator
-1
```

class sympy.physics.secondquant.**AntiSymmetricTensor**(*symbol, upper, lower*)

Stores upper and lower indices in separate Tuple's.

Each group of indices is assumed to be antisymmetric.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.secondquant import AntiSymmetricTensor
>>> i, j = symbols('i j', below_fermi=True)
>>> a, b = symbols('a b', above_fermi=True)
>>> AntiSymmetricTensor('v', (a, i), (b, j))
AntiSymmetricTensor(v, (a, i), (b, j))
>>> AntiSymmetricTensor('v', (i, a), (b, j))
-AntiSymmetricTensor(v, (a, i), (b, j))
```

As you can see, the indices are automatically sorted to a canonical form.

property lower

Returns the lower indices.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.secondquant import AntiSymmetricTensor
>>> i, j = symbols('i,j', below_fermi=True)
>>> a, b = symbols('a,b', above_fermi=True)
>>> AntiSymmetricTensor('v', (a, i), (b, j))
AntiSymmetricTensor(v, (a, i), (b, j))
>>> AntiSymmetricTensor('v', (a, i), (b, j)).lower
(b, j)
```

property symbol

Returns the symbol of the tensor.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.secondquant import AntiSymmetricTensor
>>> i, j = symbols('i,j', below_fermi=True)
>>> a, b = symbols('a,b', above_fermi=True)
>>> AntiSymmetricTensor('v', (a, i), (b, j))
AntiSymmetricTensor(v, (a, i), (b, j))
>>> AntiSymmetricTensor('v', (a, i), (b, j)).symbol
v
```

property upper

Returns the upper indices.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.secondquant import AntiSymmetricTensor
>>> i, j = symbols('i,j', below_fermi=True)
>>> a, b = symbols('a,b', above_fermi=True)
>>> AntiSymmetricTensor('v', (a, i), (b, j))
AntiSymmetricTensor(v, (a, i), (b, j))
>>> AntiSymmetricTensor('v', (a, i), (b, j)).upper
(a, i)
```

sympy.physics.secondquant.**B**

alias of *AnnihilateBoson* (page 1542)

sympy.physics.secondquant.**BBra**

alias of *FockStateBosonBra* (page 1551)

sympy.physics.secondquant.**BKet**

alias of *FockStateBosonKet* (page 1552)

sympy.physics.secondquant.**Bd**

alias of *CreateBoson* (page 1547)

class sympy.physics.secondquant.**BosonicBasis**

Base class for a basis set of bosonic Fock states.

class sympy.physics.secondquant.**Commutator**(*a*, *b*)

The Commutator: [A, B] = A*B - B*A

The arguments are ordered according to .__cmp__()

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.secondquant import Commutator
>>> A, B = symbols('A,B', commutative=False)
>>> Commutator(B, A)
-Commutator(A, B)
```

Evaluate the commutator with .doit()

```
>>> comm = Commutator(A,B); comm
Commutator(A, B)
>>> comm.doit()
A*B - B*A
```

For two second quantization operators the commutator is evaluated immediately:

```
>>> from sympy.physics.secondquant import Fd, F
>>> a = symbols('a', above_fermi=True)
>>> i = symbols('i', below_fermi=True)
>>> p,q = symbols('p,q')
```

```
>>> Commutator(Fd(a),Fd(i))
2*NO(CreateFermion(a)*CreateFermion(i))
```

But for more complicated expressions, the evaluation is triggered by a call to .doit()

```
>>> comm = Commutator(Fd(p)*Fd(q),F(i)); comm
Commutator(CreateFermion(p)*CreateFermion(q), AnnihilateFermion(i))
>>> comm.doit(wicks=True)
-KroneckerDelta(i, p)*CreateFermion(q) +
 KroneckerDelta(i, q)*CreateFermion(p)
```

doit(***hints*)

Enables the computation of complex expressions.

Examples

```
>>> from sympy.physics.secondquant import Commutator, F, Fd
>>> from sympy import symbols
>>> i, j = symbols('i,j', below_fermi=True)
>>> a, b = symbols('a,b', above_fermi=True)
>>> c = Commutator(Fd(a)*F(i),Fd(b)*F(j))
>>> c.doit(wicks=True)
0
```

classmethod eval(*a*, *b*)

The Commutator [A,B] is on canonical form if A < B.

Examples

```
>>> from sympy.physics.secondquant import Commutator, F, Fd
>>> from sympy.abc import x
>>> c1 = Commutator(F(x), Fd(x))
>>> c2 = Commutator(Fd(x), F(x))
>>> Commutator.eval(c1, c2)
0
```

class sympy.physics.secondquant.**CreateBoson**(*k*)

Bosonic creation operator.

apply_operator(*state*)

Apply state to self if self is not symbolic and state is a FockStateKet, else multiply self by state.

Examples

```
>>> from sympy.physics.secondquant import B, Dagger, BKet
>>> from sympy.abc import x, y, n
>>> Dagger(B(x)).apply_operator(y)
y*CreateBoson(x)
>>> B(0).apply_operator(BKet((n,)))
sqrt(n)*FockStateBosonKet((n - 1,))
```

class sympy.physics.secondquant.**CreateFermion**(*k*)

Fermionic creation operator.

apply_operator(*state*)

Apply state to self if self is not symbolic and state is a FockStateKet, else multiply self by state.

Examples

```
>>> from sympy.physics.secondquant import B, Dagger, BKet
>>> from sympy.abc import x, y, n
>>> Dagger(B(x)).apply_operator(y)
y*CreateBoson(x)
>>> B(0).apply_operator(BKet((n,)))
sqrt(n)*FockStateBosonKet((n - 1,))
```

property is_only_q_annihilator

Always destroy a quasi-particle? (annihilate hole or annihilate particle)

Examples

```
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import Fd
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
```

```
>>> Fd(a).is_only_q_annihilator
False
>>> Fd(i).is_only_q_annihilator
True
>>> Fd(p).is_only_q_annihilator
False
```

property is_only_q_creator

Always create a quasi-particle? (create hole or create particle)

Examples

```
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import Fd
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
```

```
>>> Fd(a).is_only_q_creator
True
>>> Fd(i).is_only_q_creator
False
>>> Fd(p).is_only_q_creator
False
```

property is_q_annihilator

Can we destroy a quasi-particle? (annihilate hole or annihilate particle) If so, would that be above or below the fermi surface?

Examples

```
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import Fd
>>> a = Symbol('a', above_fermi=1)
>>> i = Symbol('i', below_fermi=1)
>>> p = Symbol('p')
```

```
>>> Fd(a).is_q_annihilator
0
>>> Fd(i).is_q_annihilator
-1
>>> Fd(p).is_q_annihilator
-1
```

property is_q_creator

Can we create a quasi-particle? (create hole or create particle) If so, would that be above or below the fermi surface?

Examples

```
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import Fd
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
```

```
>>> Fd(a).is_q_creator
1
>>> Fd(i).is_q_creator
```

(continues on next page)

(continued from previous page)

```
0
>>> Fd(p).is_q_creator
1
```

class sympy.physics.secondquant.**Dagger**(*arg*)

Hermitian conjugate of creation/annihilation operators.

Examples

```
>>> from sympy import I
>>> from sympy.physics.secondquant import Dagger, B, Bd
>>> Dagger(2*I)
-2*I
>>> Dagger(B(0))
CreateBoson(0)
>>> Dagger(Bd(0))
AnnihilateBoson(0)
```

classmethod eval(*arg*)

Evaluates the Dagger instance.

Examples

```
>>> from sympy import I
>>> from sympy.physics.secondquant import Dagger, B, Bd
>>> Dagger(2*I)
-2*I
>>> Dagger(B(0))
CreateBoson(0)
>>> Dagger(Bd(0))
AnnihilateBoson(0)
```

The eval() method is called automatically.

sympy.physics.secondquant.**F**

alias of *AnnihilateFermion* (page 1543)

sympy.physics.secondquant.**FBra**

alias of *FockStateFermionBra* (page 1552)

sympy.physics.secondquant.**FKet**

alias of *FockStateFermionKet* (page 1552)

sympy.physics.secondquant.**Fd**

alias of *CreateFermion* (page 1548)

class sympy.physics.secondquant.**FixedBosonicBasis**(*n_particles, n_levels*)

Fixed particle number basis set.

Examples

```
>>> from sympy.physics.secondquant import FixedBosonicBasis
>>> b = FixedBosonicBasis(2, 2)
>>> state = b.state(1)
>>> b
[FockState((2, 0)), FockState((1, 1)), FockState((0, 2))]
>>> state
FockStateBosonKet((1, 1))
>>> b.index(state)
1
```

index(*state*)

Returns the index of state in basis.

Examples

```
>>> from sympy.physics.secondquant import FixedBosonicBasis
>>> b = FixedBosonicBasis(2, 3)
>>> b.index(b.state(3))
3
```

state(*i*)

Returns the state that lies at index i of the basis

Examples

```
>>> from sympy.physics.secondquant import FixedBosonicBasis
>>> b = FixedBosonicBasis(2, 3)
>>> b.state(3)
FockStateBosonKet((1, 0, 1))
```

class sympy.physics.secondquant.**FockState**(*occupations*)

Many particle Fock state with a sequence of occupation numbers.

Anywhere you can have a FockState, you can also have S.Zero. All code must check for this!

Base class to represent FockStates.

class sympy.physics.secondquant.**FockStateBosonBra**(*occupations*)

Describes a collection of BosonBra particles.

Examples

```
>>> from sympy.physics.secondquant import BBra
>>> BBra([1, 2])
FockStateBosonBra((1, 2))
```

class sympy.physics.secondquant.**FockStateBosonKet**(*occupations*)

Many particle Fock state with a sequence of occupation numbers.

Occupation numbers can be any integer >= 0.

Examples

```
>>> from sympy.physics.secondquant import BKet
>>> BKet([1, 2])
FockStateBosonKet((1, 2))
```

class sympy.physics.secondquant.**FockStateBra**(*occupations*)

Representation of a bra.

class sympy.physics.secondquant.**FockStateFermionBra**(*occupations, fermi_level=0*)

Examples

```
>>> from sympy.physics.secondquant import FBra
>>> FBra([1, 2])
FockStateFermionBra((1, 2))
```

See also:

FockStateFermionKet (page 1552)

class sympy.physics.secondquant.**FockStateFermionKet**(*occupations, fermi_level=0*)

Many-particle Fock state with a sequence of occupied orbits.

Explanation

Each state can only have one particle, so we choose to store a list of occupied orbits rather than a tuple with occupation numbers (zeros and ones).

states below fermi level are holes, and are represented by negative labels in the occupation list.

For symbolic state labels, the fermi_level caps the number of allowed hole- states.

Examples

```
>>> from sympy.physics.secondquant import FKet
>>> FKet([1, 2])
FockStateFermionKet((1, 2))
```

class sympy.physics.secondquant.**FockStateKet**(*occupations*)

Representation of a ket.

class sympy.physics.secondquant.**InnerProduct**(*bra*, *ket*)

An unevaluated inner product between a bra and ket.

Explanation

Currently this class just reduces things to a product of Kronecker Deltas. In the future, we could introduce abstract states like |a> and |b>, and leave the inner product unevaluated as <a|b>.

property bra

Returns the bra part of the state

property ket

Returns the ket part of the state

class sympy.physics.secondquant.**KroneckerDelta**(*i*, *j*, *delta_range=None*)

The discrete, or Kronecker, delta function.

Parameters

i : Number, Symbol

The first index of the delta function.

j : Number, Symbol

The second index of the delta function.

Explanation

A function that takes in two integers i and j. It returns 0 if i and j are not equal, or it returns 1 if i and j are equal.

Examples

An example with integer indices:

```
>>> from sympy import KroneckerDelta
>>> KroneckerDelta(1, 2)
0
>>> KroneckerDelta(3, 3)
1
```

Symbolic indices:

```
>>> from sympy.abc import i, j, k
>>> KroneckerDelta(i, j)
KroneckerDelta(i, j)
>>> KroneckerDelta(i, i)
1
>>> KroneckerDelta(i, i + 1)
0
>>> KroneckerDelta(i, i + 1 + k)
KroneckerDelta(i, i + k + 1)
```

See also:

eval (page 1554), *DiracDelta* (page 450)

References

[R691]

classmethod eval(*i, j, delta_range=None*)

Evaluates the discrete delta function.

Examples

```
>>> from sympy import KroneckerDelta
>>> from sympy.abc import i, j, k
```

```
>>> KroneckerDelta(i, j)
KroneckerDelta(i, j)
>>> KroneckerDelta(i, i)
1
>>> KroneckerDelta(i, i + 1)
0
>>> KroneckerDelta(i, i + 1 + k)
KroneckerDelta(i, i + k + 1)
```

# indirect doctest

property indices_contain_equal_information

Returns True if indices are either both above or below fermi.

Examples

```
>>> from sympy import KroneckerDelta, Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, q).indices_contain_equal_information
True
```

(continues on next page)

(continued from previous page)

```
>>> KroneckerDelta(p, q+1).indices_contain_equal_information
True
>>> KroneckerDelta(i, p).indices_contain_equal_information
False
```

property is_above_fermi

True if Delta can be non-zero above fermi.

Examples

```
>>> from sympy import KroneckerDelta, Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, a).is_above_fermi
True
>>> KroneckerDelta(p, i).is_above_fermi
False
>>> KroneckerDelta(p, q).is_above_fermi
True
```

See also:

is_below_fermi (page 1555), *is_only_below_fermi* (page 1556), *is_only_above_fermi* (page 1555)

property is_below_fermi

True if Delta can be non-zero below fermi.

Examples

```
>>> from sympy import KroneckerDelta, Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, a).is_below_fermi
False
>>> KroneckerDelta(p, i).is_below_fermi
True
>>> KroneckerDelta(p, q).is_below_fermi
True
```

See also:

is_above_fermi (page 1555), *is_only_above_fermi* (page 1555), *is_only_below_fermi* (page 1556)

property is_only_above_fermi

True if Delta is restricted to above fermi.

Examples

```
>>> from sympy import KroneckerDelta, Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, a).is_only_above_fermi
True
>>> KroneckerDelta(p, q).is_only_above_fermi
False
>>> KroneckerDelta(p, i).is_only_above_fermi
False
```

See also:

is_above_fermi (page 1555), *is_below_fermi* (page 1555), *is_only_below_fermi* (page 1556)

property is_only_below_fermi

True if Delta is restricted to below fermi.

Examples

```
>>> from sympy import KroneckerDelta, Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, i).is_only_below_fermi
True
>>> KroneckerDelta(p, q).is_only_below_fermi
False
>>> KroneckerDelta(p, a).is_only_below_fermi
False
```

See also:

is_above_fermi (page 1555), *is_below_fermi* (page 1555), *is_only_above_fermi* (page 1555)

property killable_index

Returns the index which is preferred to substitute in the final expression.