drudge Documentation

Release 0.1.0

Jinmo Zhao and Gustavo E Scuseria

Sep 29, 2017

CONTENTS:

CHAPTER

ONE

INTRODUCTION

Drudge is a computer algebra system based on SymPy for noncommutative and tensor algebras, with a specific emphasis on many-body theory. To get started, Python of version at least 3.5 and a C++ compiler with good C++14 support are needed. In the development of drudge, Clang++ 3.9 and g++ 6.3 has been fully tested. Also Apache Spark of version later than 2.1 is needed for the parallel execution of drudge. For small tasks without requirement on parallelization, a fork of the DummyRDD project can be used in place of an actual Spark context. For parallel execution on supercomputers managed by the SLURM queueing system, the script `sbin/start-spark-on-slurm.sh` in the project source tree can be helpful.

As an experimental project, the documentation can be outdated, incomplete, incorrect, or have a lot of bad formatting. For any confusion, UTSL.

CHAPTER

TWO

DRUDGE TUTORIAL FOR BEGINNERS

2.1 Get started

Drudge is a library built on top of the SymPy computer algebra library for noncommutative and tensor alegbras. Usually for these style of problems, the symbolic manipulation and simplification of mathematical expressions requires a lot of context-dependent information, like the specific commutation rules and things like the dummy symbols to be used for different ranges. So the primary entry point for using the library is the *Drudge* class, which serves as a central repository of all kinds of domain-specific informations. To create a drudge instance, we need to give it a Spark context so that it is capable of parallelize things. For instance, to run things locally with all available cores, we can do

```
>>> import pyspark
>>> import drudge
>>> ctx = pyspark.SparkContext('local[*]', 'drudge-tutorial')
>>> dr = drudge.Drudge(ctx)
```

Then from it, we can create the symbolic expressions as *Tensor* objects, which are basically mathematical expressions containing noncommutative objects and symbolic summations. For the noncommutativity, in spite of the availability of some basic support of it in SymPy, here we have the *Vec* class to specifically designate the noncommutativity of its multiplication. It can be created with a label and indexed with SymPy expressions.

```
>>> v = drudge.Vec('v')
>>> import sympy
>>> a = sympy.Symbol('a')
>>> str(v[a])
'v[a]'
```

For the symbolic summations, we have the *Range* class, which denotes a symbolic set that a variable could be summed over. It can be created by just a label.

```
>>> l = drudge.Range('L')
```

With these, we can create tensor objects by using the *Drudge.sum()* method,

```
>>> x = sympy.IndexedBase('x')
>>> tensor = dr.sum((a, l), x[a] * v[a])
>>> str(tensor)
'sum_{a} x[a] * v[a]'
```

Now we got a symbolic tensor of a sum of vectors modulated by a SymPy IndexedBase. Actually any type of SymPy expression can be used to modulate the noncommutative vectors.

```
>>> tensor = dr.sum((a, l), sympy.sin(a) * v[a])
>>> str(tensor)
'sum_{a} sin(a) * v[a]'
```

And we can also have multiple summations and product of the vectors.

```
>>> b = sympy.Symbol('b')
>>> tensor = dr.sum((a, l), (b, l), x[a, b] * v[a] * v[b])
>>> str(tensor)
'sum_{a, b} x[a, b] * v[a] * v[b]'
```

Of cause the multiplication of the vectors will not be commutative,

```
>>> tensor = dr.sum((a, l), (b, l), x[a, b] * v[b] * v[a])
>>> str(tensor)
'sum_{a, b} x[a, b] * v[b] * v[a]'
```

Normally, for each symbolic range, we have some traditional symbols used as dummies for summations over them, giving these information to drudge objects can be very helpful. Here in this demonstration, we can use the *Drudge.set_dumms()* method.

```
>>> dr.set_dumms(l, sympy.symbols('a b c d'))
[a, b, c, d]
>>> dr.add_resolver_for_dumms()
```

where the call to the *Drudge.add_resolver_for_dumms()* method could tell the drudge to interpret all the dummy symbols to be over the range that they are set to. By giving drudge object such domain-specific information, we can have a lot convenience. For instance, now we can use Einstein summation convention to create tensor object, without the need to spell all the summations out.

```
>>> tensor = dr.einst(x[a, b] * v[a] * v[b])
>>> str(tensor)
'sum_{a, b} x[a, b] * v[a] * v[b]'
```

Also the drudge knows what to do when more dummies are needed in mathematical operations. For instance, when we multiply things,

```
>>> tensor = dr.einst(x[a] * v[a])
>>> prod = tensor * tensor
>>> str(prod)
'sum_{a, b} x[a]*x[b] * v[a] * v[b]'
```

Here the dummy b is automatically used since the drudge object knows available dummies for its range. Also the range and the dummies are automatically added to the name archive of the drudge, which can be access by *Drudge.names*.

```
>>> p = dr.names
>>> p.L
Range('L')
>>> p.L_dumms
[a, b, c, d]
>>> p.d
d
```

Here in this example, we set the dummies ourselves by *Drudge.set_dumms()*. Normally, in subclasses of *Drudge* for different specific problems, such setting up is already finished within the class. We can just directly get what we need from the names archive. There is also a method *Drudge.inject_names()* for the convenience of interactive work.

2.2 Tensor manipulations

Now with tensors created by *Drudge.sum()* or *Drudge.einst()*, a lot of mathematical operations are available to them. In addition to the above example of (noncommutative) multiplication, we can also have the linear algebraic operations of addition and scalar multiplication.

```
>>> tensor = dr.einst(x[a] * v[a])
>>> y = sympy.IndexedBase('y')
>>> res = tensor + dr.einst(y[a] * v[a])
>>> str(res)
'sum_{a} x[a] * v[a]\n + sum_{a} y[a] * v[a]'

>>> res = 2 * tensor
>>> str(res)
'sum_{a} 2*x[a] * v[a]'
```

We can also perform some complex substitutions on either the vector or the amplitude part, by using the `Drudge.subst()` method.

```
>>> t = sympy.IndexedBase('t')
>>> w = drudge.Vec('w')
>>> substed = tensor.subst(v[a], dr.einst(t[a, b] * w[b]))
>>> str(substed)
'sum_{a, b} x[a]*t[a, b] * w[b]'

>>> substed = tensor.subst(x[a], sympy.sin(a))
>>> str(substed)
'sum_{a} sin(a) * v[a]'
```

Note that here the substituted vector does not have to match the left-hand side of the substitution exactly, pattern matching is done here. Other mathematical operations are also available, like symbolic differentiation by *Tensor.diff()* and commutation by `|` operator *Tensor.__or__()*.

Tensors are purely mathematical expressions, while the utility class *TensorDef* can be construed as tensor expressions with a left-hand side. They can be easily created by *Drudge.define()* and *Drudge.define_einst()*.

```
>>> v_def = dr.define_einst(v[a], t[a, b] * w[b])
>>> str(v_def)
'v[a] = sum_{b} t[a, b] * w[b]'
```

Their method *TensorDef.act()* is like a active voice version of *Tensor.subst()* and could come handy when we need to substitute the same definition in multiple inputs.

```
>>> res = v_def.act(tensor)
>>> str(res)
'sum_{a, b} x[a]*t[a, b] * w[b]'
```

More importantly, the definitions can be indexed directly, and the result is designed to work well inside *Drudge.sum()* or *Drudge.einst()*. For instance, for the same result, we could have,

```
>>> res = dr.einst(x[a] * v_def[a])
>>> str(res)
'sum_{b, a} x[a]*t[a, b] * w[b]'
```

When the only purpose of a vector or indexed base is to be substituted and we never intend to write tensor expressions directly in terms of them, we can just name the definition with a short name directly and put the actual base inside only. For instance,

```
>>> c = sympy.Symbol('c')
>>> f = dr.define_einst(sympy.IndexedBase('f')[a, b], x[a, c] * y[c, b])
>>> str(f)
'f[a, b] = sum_{c} x[a, c]*y[c, b]'
>>> str(dr.einst(f[a, a]))
'sum_{b, a} x[a, b]*y[b, a]'
```

which also demonstrates that the tensor definition facility can also be used for scalar quantities. *TensorDef* is also at the core of the code optimization and generation facility in the `gristmill` package.

Usually for tensorial problems, full simplification requires the utilization of some symmetries present on the indexed quantities by permutations among their indices. For instance, an anti-symmetric matrix entry changes sign when we transpose the two indices. Such information can be told to drudge by using the *Drudge.set_symm()* method, by giving generators of the symmetry group by *Perm* instances. For instance, we can do,

```
dr.set_symm(x, drudge.Perm([1, 0], drudge.NEG))
```

Then the master simplification algorithm in *Tensor.simplify()* is able to take full advantage of such information.

```
>>> tensor = dr.einst(x[a, b] * v[a] * v[b] + x[b, a] * v[a] * v[b])
>>> str(tensor)
'sum_{a, b} x[a, b] * v[a] * v[b]\n + sum_{a, b} x[b, a] * v[a] * v[b]'
>>> str(tensor.simplify())
'0'
```

Normally, drudge subclasses for specific problems add symmetries for some important indexed bases in the problem. And some drudge subclasses have helper methods for the setting of such symmetries, like *FockDrudge.set_n_body_base()* and *FockDrudge.set_dbbar_base()*.

For the simplification of the noncommutative vector parts, the base *Drudge* class does **not** consider any commutation rules among the vectors. It works on the free algebra, while the subclasses could have the specific commutation rules added for the algebraic system. For instance, *WickDrudge* add abstract commutation rules where all the commutators have scalar values. Based on it, its special subclass *FockDrudge* implements the canonical commutation relations for bosons and the canonical anti-commutation relations for fermions.

These drudge subclasses only has the mathematical commutation rules implemented, for convenience in solving problems, many drudge subclasses are built-in with a lot of domain-specific information like the ranges and dummies, which are listed in *Direct support of different problems*. For instance, we can easily see the commutativity of two particle-hole excitation operators by using the *PartHoleDrudge*.

```
>>> phdr = drudge.PartHoleDrudge(ctx)
>>> t = sympy.IndexedBase('t')
>>> u = sympy.IndexedBase('u')
>>> p = phdr.names
>>> a, i = p.a, p.i
>>> excit1 = phdr.einst(t[a, i] * p.c_dag[a] * p.c_[i])
>>> excit2 = phdr.einst(u[a, i] * p.c_dag[a] * p.c_[i])
>>> comm = excit1 | excit2
>>> str(comm)
'sum_{i, a, j, b} t[a, i]*u[b, j] * c[CR, a] * c[AN, i] * c[CR, b] * c[AN, j]\n + sum_
↪{i, a, j, b} -t[a, i]*u[b, j] * c[CR, b] * c[AN, j] * c[CR, a] * c[AN, i]'
>>> str(comm.simplify())
'0'
```

Note that here basically all things related to the problem, like the vector for creation and annihilation operator, the conventional dummies a and i for particle and hole labels, are directly read from the name archive of the drudge. Problem-specific drudges are supposed to give such convenience.

In addition to providing context-dependent information for general tensor operations, drudge subclasses could also provide additional operations on tensors created from them. For instance, for the above commutator, we can directly compute the expectation value with respect to the Fermi vacuum by

```
>>> str(comm.eval_fermi_vev())
'0'
```

These additional operations are called tensor methods and are documented in the drudge subclasses.

2.3 Examples on real-world applications

In this tutorial, some simple examples are run directly inside a Python interpreter. Actually drudge is designed to work well inside Jupyter notebooks. By calling the *Tensor.display()* method, tensor objects can be mathematically displayed in Jupyter sessions. An example of interactive usage of drudge, we have a sample notebook in `docs/examples/ccsd.ipynb` in the project source. Also included is a general script `gencc.py` for the automatic derivation of coupled-cluster theories, mostly to demonstrate using drudge programmatically. And we also have a script for RCCSD theory to demonstrate its usage in large-scale spin-explicit coupled-cluster theories.

CHAPTER

THREE

DRUDGE API REFERENCE GUIDE

3.1 Base drudge system

The base drudge system handles the part of program logic universally applicable to any tensor and noncommutative algebra system.

3.1.1 Building blocks of the basic drudge data structure

class drudge.**Range**(*label*, *lower=None*, *upper=None*)

A symbolic range that can be summed over.

This class is for symbolic ranges that is going to be summed over in tensors. Each range should have a label, and optionally lower and upper bounds, which should be both given or absent. The label can be any hashable and ordered Python type. The bounds will not be directly used for symbolic computation, but rather designed for printers and conversion to SymPy summation. Note that ranges are assumed to be atomic and disjoint. Even in the presence of lower and upper bounds, unequal ranges are assumed to be disjoint.

> **Warning:** Bounds with the same label but different bounds will be considered unequal. Although no error be given, using different bounds with identical label is strongly advised against.

> **Warning:** Unequal ranges are always assumed to be disjoint.

__init__(*label*, *lower=None*, *upper=None*)

Initialize the symbolic range.

label

The label of the range.

lower

The lower bound of the range.

upper

The upper bound of the range.

size

The size of the range.

This property given None for unbounded ranges. For bounded ranges, it is the difference between the lower and upper bound. Note that this contradicts the deeply entrenched mathematical convention of including other ends for a range. But it does gives a lot of convenience and elegance.

bounded
If the range is explicitly bounded.

args
The arguments for range creation.

When the bounds are present, we have a triple, or we have a singleton tuple of only the label.

__hash__()
Hash the symbolic range.

__eq__(*other*)
Compare equality of two ranges.

__repr__()
Form the representative string.

__str__()
Form readable string representation.

sort_key
The sort key for the range.

replace_label(*new_label*)
Replace the label of a given range.

The bounds will be the same as the original range.

__lt__(*other*)
Compare two ranges.

This method is meant to skip explicit calling of the sort key when it is not convenient.

class drudge.**Vec**(*label*, *indices=()*)
Vectors.

Vectors are the basic non-commutative quantities. Its objects consist of an label for its base and some indices. The label is allowed to be any hashable and ordered Python object, although small objects, like string, are advised. The indices are always sympified into SymPy expressions.

Its objects can be created directly by giving the label and indices, or existing vector objects can be subscribed to get new ones. The semantics is similar to Haskell functions.

Note that users cannot directly assign to the attributes of this class.

This class can be used by itself, it can also be subclassed for special use cases.

Despite very different internal data structure, the this class is attempted to emulate the behaviour of the SymPy `IndexedBase` class

__init__(*label*, *indices=()*)
Initialize a vector.

Atomic indices are added as the only index. Iterable values will have all of its entries added.

label
The label for the base of the vector.

base
The base of the vector.

This base can be subscribed to get other vectors.

indices
The indices to the vector.

__getitem__(*item*)
Append the given indices to the vector.

When multiple new indices are to be given, they have to be given as a tuple.

__repr__()
Form repr string form the vector.

__str__()
Form a more readable string representation.

__hash__()
Compute the hash value of a vector.

__eq__(*other*)
Compares the equality of two vectors.

sort_key
The sort key for the vector.

This is a generic sort key for vectors. Note that this is only useful for sorting the simplified terms and should not be used in the normal-ordering operations.

map(*func*)
Map the given function to indices.

terms
Get the terms from the vector.

This is for the user input.

class drudge.**Term**(*sums: typing.Tuple[typing.Tuple[sympy.core.symbol.Symbol, drudge.term.Range], ...], amp: sympy.core.expr.Expr, vecs: typing.Tuple[drudge.term.Vec, ...], free_vars: typing.FrozenSet[sympy.core.symbol.Symbol] = None, dumms: typing.Mapping[sympy.core.symbol.Symbol, drudge.term.Range] = None*)
Terms in tensor expression.

This is the core class for storing symbolic tensor expressions. The actual symbolic tensor type is just a shallow wrapper over a list of terms. It is basically comprised of three fields, a list of summations, a SymPy expression giving the amplitude, and a list of non-commutative vectors.

__init__(*sums: typing.Tuple[typing.Tuple[sympy.core.symbol.Symbol, drudge.term.Range], ...], amp: sympy.core.expr.Expr, vecs: typing.Tuple[drudge.term.Vec, ...], free_vars: typing.FrozenSet[sympy.core.symbol.Symbol] = None, dumms: typing.Mapping[sympy.core.symbol.Symbol, drudge.term.Range] = None*)
Initialize the tensor term.

Users seldom have the need to create terms directly by this function. So this constructor is mostly a developer function, no sanity checking is performed on the input for performance. Most importantly, this constructor does **not** copy either the summations or the vectors and directly expect them to be tuples (for hashability). And the amplitude is **not** simpyfied.

Also, it is important that the free variables and dummies dictionary be given only when they really satisfy what we got for them.

sums
The summations of the term.

amp
The amplitude expression.

vecs
The vectors in the term.

is_scalar
If the term is a scalar.

args
The triple of summations, amplitude, and vectors.

__hash__()
Compute the hash of the term.

__eq__(*other*)
Evaluate the equality with another term.

__repr__()
Form the representative string of a term.

__str__()
Form the readable string representation of a term.

sort_key
The sort key for a term.

This key attempts to sort the terms by complexity, with simpler terms coming earlier. This capability of sorting the terms will make the equality comparison of multiple terms easier.

This sort key also ensures that terms that can be merged are always put into adjacent positions.

terms
The singleton list of the current term.

This property is for the rare cases where direct construction of tensor inputs from SymPy expressions and vectors are not sufficient.

scale(*factor*)
Scale the term by a factor.

mul_term(*other*, *dumms=None*, *excl=None*)
Multiply with another tensor term.

Note that by this function, the free symbols in the two operands are not automatically excluded.

comm_term(*other*, *dumms=None*, *excl=None*)
Commute with another tensor term.

In ths same way as the multiplication operation, here the free symbols in the operands are not automatically excluded.

reconcile_dumms(*other*, *dumms*, *excl*)
Reconcile the dummies in two terms.

exprs
Loop over the sympy expression in the term.

Note that the summation dummies are not looped over.

free_vars
The free symbols used in the term.

dumms
Get the mapping from dummies to their range.

amp_factors
The factors in the amplitude expression.

The indexed factors and factors involving dummies will be returned as a list, with the rest returned as a single SymPy expression.

Error will be raised if the amplitude is not a monomial.

map (*func*, *sums=None*, *amp=None*, *vecs=None*, *skip_vecs=False*)
Map the given function to the SymPy expressions in the term.

The given function will **not** be mapped to the dummies in the summations. When operations on summations are needed, a **tuple** for the new summations can be given.

By passing the identity function, this function can also be used to replace the summation list, the amplitude expression, or the vector part.

subst (*substs*, *sums=None*, *amp=None*, *vecs=None*, *purge_sums=False*)
Perform symbol substitution on the SymPy expressions.

After the replacement of the fields given, the given substitutions are going to be performed using SymPy `xreplace` method simultaneously.

If purge sums is set, the summations whose dummy is substituted is going to be removed.

reset_dumms (*dumms*, *dummbegs=None*, *excl=None*)
Reset the dummies in the term.

The term with dummies reset will be returned alongside with the new dummy begins dictionary. Note that the dummy begins dictionary will be mutated if one is given.

ValueError will be raised when no more dummies are available.

static reset_sums (*sums*, *dumms*, *dummbegs=None*, *excl=None*)
Reset the given summations.

The new summation list, substitution dictionary, and the new dummy begin dictionary will be returned.

simplify_deltas (*resolvers*)
Simplify deltas in the amplitude of the expression.

expand ()
Expand the term into many terms.

canon (*symms=None*, *vec_colour=None*)
Canonicalize the term.

The given vector colour should be a callable accepting the index within vector list (under the keyword `idx`) and the vector itself (under keyword `vec`). By default, vectors has colour the same as its index within the list of vectors.

Note that whether or not colours for the vectors are given, the vectors are never permuted in the result.

has_base (*base*)
Test if the given base is present in the current term.

3.1.2 Canonicalization of indexed quantities with symmetry

Some actions are supported to accompany the permutation of indices to indexed quantities. All of these accompanied action can be composed by using the bitwise or operator `|`.

`drudge.`**IDENT**
The identitiy action. Nothing is performed for the permutation.

`drudge.`**NEG**
Negation. When the given permutation is performed, the indexed quantity needs to be negated. For instance, in anti-symmetric matrix.

drudge.**CONJ**
Conjugation. When the given permutation is performed, the indexed quantity needs to be taken it complex conjugate. Note that this action can only be used in the symmetry of scalar indexed quantities.

class drudge.**Perm**
Permutation of points with accompanied action.

Permutations can be constructed from an iterable giving the pre-image of the points and an optional integral value for the accompanied action. The accompanied action can be given positionally or by the keyword `acc`, and it will be manipulated according to the convention in libcanon.

Querying the length of a Perm object gives the size of the permutation domain, while indexing it gives the pre-image of the given integral point. The accompanied action can be obtained by getting the attribute `acc`. Otherwise, this data type is mostly opaque.

acc
The accompanied action.

class drudge.**Group**
Permutations groups.

To create a permutation group, an iterable of Perm objects or pre-image array action pair can be given for the generators of the group. Then the Schreier-Sims algorithm in libcanon will be invoked to generate the Sims transversal system, which will be stored internally for the group. This class is mostly designed to be used to give input for the Eldag canonicalization facility. So it is basically an opaque object after its creation.

Internally, the transversal system can also be constructed directly from the transversal system, without going through the Schreier-Sims algorithm. However, that is more intended for serialization rather than direct user invocation.

3.1.3 Primary interface

class drudge.**Drudge**(*ctx: pyspark.context.SparkContext*, *num_partitions=True*)
The main drudge class.

A drudge is a robot who can help you with the menial tasks of symbolic manipulation for tensorial and noncommutative alegbras. Due to the diversity and non-uniformity of tensor and noncommutative algebraic problems, to set up a drudge, domain-specific information about the problem needs to be given. Here this is a base class, where the basic operations are defined. Different problems could subclass this base class with customized behaviour. Most importantly, the method *normal_order()* should be overridden to give the commutation rules for the algebraic system studied.

__init__(*ctx: pyspark.context.SparkContext*, *num_partitions=True*)
Initialize the drudge.

Parameters

- **ctx** – The Spark context to be used.
- **num_partitions** – The preferred number of partitions. By default, it is the default parallelism of the given Spark environment. Or an explicit integral value can be given. It can be set to None, which disable all explicit load-balancing by shuffling.

ctx
The Spark context of the drudge.

num_partitions
The preferred number of partitions for data.

full_simplify
If full simplification is to be performed on amplitudes.

It can be used to disable full simplification of the amplitude expression by SymPy. For simple polynomial amplitude, this option is generally safe to be disabled.

simple_merge
If only simple merge is to be carried out.

When it is set to true, only terms with same factors involving dummies are going to be merged. This might be helpful for cases where the amplitude are all simple polynomials of tensorial quantities. Note that this could disable some SymPy simplification.

> **Warning:** This option might not give much more than disabling full simplification but taketh away many simplifications. It is in general not recommended to be used.

set_name(**args*, ***kwargs*)
Set the object into the name archive of the drudge.

For positional arguments, the str form of the given label is going to be used for the name of the object. For keyword arguments, the keyword will be used for the name.

names
The name archive for the drudge.

The name archive object can be used for convenient accessing of objects related to the problem.

inject_names(*prefix=''*, *suffix=''*)
Inject the names in the name archive into the current global scope.

This function is for the convenience of users, especially interactive users. Itself is not used in official drudge code except its own tests.

Note that this function injects the names in the name archive into the **global** scope of the caller, rather than the local scope, even when called inside a function.

set_dumms(*range_: drudge.term.Range*, *dumms*, *set_range_name=True*, *dumms_suffix='_dumms'*, *set_dumm_names=True*)
Set the dummies for a range.

Note that this function overwrites the existing dummies if the range has already been given.

dumms
The broadcast form of the dummies dictionary.

set_symm(*base*, **symms*, *valence=None*, *set_base_name=True*)
Set the symmetry for a given base.

Permutation objects in the arguments are interpreted as single generators, other values will be attempted to be iterated over to get their entries, which should all be permutations.

Parameters

- **base** – The SymPy indexed base object or vectors whose symmetry is to be set.
- **symms** – The generators of the symmetry. It can be a single None to remove the symmetry of the given base.
- **valence** (*int*) – When it is set, only the indexed quantity of the base with the given valence will have the given symmetry.
- **set_base_name** – If the base name is to be added to the name archive of the drudge.

symms
The broadcast form of the symmetries.

add_resolver(*resolver*)
Append a resolver to the list of resolvers.

The given resolver can be either a mapping from SymPy expression, including atomic symbols, to the corresponding ranges. Or a callable to be called with SymPy expressions. For callable resolvers, None can be returned to signal the incapability to resolve the expression. Then the resolution will be dispatched to the next resolver.

add_resolver_for_dumms()
Add the resolver for the dummies for each range.

With this method, the default dummies for each range will be resolved to be within the range for all of them. This method should normally be called by all subclasses after the dummies for all ranges have been properly set.

Note that dummies added later will not be automatically added. This method can be called again.

resolvers
The broadcast form of the resolvers.

set_tensor_method(*name*, *func*)
Set a new tensor method under the given name.

A tensor method is a method that can be called from tensors created from the current drudge as if it is a method of the given tensor. This could give cleaner and more consistent code for all tensor manipulations.

The given function, or bounded method, should be able to accept the tensor as the first argument.

get_tensor_method(*name*)
Get a tensor method with given name.

When the name cannot be resolved, KeyError will be raised.

vec_colour
The vector colour function.

Note that this accessor accesses the **function**, rather than directly computes the colour for any vector.

normal_order(*terms*, ***kwargs*)
Normal order the terms in the given tensor.

This method should be called with the RDD of some terms, and another RDD of terms, where all the vector parts are normal ordered according to domain-specific rules, should be returned.

By default, we work for the free algebra. So nothing is done by this function. For noncommutative algebraic system, this function needs to be overridden to return an RDD for the normal-ordered terms from the given terms.

sum(**args*, *predicate=None*) → drudge.drudge.Tensor
Create a tensor for the given summation.

This is the core function for creating tensors from scratch. The arguments should start with the summations, each of which should be given as a sequence, normally a tuple, starting with a SymPy symbol for the summation dummy in the first entry. Then comes possibly multiple domains that the dummy is going to be summed over, which can be symbolic range, SymPy expression, or iterable over them. When symbolic ranges are given as `Range` objects, the given dummy will be set to be summed over the ranges symbolically. When SymPy expressions are given, the given values will substitute all appearances of the dummy in the summand. When we have multiple summations, terms in the result are generated from the Cartesian product of them.

The last argument should give the actual thing to be summed, which can be something that can be interpreted as a collection of terms, or a callable that is going to return the summand when given a dictionary giving the action on each of the dummies. The dictionary has an entry for all the dummies. Dummies summed over symbolic ranges will have the actual range as its value, or the actual SymPy expression when it is given a concrete range. In the returned summand, if dummies still exist, they are going to be treated in the same way as statically-given summands.

The predicate can be a callable going to return a boolean when called with same dictionary. False values can be used the skip some terms. It is guaranteed that the same dictionary will be used for both predicate and the summand when they are given as callables.

Note that this function can also be called on existing tensor objects with the same semantics on the terms. Existing summations are not touched by it.

einst(*summand*) → drudge.drudge.Tensor
Create a tensor from Einstein summation convention.

By calling this function, summations according to the Einstein summation convention will be added to the terms. Note that for a symbol to be recognized as a summation, it must appear exactly twice in its **original form** in indices, and its range needs to be able to be resolved. When a symbol is suspiciously an Einstein summation dummy but does not satisfy the requirement precisely, it will **not** be added as a summation, but a warning will also be given for reference.

Note that in addition to creating tensors from scratch, this method can also be called on an existing tensor to add new summations. In that case, no existing summations will be touched.

create_tensor(*terms*)
Create a tensor with the terms given in the argument.

The terms should be given as an iterable of Term objects. This function should not be necessary in user code.

define(**args*) → drudge.drudge.TensorDef
Make a tensor definition.

This is a helper method for the creation of *TensorDef* instances.

Parameters

- **arguments** (*initial*) – The left-hand side of the definition. It can be given as an indexed quantity, either SymPy Indexed instances or an indexed vector, with all the indices being plain symbols whose range is able to be resolved. Or a base can be given, followed by the symbol/range pairs for the external indices.
- **argument** (*final*) – The definition of the LHS, can be tensor instances, or anything capable of being interpreted as such. Note that no summation is going to be automatically added.

define_einst(**args*) → drudge.drudge.TensorDef
Make a tensor definition based on Einstein summation convention.

Basically the same function as the *define()*, just the content will be interpreted according to the Einstein summation convention.

format_latex(*inp*, *sep_lines=False*)
Get the LaTeX form of a given tensor or tensor definition.

Subclasses should fine-tune the appearance of the resulted LaTeX form by overriding methods `_latex_sympy`, `_latex_vec`, and `_latex_vec_mul`.

__weakref__
list of weak references to the object (if defined)

report(*filename*, *title*)
Make a report for results.

This function should be used within a `with` statement to open a report for results.

class drudge.**Tensor**(*drudge: drudge.drudge.Drudge*, *terms: pyspark.rdd.RDD*, *free_vars: typing.Set[sympy.core.symbol.Symbol] = None*, *expanded=False*, *repartitioned=False*)
The main tensor class.

A tensor is an aggregate of terms distributed and managed by Spark. Here most operations needed for tensors are defined.

Normally, tensor instances are created from drudge methods or tensor operations. Direct invocation of its constructor is seldom in user scripts.

__init__(*drudge: drudge.drudge.Drudge*, *terms: pyspark.rdd.RDD*, *free_vars: typing.Set[sympy.core.symbol.Symbol] = None*, *expanded=False*, *repartitioned=False*)
Initialize the tensor.

This function is not designed to be called by users directly. Tensor creation should be carried out by factory function inside drudges and the operations defined here.

The default values for the keyword arguments are always the safest choice, for better performance, manipulations are encouraged to have proper consideration of all the keyword arguments.

drudge
The drudge created the tensor.

terms
The terms in the tensor, as an RDD object.

Although for users, normally there is no need for direct manipulation of the terms, it is still exposed here for flexibility.

local_terms
Gather the terms locally into a list.

The list returned by this is for read-only and should **never** be mutated.

> **Warning:** This method will gather all terms into the memory of the driver.

n_terms
Get the number of terms.

A zero number of terms signatures a zero tensor. Accessing this property will make the tensor to be cached automatically.

cache()
Cache the terms in the tensor.

This method should be called when this tensor is an intermediate result that will be used multiple times. The tensor itself will be returned for the ease of chaining.

repartition(*num_partitions=None*, *cache=False*)
Repartition the terms across the Spark cluster.

This function should be called when the terms need to be rebalanced among the workers. Note that this incurs an Spark RDD shuffle operation and might be very expensive. Its invocation and the number of partitions used need to be fine-tuned for different problems to achieve good performance.

Parameters

- **num_partitions** (*int*) – The number of partitions. By default, the number is read from the drudge object.
- **cache** (*bool*) – If the result is going to be cached.

is_scalar
If the tensor is a scalar.

A tensor is considered a scalar when none of its terms has a vector part. This property will make the tensor automatically cached.

free_vars
The free variables in the tensor.

expanded
If the tensor is already expanded.

repartitioned
If the terms in the tensor is already repartitioned.

has_base(*base: typing.Union[sympy.tensor.indexed.IndexedBase, sympy.core.symbol.Symbol, drudge.term.Vec]*) → bool
Find if the tensor has the given scalar or vector base.

Parameters **base** – The base whose presence is to be queried. When it is indexed base or a plain symbol, its presence in the amplitude part is tested. When it is a vector, its presence in the vector part is tested.

__str__()
Get the string representation of the tensor.

Note that this function will **gather** all terms into the driver.

latex(*sep_lines=False*)
Get the latex form for the tensor.

The actual printing is dispatched to the drudge object for the convenience of tuning the appearance.

Parameters **sep_lines** (*bool*) – If terms should be put into separate lines by separating them with \\.

display(*if_return=True*, *sep_lines=False*)
Display the tensor in interactive IPython notebook sessions.

Parameters

- **if_return** – If the resulted equation be returned rather than directly displayed. It can be disabled for displaying equation in the middle of a Jupyter cell.
- **sep_lines** – If terms should be written into separate lines.

apply(*func*, ***kwargs*)
Apply the given function to the RDD of terms.

This function is analogous to the replace function of Python named tuples, the same value from self for the tensor initializer is going to be used when it is not given. The terms get special treatment since it is the centre of tensor objects. The drudge is kept the same always.

Users generally do not need this method. It is exposed here just for flexibility and convenience.

Warning: For developers: Note that the resulted tensor will inherit all unspecified keyword arguments from self. This method can give *unexpected results* if certain arguments are not correctly reset when

they need to. For instance, when expanded is not reset when the result is no longer guaranteed to be in expanded form, later expansions could be skipped when they actually need to be performed.

So all functions using this methods need to be reviewed when new property are added to tensor class. Direct invocation of the tensor constructor is a much safe alternative.

reset_dumms(*excl=None*)
Reset the dummies.

The dummies will be set to the canonical dummies according to the order in the summation list. This method is especially useful on canonicalized tensors.

Parameters **excl** – A set of symbols to be excluded in the dummy selection. This option can be useful when some symbols already used as dummies are planned to be used for other purposes.

simplify_amps()
Simplify the amplitudes in the tensor.

This method simplifies the amplitude in the terms of the tensor by using the facility from SymPy. The zero terms will be filtered out as well.

simplify_deltas()
Simplify the deltas in the tensor.

Kronecker deltas whose operands contains dummies will be attempted to be simplified.

expand()
Expand the terms in the tensor.

By calling this method, terms in the tensor whose amplitude is the addition of multiple parts will be expanded into multiple terms.

sort()
Sort the terms in the tensor.

The terms will generally be sorted according to increasing complexity.

merge()
Merge terms with the same vector and summation part.

This function merges terms only when their summation list and vector part are *syntactically* the same. So it is more useful when the canonicalization has been performed and the dummies reset.

canon()
Canonicalize the terms in the tensor.

This method will first expand the terms in the tensor. Then the canonicalization algorithm is going to be applied to each of the terms. Note that this method does not rename the dummies.

normal_order()
Normal order the terms in the tensor.

The actual work is dispatched to the drudge, who has domain specific knowledge about the noncommutativity of the vectors.

simplify()
Simplify the tensor.

This is the master driver function for tensor simplification.

__eq__(*other*)
Compare the equality of tensors.

Note that this function only compares the syntactical equality of tensors. Mathematically equal tensors might be compared to be unequal by this function when they are not simplified.

Note that only comparison with zero is performed by counting the number of terms distributed. Or this function gathers all terms in both tensors and can be very expensive. So direct comparison of two tensors is mostly suitable for testing and debugging on small problems only. For large scale problems, it is advised to compare the simplified difference with zero.

__add__(*other*)
Add the two tensors together.

The terms in the two tensors will be concatenated together, without any further processing.

In addition to full tensors, tensor inputs can also be directly added.

__radd__(*other*)
Add tensor with something in front.

__sub__(*other*)
Subtract another tensor from this tensor.

__rsub__(*other*)
Subtract the tensor from another quantity.

__mul__(*other*) → drudge.drudge.Tensor
Multiply the tensor.

This multiplication operation is done completely within the framework of free algebras. The vectors are only concatenated without further processing. The actual handling of the commutativity should be carried out at the normal ordering operation for different problems.

In addition to full tensors, tensors can also be multiplied to user tensor input directly.

__rmul__(*other*)
Multiply the tensor on the right.

__or__(*other*)
Compute the commutator with another tensor.

In the same way as multiplication, this can be used for both full tensors and local tensor input.

__ror__(*other*)
Compute the commutator with another tensor on the right.

subst(*lhs*, *rhs*, *wilds=None*)
Substitute the all appearance of the defined tensor.

When the given LHS is a plain SymPy symbol, all its appearances in the amplitude of the tensor will be replaced. Or the LHS can also be indexed SymPy expression or indexed Vector, for which all of the appearances of the indexed base or vector base will be attempted to be matched against the indices on the LHS. When a matching succeeds for all the indices, the RHS, with the substitution found in the matching performed, will be replace the indexed base in the amplitude, or the vector. Note that for scalar LHS, the RHS must contain no vector.

Since we do not commonly define tensors with wild symbols, an option `wilds` can be used to give a mapping translating plain symbols on the LHS and the RHS to the wild symbols that would like to be used. The default value of None could make all **plain** symbols in the indices of the LHS to be translated into a wild symbol with the same name and no exclusion. And empty dictionary can be used to disable all such automatic translation. The default value of None should satisfy most needs.

subst_all(*defs*, *simplify=False*)
Substitute all given definitions serially.

The definitions should be given as an iterable of either *TensorDef* instances or pairs of left-hand side and right-hand side of the substitutions. Note that the substitutions are going to be performed **according to the given order** one-by-one, rather than simultaneously.

diff(*variable*, *real=False*, *wirtinger_conj=False*)
Differentiate the tensor to get the analytic gradient.

By this function, support is provided for evaluating the derivative with respect to either a plain symbol or a tensor component. This is achieved by leveraging the core differentiation operation to SymPy. So very wide range of expressions are supported.

Warning: For non-analytic complex functions, this function gives the Wittinger derivative with respect to the given variable only. The other non-vanishing derivative with respect to the conjugate needs to be evaluated by another invocation with `wittinger_conj` set to true.

Warning: The differentiation algorithm currently does **not** take the symmetry of the tensor to be differentiated with respect to into account. For differentiate with respect to symmetric tensor, further symmetrization of the result might be needed.

Parameters

- **variable** – The variable to differentiate with respect to. It should be either a plain SymPy symbol or a indexed quantity. When it is an indexed quantity, the indices should be plain symbols with resolvable range.
- **real** (*bool*) – If the variable is going to be assumed to be real. Real variables has conjugate equal to themselves.
- **wirtinger_conj** (*bool*) – If we evaluate the Wirtinger derivative with respect to the conjugate of the variable.

filter(*crit*)
Filter out terms satisfying the given criterion.

map2scalars(*action*, *skip_vecs=False*)
Map the given action to the scalars in the tensor.

The given action should return SymPy expressions for SymPy expressions, the amplitude for each terms and the indices to the vectors, in the tensor. Note that this function does not change the summations in the terms and the dummies.

Parameters

- **action** – The callable to be applied to the scalars inside the tensor.
- **skip_vecs** – When it is set, the callable will no longer be mapped to the indices to the vectors. It could be used to boost the performance when we know that the action need no application on the indices.

__getattr__(*item*)
Try to see if the item is a tensor method from the drudge.

This enables individual drudges to dynamically add domain-specific operations on tensors.

class drudge.**TensorDef**(*base*, *exts*, *tensor: drudge.drudge.Tensor*)
Definition of a tensor.

__init__(*base*, *exts*, *tensor: drudge.drudge.Tensor*)
Initialize the tensor definition.

In the same way as the initializer for the *Tensor* class, this initializer is also unlikely to be used directly in user code. Drudge methods *Drudge.define()* and *Drudge.define_einst()* can be more convenient.

is_scalar
If the tensor defined is a scalar.

rhs
Get the right-hand-side of the definition.

rhs_terms
Gather the terms on the right-hand-side of the definition.

lhs
Get the standard left-hand-side of the definition.

base
The base of the tensor definition.

exts
The external indices.

simplify()
Simplify the tensor in the definition.

Due to the scarcity of the usefulness of keeping both the unsimplified and the simplified tensor definition, this method will mutate the tensor with its simplified form.

__eq__(*other*)
Compare two tensor definitions for equality.

Note that similar to the equality comparison of tensors, here we only compare the syntactic equality rather than the mathematical equality.

__str__()
Form simple readable string for a definition.

latex(*sep_lines=False*)
Get the latex form for the tensor definition.

The result will just be the form from *Tensor.latex()* with the RHS prepended.

Parameters **sep_lines** (*bool*) – If terms should be put into separate lines by separating them with \\.

display(*if_return=True*, *sep_lines=False*)
Display the tensor definition in interactive notebook sessions.

The parameters here all have the same meaning as in *Tensor.display()*.

act(*tensor*, *wilds=None*)
Act the definition on a tensor.

This method is the active voice version of the *Tensor.subst()* function. All appearances of the defined object in the tensor will be substituted.

__getitem__(*item*)
Get the tensor when the definition is indexed.

3.1.4 Miscellaneous utilities

drudge.**sum_**(*obj*)
Sum the values in the given iterable.

Different from the built-in summation function, here a value zero is created only when the iterator is empty. Or the summation is based on the first item in the iterable.

drudge.**prod_**(*obj*)
Product the values in the given iterable.

Similar to the summation utility function, here the initial value for the reduction is the first element. Different from the summation, here a integer unity will be returned for empty iterator.

class drudge.**Stopwatch**(*print_cb=<built-in function print>*)
Utility class for printing timing information.

This class helps to timing the progression of batch jobs. It is capable of getting and formatting the elapsed wall time between consecutive steps. Note that the timing here might not be accurate to one second.

__init__(*print_cb=<built-in function print>*)
Initialize the stopwatch.

Parameters **print_cb** – The function will be called with the formatted time-stamp. By default, it will just be written to stdout.

tick()
Reset the timer.

tock(*label*, *tensor=None*)
Make a timestamp.

The formatted timestamp will be given to the callback of the current stamper.

Parameters

- **label** – The label for the current step.
- **tensor** – When a tensor is given, it will be cached, counted its number of terms. This method has this parameter since if no reduction is performed on the tensor, it might remain unevaluated inside Spark and give misleading timing information.

__weakref__
list of weak references to the object (if defined)

3.2 Support of different algebraic systems

The base system does not assume any commutation rules amongst the generators of the algebra, *ie* free algebra or tensor algebra is assumed. However, by subclassing the *Drudge* class, domain specific knowledge about the algebraic system in the problem can be given. Inside drudge, we have some algebraic systems that is already built in.

class drudge.**WickDrudge**(**args*, *wick_parallel=0*, ***kwargs*)
Drudge for Wick-style algebras.

A Wick-style algebra is an algebraic system where the commutator between any generators of the algebra is a simple scalar value. This drudge will attempt to put the vectors into normal order based on the given comparator and contractor by Wick theorem.

Normally, subclasses need to override the properties *phase*, *contractor*, and *comparator* with domain-specific knowledge.

__init__(**args*, *wick_parallel=0*, ***kwargs*)
Initialize the Wick drudge.

This level just have one option to handle, the parallelism option.

wick_parallel
Get the Wick parallelism level.

contractor
Get the contractor for the algebraic system.

The contractor is going to be called with two vectors to return the value of their contraction.

phase
Get the phase for the commutation rule.

The phase should be a constant defining the phase of the commutation rule.

comparator
Get the comparator for the canonicalized vectors.

The normal ordering operation will be performed according to this comparator. It will be called with two **canonicalized vectors** for a boolean value. True should be returned if the first given vector is less than the second vector. The two vectors will be attempted to be transposed when False is returned.

normal_order(*terms: pyspark.rdd.RDD*, ***kwargs*)
Normal order the terms according to generalized Wick theorem.

The actual expansion is based on the information given in the subclasses by the abstract properties.

class drudge.**FockDrudge**(**args*, *exch=-1*, ***kwargs*)
Drudge for doing fermion/boson operator algebra on Fock spaces.

This is the general base class for drudges working on fermion/boson operator algebras. Here general methods are defined for working on these algebraic systems, but no problem specific information, like ranges or operator base, is defined. Generally, operators for Fock space problems has either *CR* or *AN* as the first index to give their creation or annihilation character.

To customize the details of the commutation rules, properties *op_parser* and *ancr_contractor* can be overridden.

__init__(**args*, *exch=-1*, ***kwargs*)
Initialize the drudge.

Parameters **exch** (*{1, -1}*) – The exchange symmetry for the Fock space. Constants *FERMI* and *BOSE* can be used.

contractor
Get the contractor for the algebra.

The operations are read here on-the-fly so that possibly customized behaviour from the subclasses can be read.

phase
Get the phase for the commutation rules.

comparator
Get the comparator for the normal ordering operation.

vec_colour
Get the vector colour evaluator.

OP_PARSER
alias of Callable

op_parser
Get the parser for field operators.

The result should be a callable taking an vector and return a triple of operator base, operator character, and the actual indices to the operator. This can be helpful for cases where the interpretation of the operators needs to be tweeked.

ANCR_CONTRACTOR
alias of `Callable`

ancr_contractor
Get the contractor for annihilation and creation operators.

In this drudge, the contraction between creation/creation, annihilation/annihilation, and creation/annihilation operators are fixed. By this property, a callable for contracting annihilation operators with a creation operator can be given. It will be called with the base and indices (excluding the character) of the annihilation operators and the base and indices of the creation operator. A simple SymPy expression is expected in the result.

By default, the result will be a simple delta.

eval_vev(*tensor: drudge.drudge.Tensor*, *contractor*)
Evaluate vacuum expectation value.

The contractor needs to be given as a callable accepting two operators. And this function is also set as a tensor method by the same name.

eval_phys_vev(*tensor: drudge.drudge.Tensor*)
Evaluate expectation value with respect to the physical vacuum.

Here the contractor from normal-ordering will be used. And this function is also set as a tensor method by the same name.

set_n_body_base(*base: sympy.tensor.indexed.IndexedBase*, *n_body: int*)
Set an indexed base as an n-body interaction.

The symmetry of an n-body interaction has full permutation symmetry among the corresponding slots in the first and second half.

When the body count if less than two, no symmetry is added. And the added symmetry is for the given valence only.

set_dbbar_base(*base: sympy.tensor.indexed.IndexedBase*, *n_body: int*, *n_body2=None*)
Set an indexed base as a double-bar interaction.

A double barred interaction has full permutation symmetry among its first half of slots and its second half individually. For fermion field, the permutation is assumed to be anti-commutative.

The size of the second half can be given by another optional argument, or it is assumed to have the same size as the first half. It can also be zero, which gives one chunk of symmetric slots only.

drudge.**CR**
The label for creation operators.

drudge.**AN**
The label for annihilation operators.

drudge.**FERMI**
The label for fermion exchange symmetry.

drudge.**BOSE**
The label for boson exchange symmetry.

3.3 Direct support of different problems

In addition to the algebraic rules, more domain specific knowledge can be added to drudge subclasses for the convenience of working on specific problems. In these `Drudge` subclasses, we have not only the general mathematical knowledge like commutation rules, but more detailed information about the problem as well, like some commonly used ranges, dummies.

class drudge.**GenMBDrudge**(**args*, *exch=-1*, *op_label='c'*, *orb=((Range('L'), 'abcdefghijklmnopq'),)*, *spin=()*, *one_body=t*, *two_body=u*, *dbbar=False*, ***kwargs*)

Drudge for general many-body problems.

In a general many-body problem, a state for the particle is given by a symbolic **orbital** quantum numbers for the external degrees of freedom and optionally a concrete **spin** quantum numbers for the internal states of the particles. Normally, there is just one orbital quantum number and one or no spin quantum number.

In this model, a default Hamiltonian of the model is constructed from a one-body and two-body interaction, both of them are assumed to be spin conserving.

op
The vector base for the field operators.

cr
The base for the creation operator.

an
The base for the annihilation operator.

orb_ranges
A list of all the ranges for the orbital quantum number.

spin_vals
A list of all the explicit spin values.

orig_ham
The original form of the Hamiltonian without any simplification.

ham
The simplified form of the Hamiltonian.

__init__(**args*, *exch=-1*, *op_label='c'*, *orb=((Range('L'), 'abcdefghijklmnopq'),)*, *spin=()*, *one_body=t*, *two_body=u*, *dbbar=False*, ***kwargs*)

Initialize the drudge object.

Parameters

- **exch** – The exchange symmetry of the identical particle.
- **op_label** – The label for the field operators. The creation operator will be registered in the names archive by name of this label with `_dag` appended. And the annihilation operator will be registered with a single trailing underscore.
- **orb** – An iterable of range and dummies pairs for the orbital quantum number, which is considered to be over the **direct sum** of all the ranges given. All the ranges and dummies will be registered to the names archive by `Drudge.set_dumms()`.
- **spin** – The values for the explicit spin quantum number.
- **one_body** – The indexed base for the amplitude in the one-body part of the Hamiltonian. It will also be added to the name archive.
- **two_body** – The indexed base for the two-body part of the Hamiltonian. It will also be added to the name archive.

- **dbbar** (*bool*) – If the two-body part of the Hamiltonian is double-bared.

class drudge.**PartHoleDrudge**(**args*, *op_label='c'*, *part_orb=(Range('V', 0, nv), (a, b, c, d, e, f, g, h, a0, a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12, a13, a14, a15, a16, a17, a18, a19, a20, a21, a22, a23, a24, a25, a26, a27, a28, a29, a30, a31, a32, a33, a34, a35, a36, a37, a38, a39, a40, a41, a42, a43, a44, a45, a46, a47, a48, a49))*, *hole_orb=(Range('O', 0, no), (i, j, k, l, m, n, p, q, i0, i1, i2, i3, i4, i5, i6, i7, i8, i9, i10, i11, i12, i13, i14, i15, i16, i17, i18, i19, i20, i21, i22, i23, i24, i25, i26, i27, i28, i29, i30, i31, i32, i33, i34, i35, i36, i37, i38, i39, i40, i41, i42, i43, i44, i45, i46, i47, i48, i49))*, *spin=()*, *one_body=t*, *two_body=u*, *fock=f*, *dbbar=True*, ***kwargs*)

Drudge for the particle-hole problems.

This is a shallow subclass of `GenMBDrudge` for the particle-hole problems. It contains different forms of the Hamiltonian.

orig_ham
The original form of the Hamiltonian, written in terms of bare one-body and two-body interaction tensors without normal-ordering with respect to the Fermion vacuum.

full_ham
The full form of the Hamiltonian in terms of the bare interaction tensors, normal-ordered with respect to the Fermi vacuum.

ham_energy
The zero energy inside the full Hamiltonian.

one_body_ham
The one-body part of the full Hamiltonian, written in terms of the bare interaction tensors.

ham
The most frequently used form of the Hamiltonian, written in terms of Fock matrix and the two-body interaction tensor.

__init__(**args*, *op_label='c'*, *part_orb=(Range('V', 0, nv), (a, b, c, d, e, f, g, h, a0, a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12, a13, a14, a15, a16, a17, a18, a19, a20, a21, a22, a23, a24, a25, a26, a27, a28, a29, a30, a31, a32, a33, a34, a35, a36, a37, a38, a39, a40, a41, a42, a43, a44, a45, a46, a47, a48, a49))*, *hole_orb=(Range('O', 0, no), (i, j, k, l, m, n, p, q, i0, i1, i2, i3, i4, i5, i6, i7, i8, i9, i10, i11, i12, i13, i14, i15, i16, i17, i18, i19, i20, i21, i22, i23, i24, i25, i26, i27, i28, i29, i30, i31, i32, i33, i34, i35, i36, i37, i38, i39, i40, i41, i42, i43, i44, i45, i46, i47, i48, i49))*, *spin=()*, *one_body=t*, *two_body=u*, *fock=f*, *dbbar=True*, ***kwargs*)

Initialize the particle-hole drudge.

op_parser
Get the special operator parser for particle-hole problems.

Here when the first index to the operator is resolved to be a hole state, the creation/annihilation character of the operator will be flipped.

eval_fermi_vev(*tensor: drudge.drudge.Tensor*)
Evaluate expectation value with respect to Fermi vacuum.

This is just an alias to the actual `FockDrudge.eval_phys_vev()` method to avoid confusion about the terminology in particle-hole problems. And it is set as a tensor method by the same name.

parse_tce(*tce_out: str, cc_bases: typing.Mapping[int, sympy.tensor.indexed.IndexedBase]*)
Parse TCE output into a tensor.

The CC amplitude bases should be given as a dictionary mapping from the excitation order to the actual base.

drudge.**UP**
The symbol for spin up.

drudge.**DOWN**
The symbolic value for spin down.

class drudge.**SpinOneHalfGenDrudge**(**args*, ***kwargs*)
Drudge for many-body problems of particles with explicit 1/2 spin.

This is just a shallow subclass of the drudge for general many-body problems, with exchange set to fermi and has explicit spin values of *UP* and *DOWN*.

__init__(**args*, ***kwargs*)
Initialize the drudge object.

class drudge.**SpinOneHalfPartHoleDrudge**(**args*, *part_orb=(Range('V')*, *(a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *a0*, *a1*, *a2*, *a3*, *a4*, *a5*, *a6*, *a7*, *a8*, *a9*, *a10*, *a11*, *a12*, *a13*, *a14*, *a15*, *a16*, *a17*, *a18*, *a19*, *a20*, *a21*, *a22*, *a23*, *a24*, *a25*, *a26*, *a27*, *a28*, *a29*, *a30*, *a31*, *a32*, *a33*, *a34*, *a35*, *a36*, *a37*, *a38*, *a39*, *a40*, *a41*, *a42*, *a43*, *a44*, *a45*, *a46*, *a47*, *a48*, *a49*, *beta*, *gamma))*, *hole_orb=(Range('O')*, *(i*, *j*, *k*, *l*, *m*, *n*, *p*, *q*, *i0*, *i1*, *i2*, *i3*, *i4*, *i5*, *i6*, *i7*, *i8*, *i9*, *i10*, *i11*, *i12*, *i13*, *i14*, *i15*, *i16*, *i17*, *i18*, *i19*, *i20*, *i21*, *i22*, *i23*, *i24*, *i25*, *i26*, *i27*, *i28*, *i29*, *i30*, *i31*, *i32*, *i33*, *i34*, *i35*, *i36*, *i37*, *i38*, *i39*, *i40*, *i41*, *i42*, *i43*, *i44*, *i45*, *i46*, *i47*, *i48*, *i49*, *u*, *v))*, ***kwargs*)
Drudge for the particle-hole problems with explicit one-half spin.

This is a shallow subclass over the general particle-hole drudge without explicit spin. The spin values are given explicitly to be *UP* and *DOWN* and the double-bar of the two-body interaction is disabled. And some additional dummies traditional in the field are also added.

__init__(**args*, *part_orb=(Range('V')*, *(a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *a0*, *a1*, *a2*, *a3*, *a4*, *a5*, *a6*, *a7*, *a8*, *a9*, *a10*, *a11*, *a12*, *a13*, *a14*, *a15*, *a16*, *a17*, *a18*, *a19*, *a20*, *a21*, *a22*, *a23*, *a24*, *a25*, *a26*, *a27*, *a28*, *a29*, *a30*, *a31*, *a32*, *a33*, *a34*, *a35*, *a36*, *a37*, *a38*, *a39*, *a40*, *a41*, *a42*, *a43*, *a44*, *a45*, *a46*, *a47*, *a48*, *a49*, *beta*, *gamma))*, *hole_orb=(Range('O')*, *(i*, *j*, *k*, *l*, *m*, *n*, *p*, *q*, *i0*, *i1*, *i2*, *i3*, *i4*, *i5*, *i6*, *i7*, *i8*, *i9*, *i10*, *i11*, *i12*, *i13*, *i14*, *i15*, *i16*, *i17*, *i18*, *i19*, *i20*, *i21*, *i22*, *i23*, *i24*, *i25*, *i26*, *i27*, *i28*, *i29*, *i30*, *i31*, *i32*, *i33*, *i34*, *i35*, *i36*, *i37*, *i38*, *i39*, *i40*, *i41*, *i42*, *i43*, *i44*, *i45*, *i46*, *i47*, *i48*, *i49*, *u*, *v))*, ***kwargs*)
Initialize the particle-hole drudge.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

A

B

C

S

T

U

V

W