drudge Documentation

Release 0.8.0

Jinmo Zhao and Gustavo E Scuseria

Sep 29, 2017

CONTENTS:

Acknowledgment

This work was supported as part of the Center for the Computational Design of Functional Layered Materials, an Energy Frontier Research Center funded by the U.S. Department of Energy, Office of Science, Basic Energy Sciences under Award DE-SC0012575.

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 3.6 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 2.2 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 in spark-in-slurm can be helpful. Throughout the entire documentation, `SparkContext()` will be used to create a minimal Spark context for demonstration purpose. The actual context should be created according to your site.

As an experimental project, the documentation can be outdated, incomplete, incorrect, or have a lot of bad formatting.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

CHAPTER

TWO

RELEASE HISTORY

2.1 0.2.0

In this release, major revisions and enhancements have been made to the drudge package. Primarily,

1. TensorDef class are added to support better substitution and direct indexing.
2. Drudge.einst and Drudge.sum has been revised to work with existing tensors.
3. Variable-valence symmetry are supported and used in many-body drudges.
4. The timing facility has been renamed to Matlab convention.

Since the code has not yet been widely deployed, a few changes might break backward compatibility,

1. act method of tensor is removed.
2. set_name method of drudge is updated to a new interface.
3. Timing facility is renamed.

2.2 0.3.0

In this release, in addition to various performance optimization, small bug fixing, and some internal cleaning-up, the most significant enhancement is the handling of algebraic systems other than the fermion/boson algebra. Currently, Clifford algebra and $\mathfrak{su}(2)$ algebra are supported. Other algebraic systems should be able to be added with ease.

Also the documentation has been expanded, especially with more examples in the API reference guide. Also the utilities are added for convenient pickling of tensors.

2.3 0.4.0

This is mainly a bug-fix release.

- The incomplete simplification problem for Fock drudge is fixed. Thanks to Ethan Qiu for pointing this out.
- The incompetent script for running Spark inside SLURM is removed and got improved in the separate *spark-in-slurm* project.
- Spin-1/2 particle-hole drudge now has explicit orbital range sizes. Thanks to Roman Schutski for fixing this.

2.4 0.5.0

The most prominent change of this release is the addition of the `RestrictedPartHoleDrudge`. Compared with the `SpinOneHalfPartHoleDrudge`, this class is less flexible but can be orders of magnitude faster. Note that no backward compatibility is broken by this. All previous script should still run fine. For problems originally written in terms of the unitary group generators, switching to this new drudge can be really easy. All these needs to be done is to remove your own definitions of the unitary group generators, and use the generator `e_` provided by the drudge instead. For instance, as in the patch for the RCCSD example.

With such simple changes, the internals of the evaluations will be switched to the new scheme and the code will be significantly faster.

In addition of this change, there are also some other revisions,

- A bug in Spark has been circumvented in tensor rewriting. Thanks to Ethan Qiu for pointing this out.
- Tensor definition now allow generic external indices without explicit range.
- A new simplification heuristics is added to simplify summations whose dummy is not actually involved in the tensor.
- Add total timing support in `Stopwatch`.
- Optimize summation discovery based on Einstein summation convention.

2.5 0.6.0

In the previous release, `RestrictedPartHoleDrudge` is narrower in scope than the `SpinOneHalfPartHoleDrudge`. After some internal revision, now all problems for `SpinOneHalfPartHoleDrudge` should be able to be handled with `RestrictedPartHoleDrudge`. For expressions with many terms coming from concrete spin summation, significant speed up can be brought.

To update to the new drudge,

- When your theory is already based on unitary group generators, just remove your definition and use the definition from the drudge.
- When your theory is not written in terms of unitary group generators, it is strongly advised that your theory is rewritten in terms of them. If it really cannot, use symbolic summation over spin values as much as possible.
- Even when the spin values are never summed abstractly, simply change the drudge to this new one might still be beneficial.

2.6 0.7.0

This releases adds some convenience operations defined for tensorial objects, like direct negation and division by scalars. For instance, to negate a tensor `t`, now we can just write `-t` instead of the more cumbersome `-1 * t`. Similarly, to divide it by two, we can now just write `t / 2`, compared with the previous syntax of:

```
t * sympy.Rational(1, 2).
```

The examples for CC theories are also updated for the new syntax. And some other convenience enhancements are added. For instance, now LaTeX or pdf report can directly be generated, with the structure of the report more flexible and tunable. And the LaTeX formatting is enhanced with more options to fine tune its behaviour. Also `memoize` method is added to the `Drudge` class for the convenience of caching intermediate results.

Also multiple bugs are fixed,

- Now the code no longer crashes when a deltas contains no dummy.
- Incorrectness from highly cyclic delta resolution result is fixed.
- The dummies in simplified result for particle-hole problems are made conventional.
- Simplification of `TensorDef` now resets external dummies as well.
- Core dependency `libcanon` is updated to 0.1.2 such that the erroneous requirement of twice simplification for some complex expressions is fixed.
- A PySpark bug in `reduce` is circumvented by doing boolean reduction locally.

2.7 0.8.0

The primary highlight of this research is the introduction of drudge scripts, which is a convenient and flexible domain-specific language for doing symbolic computations in drudge. This could make drudge a lot more convenient for simple tasks and more accessible for new comers, especially those unfamiliar with the Python language. Drudge scripts can be executed either by using `exec_drs` method of the Drudge class or using drudge as the main program.

Motivated by the drudge script, the tensor definitions are made more convenient to use even in the normal Python interface. Now tensor definitions subclasses the tensor class. So all arithmetic operations are automatically available. And they can be more conveniently created by the new `Drudge.def_` method and added to the name archive by the `Drudge.set_name` method.

Also the LaTeX formatting has been improved with the option to suppress summations and the capability of using the LaTeX `breqn` package to automatically format long terms with a lot of factors. Also the LaTeX printing of tensors whose base is parsed by SymPy to have a subscript is fixed. Previously we get double subscripts for a base, which crashes both the original TeX and MathJAX.

CHAPTER

THREE

DRUDGE TUTORIAL FOR BEGINNERS

3.1 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

```
>>> from pyspark import SparkContext
>>> spark_ctx = SparkContext('local[*]', 'drudge-tutorial')
```

For using Spark in cluster computing environment, please refer to the Spark documentation and setting of your cluster. With the spark context created, we can make the main entry point for drudge,

```
>>> import drudge
>>> dr = drudge.Drudge(spark_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.

3.2 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. Also based on it, the subclass *CliffordDrudge* is capable of treating all kinds of Clifford algebras, like geometric algebra, Pauli matrices, Dirac matrices, and Majorana fermion operators. For algebraic systems where the commutator is not always a scalar, the abstract base class *GenQuadDrudge* can be used for basically all kinds of commutation rules. For instance, its subclass *SU2LatticeDrudge* can be used for $\mathfrak{su}(2)$ algebra in Cartan-Weyl form.

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(spark_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.

3.3 Drudge scripts

For maximum flexibility, drudge has been designed to be a Python library from the beginning. However, in a lot of cases, like for small tasks or for users unfamiliar with the Python language or the Spark environment, a domain-specific language capable of making simple tasks simple can be desired. Drudge script is such a language for this purpose.

A drudge script is essentially a Python script heavily tweaked to be executed inside a special environment. So all Python lexicographical and syntactical rules apply. For a technical description of the pre-processing and execution drudge scripts, please see *`Drudge.exec_drs()`*. To execute a drudge script, we first need a *`Drudge`* object, such that the domain specific information about the current problem can be available. For this, we can either have a normal Python script, where a Drudge object is created with its *`Drudge.exec_drs()`* called with the source code for the drudge script, and execute it normally as Python scripts. Or drudge can also be used as the main program, either by `python3 -m drudge` or `drudge`. Then two files needs to be given as arguments. The first one is a configuration script, which is a normal Python script with a Drudge object assigned to a special variable `DRUDGE`. Then this Drudge object will be used for the execution of the actual drudge script given in the second argument.

As an example illustrating the basic principles and ease of drudge scripts, we assume that we are working on a drudge with a single range registered in the name archive as `R`. To create a symbolic definition of a matrix as a product of two matrices, suppose the drudge object can be accessed by a variable `dr`, we need to write something like:

```
p = dr.names
r = sympy.IndexedBase('r')
x = sympy.IndexedBase('x')
y = sympy.IndexedBase('y')
i, j, k = sympy.symbols('i j k')
def_ = dr.define(r, (i, p.R), (j, p.R), dr.sum((k, p.R), x[i, k] * y[k, j]))
```

which can be quite cumbersome for such a simple task. Suppose the drudge has a resolver capable of resolving any index to the range, we can write:

```
r = sympy.IndexedBase('r')
x = sympy.IndexedBase('x')
y = sympy.IndexedBase('y')
i, j, k = sympy.symbols('i j k')
def_ = dr.define_einst(r[i, j], x[i, k] * y[k, j])
```

which although is simplified a lot, still contains quite a lot of noise. Because of the Python execution model and scoping rules, the indexed bases and symbols must be explicitly created before they can be used.

Inside a drudge script, names in the name archive, all methods of the current drudge object, as well as names from the drudge, gristmill (if installed), and the SymPy package can directly be used without any qualification. More importantly, Symbol objects and IndexedBase objects are no longer needed to be explicitly created. All undefined names will be resolved as an atomic symbol, which can be construed as both a SymPy symbol and a SymPy IndexedBase. With these, the above definition can be simplified into:

```
def_ = define_einst(r[i, j], x[i, k] * y[k, j])
```

Due to the ubiquity of tensor definitions in common drudge tasks, a special operator <<= (Python left-shift augmented assignment operator) is introduced for the making definitions. With this, the above definition can be written as:

```
r[i, j] <<= sum((k, R), x[i, k] * y[k, j])
```

which makes the definition and put the definition in the name archive by *`Drudge.set_name()`*. So by default, the definition is put into the name archive under name `r` as a *`TensorDef`* object, and the base of the definition is put under name `_r`. Since names in the name archive do not need to be qualified in drudge scripts:

```
sum((k, R), r[i, k] * r[k, j])
```

directly gives us the chain product $\mathbf{XYXY}$. And symbolic references to the `r` tensor without the concrete definition substituted in can still be made by using `_r`, like:

```
s = sum((k, R), _r[i, k] * _r[k, j])
```

which gives us the product $\mathbf{RR}$. For this, the actual definition can be substituted explicitly when desired, for example, by:

```
s.subst(r)
```

which gives us $\mathbf{XYXY}$.

Note that the definition by <<= is made by using the *`Drudge.def_()`* method. As a result, when the drudge property *`Drudge.default_einst()`* is set, Einstein summation convention is going to be automatically applied to the right-hand side. So we can simply write:

```
r[i, j] <<= x[i, k] * y[k, j]
```

when the ranges of i, j, k can be resolved by the drudge.

In cases where tainting of the global name archive is undesired for a tensor definition, we can use the <= operator, which simply returns the definition object without adding it to the name archive. For instance, to store the tensor definition in a variable `def_`, we can use:

```
def_ = r[i, j] <= x[i, k] * y[k, j]
```

This can be useful in functions inside drudge scripts.

Additionally, drudges could have more functions specifically to be used inside drudge scripts. For instance, in the base *`Drudge`* class, we have a simple constructor `S`, for converting strings to the special kind of symbols that can be indexed and used in <<= in drudge scripts. Also have `sum_` for the actual Python built-in `sum` function, which is shadowed by the *`Drudge.sum()`* method.

For the taste of users without much object-oriented programming, inside drudge scripts, method calling like `obj.meth(args)` can also be written as `meth(obj, args)`. For instance, for a tensor `tensor`:

```
simplify(tensor)
```

is equivalent to:

```
tensor.simplify()
```

Attribute access can be done in the same way, for instance,:

```
n_terms(tensor)
```

is equivalent to:

```
tensor.n_terms
```

Note that a caveat of this syntactic sugar is that the method name cannot be defined to be anything else before the calling. For instance,:

```
n_terms = 10
n_terms(tensor)
```

does not work, since `n_terms` is already defined to the integer 10, thus cannot be called any more. Another caveat is that static methods cannot be called in this way, which fortunately does not appear a lot in common usages of drudge.

For the convenience of symbolic computation, all integer literals inside drudge scripts are automatically resolved to SymPy integer values, rather than the built-in integer values. As a result, we can directly write:

```
1 / 2
```

for the rational value of one-half, without having to worry about the truncation or degradation to finite-precision floating-point numbers for Python integers. To access built-in integers, which is normally unnecessary, we can explicitly write something like `int(1)`.

For convenience of users, some drudge functions has got slightly different behaviour inside drudge scripts. For instance, the *`Tensor.simplify()`* method will eagerly compute the result and repartition the terms among the workers. And tensors also have more readable string representation inside drudge scripts.

3.4 Examples on real-world applications

In this tutorial, some simple examples are run directly inside a Python interpreter. Actually drudge is designed to work inside Jupyter notebooks as well. 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.

For drudge scripts, we have two example scripts both deriving the classical CCD theory. Both of them is based on the following configuration script `conf_ph.py`,

```
"""Configures a simple drudge for particle-hole model."""

from dummy_spark import SparkContext
from drudge import PartHoleDrudge

ctx = SparkContext()
dr = PartHoleDrudge(ctx)
```

```
dr.full_simplify = False

DRUDGE = dr
```

Here we only set a simple *PartHoleDrudge* without much modification. To illustrate the most basic usage of drudge scripts, we have example `ccd.drs`,

```
# A simple example on using drudge script for CCD theory
#
# In this example, the most basic aspects of using drudge scripts is
# illustrated.  It should be understandable for new-comers without much
# previous Python background.

# Define the cluster excitation operator.  Note that we need to inform the
# drudge that the t amplitude tensor has the double bar symmetry of
# t_{abij} = -t_{baij} = -t_{abji} = t_{baji}
set_dbbar_base(t, 2)

# Einstein summation convention can be used for easy tensor creation.
t2 = einst(
    t[a, b, i, j] * c_dag[a] * c_dag[b] * c_[j] * c_[i] / 4
)

# Get the similarity-transformed Hamiltonian.  Note that ``|`` operator
# computes the commutator between operators.
c0 = ham
c1 = simplify(c0 | t2)
c2 = simplify(c1 | t2)
c3 = simplify(c2 | t2)
c4 = simplify(c3 | t2)
h_bar = simplify(
    c0 + c1 + (1/2) * c2 + (1/6) * c3 + (1/24) * c4
)

print('Similarity-transformed Hamiltonian has {} terms'.format(
    n_terms(h_bar)
))

# Derive the working equations by projection.
en_eqn = simplify(eval_fermi_vev(h_bar))
proj = c_dag[i] * c_dag[j] * c_[b] * c_[a]
t2_eqn = simplify(eval_fermi_vev(proj * h_bar))

print('Working equation derived!')

with report('ccd.html', 'CCD theory') as rep:
    rep.add('Energy equation', en_eqn)
    rep.add('Doubles amplitude equation', t2_eqn)
```

With the comment described in the above script, we can see that drudge script can bare a lot of resemblance to the mathematical notation. To make a derivation of the many-body theory, we basically just use the operators like `+`, `*`, and `|` to do arithmetic operations on the tensors and use `simplify` to get the result simplified.

For another more advanced example, we have the `ccd_adv.drs` script,

```
# An advanced example on using drudge script for CCD theory
#
# In this example, it is emphasized that drudge scripts are just Python scripts
# with special execution.  So all Python constructions can be used for our
# convenience.  At the same time, by using drudge scripts, we can have all the
# syntactical sugar for making symbolic computation easy.
#

set_dbbar_base(t, 2)
t2 = einst(
    t[a, b, i, j] * c_dag[a] * c_dag[b] * c_[j] * c_[i] / 4
)

def compute_h_bar():
    """Compute the similarity transformed Hamiltonian."""
    # Here we use a Python loop to get the nested commutators.
    curr = ham
    h_bar = ham
    for order in range(0, 4):
        curr = simplify(curr | t2) / (order + 1)
        h_bar += curr
    return simplify(h_bar)

# By using the `memoise` function, the result can be automatically dumped into
# the given pickle file, and read from it if it is already available.  This can
# be convenient for large multi-step jobs.
h_bar = memoize(compute_h_bar, 'h_bar.pickle')
print('H-bar has {} terms'.format(n_terms(h_bar)))

# Derive the working equations by projection.  Here we make them into tensor
# definition with explicit left-hand side, so that they can be used for
# optimization.
e <<= simplify(eval_fermi_vev(h_bar))
proj = c_dag[i] * c_dag[j] * c_[b] * c_[a]
r2[a, b, i, j] <<= simplify(eval_fermi_vev(proj * h_bar))

print('Working equation derived!')

# When the gristmill package is also installed, the evaluation of the working
# equations can also be optimized with it.
eval_seq = optimize(
    [e, r2], substs={no: 1000, nv: 10000}
)

# In addition to HTML report, we can also have LaTeX report.  Note that the
# report can be structured into sections with descriptions.  For LaTeX output,
# the `dmath` environment from the `breqn` package can be used to break lines
# automatically inside large equations.

# Long descriptions of contents can be put in Python multi-line strings.
opt_description = """
The optimization is based on 1000 occupied orbitals and 10000 virtual orbitals,
which should be representative of common problems for CCD theory.
"""

with report('ccd.tex', 'CCD theory') as rep:
    rep.add(title='Working equations')
    rep.add(content=e, description='The energy equation')
    rep.add(content=r2, description='Doubles amplitude equation', env='dmath')
    rep.add(title='Optimized evaluation', description=opt_description)
    for step in eval_seq:
        rep.add(content=step, env='dmath')
```

In the example `ccd.drs`, it is attempted to be emphasized that drudge scripts are very similar to common mathematical notation and should be easy to get started. In this `ccd_adv.drs` example, the power and flexibility of drudge scripts being actually Python scripts is emphasized. Foremost, rather than spelling each order of commutation out, here the similarity-transformed Hamiltonian $\bar{\mathbf{H}}$ is computed by using a Python loop. This can be helpful for repetitive tasks. Also the computation of $\bar{\mathbf{H}}$ is put inside a function. Being able to define and execute functions makes it easy to reuse code inside drudge scripts. Here, the function is given to the *`Drudge.memoize()`* function. So its result is automatically dumped into the given pickle file. When the file is already there, the result will be directly read and used with the execution of the function skipped. This can be helpful for large multi-step jobs.

Note that <<= is used to make the working equations as tensor definitions of class *`TensorDef`*. In drudge scripts,:

```
variable = tensor
```

assigns the tensor `tensor` to the variable `variable`. The variable is a normal Python variable and works in the normal Python way. And the tensor is just a static expression of its mathematical content, with all the free symbols being free. At the same time,:

```
lhs <<= tensor
```

defines the `lhs` as the tensor, with the definition pushed into the name archive of the drudge. By using *`TensorDef`* objects, we also have a left-hand side, which enables the accompanying gristmill package to optimize the evaluation of the entire array by its advanced algorithms.

For the result, here they are written into a very structured LaTeX output, which can be easily compiled into PDF files. Note that by using the *`Report.add()`* function with different arguments, we can create structured report with sections and descriptions for the equations.

3.5 Note about importing drudge

In this tutorial, `import drudge` and `import sympy` is used and we need to give fully-qualified name to refer to objects in them. Normally, it can be convenient to use `from drudge import *` to import everything from drudge. For these cases, it needs to be careful that the importation of all objects from drudge needs to follow the importation of all objects from SymPy, or the SymPy `Range` class will shallow the actual class for symbolic range in drudge.

CHAPTER

FOUR

DRUDGE API REFERENCE GUIDE

4.1 Base drudge system

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

4.1.1 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.

This is a convenience wrapper over `get_amp_factors()` for the case of no special additional symbols.

get_amp_factors(**special_symbs*)

Get the factors in the amplitude and the coefficient.

The indexed factors and factors involving dummies or the symbols in the given special symbols set 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=<function Term.<lambda>>*, *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 the default function of 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*, *add_substs=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.

simplify_sums()

Simplify the summations in the term.

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.

canon4normal(*symms*)

Canonicalize the term for normal-ordering.

This is the preparation task for normal ordering. The term will be canonicalized with all the vectors considered the same. And the dummies will be reset internally according to the summation list.

has_base(*base*)

Test if the given base is present in the current term.

4.1.2 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.

4.1.3 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.

default_einst
If `def_()` takes Einstein convention.

This property tunes the behaviour of `def_()`. When it is set, the Einstein summation convention is always assumed for the right-hand side for that function.

form_base_name(*tensor_def: drudge.drudge.TensorDef*) → typing.Union[str, NoneType]
Form the name for the base to use for tensor definitions.

This method is called by `set_name()` to get a formatted string for the base of the tensor definition, which is to be used as the name for the base in the name archive. `None` can be returned to stop the base from being added.

By default, an underscore is put in front of the string form of the base.

form_def_name(*tensor_def: drudge.drudge.TensorDef*) → typing.Union[str, NoneType]
Form the name for a tensor definition in name archive.

The result will be used by `set_name()` as the name of the tensor definition itself in the name archive. By default, it is set just to be plain string form of the base of the definition.

set_name(**args*, ***kwargs*)
Set objects 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. Special treatment is given to tensor definitions, the base and and definition itself will be added under names given by the methods `form_base_name()`, and `form_def_name()`.

For keyword arguments, the keyword will be used for the name.

unset_name(**args*, ***kwargs*)
Unset names from name archive.

This method is mostly used to undo the effect of `set_name()`. Here, names that are not actually present in the name archive will be skipped without error.

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. Their label can be used as well.
- **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.

add_default_resolver(*range_*)
Add a default resolver.

The default resolver will resolve *any* expression to the given range. Note that all later resolvers will not be invoked at all after this resolver is added.

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.

For instance, mostly commonly, we can create a tensor by having simple summations over symbolic ranges,

```
>>> dr = Drudge(SparkContext())
>>> r = Range('R')
>>> a = Symbol('a')
>>> b = Symbol('b')
>>> x = IndexedBase('x')
>>> v = Vec('v')
>>> tensor = dr.sum((a, r), (b, r), x[a, b] * v[a] * v[b])
>>> str(tensor)
'sum_{a, b} x[a, b] * v[a] * v[b]'
```

And we can also give multiple symbolic ranges for a single dummy to sum over all of them,

```
>>> s = Range('S')
>>> tensor = dr.sum((a, r, s), x[a] * v[a])
>>> print(str(tensor))
sum_{a} x[a] * v[a]
 + sum_{a} x[a] * v[a]
```

When the objects to sum over are not symbolic ranges, we are in the concrete summation mode, for instance,

```
>>> tensor = dr.sum((a, 1, 2), x[a] * v[a])
>>> print(str(tensor))
x[1] * v[1]
 + x[2] * v[2]
```

The concrete and symbolic summation mode can be put together freely in the same summation,

```
>>> tensor = dr.sum((a, r, s), (b, 1, 2), x[b, a] * v[a])
>>> print(str(tensor))
sum_{a} x[1, a] * v[a]
 + sum_{a} x[2, a] * v[a]
 + sum_{a} x[1, a] * v[a]
 + sum_{a} x[2, a] * v[a]
```

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. For instance,

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

where we have used summation with only summand (no sums) to create simple tensor of only one term without any summation.

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.

For instance, we can have the following fairly conventional Einstein form,

```
>>> dr = Drudge(SparkContext())
>>> r = Range('R')
>>> a, b, c = dr.set_dumms(r, symbols('a b c'))
>>> dr.add_resolver_for_dumms()
>>> x = IndexedBase('x')
>>> tensor = dr.einst(x[a, b] * x[b, c])
>>> str(tensor)
'sum_{b} x[a, b]*x[b, c]'
```

However, when a dummy is not in the most conventional form, the summations cannot be automatically added. For instance,

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

`b` is not summed over since it is repeated three times. Note also that the symbol must be able to be resolved its range for it to be summed automatically.

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.

def_ (**args*) → drudge.drudge.TensorDef
Make a tensor definition according to convention set in drudge.

This method is a convenient utility for making tensor definitions. Basically it calls `define()` or `define_einst()` according to the value of the property `default_einst()`.

It is also the operations used for tensor definition operations inside drudge scripts.

format_latex (*inp, sep_lines=False, align_terms=False, proc=None, no_sum=False, scalar_mul=''*)
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`.

Parameters

- **inp** – The input tensor or tensor definition.
- **sep_lines** – If terms should be put into separate lines by separating them with `\\`.
- **align_terms** – If `&` is going to be prepended to each term to have them aligned. This option is intended for cases where the LaTeX form is going to be put inside environments supporting alignment.
- **proc** – A callable to be called with the string of the original LaTeX formatting of each of the terms to return a processed final form. The callable is also going to be given keyword arguments `term` for the actual tensor term and `idx` for the index of the term within the tensor.
- **no_sum** (*bool*) – If summation is going to be suppressed in the printing, useful for cases where a convention, like the Einstein's, exists for the summations.
- **scalar_mul** (*str*) – The text for scalar multiplication. By default, scalar multiplication is just rendered as juxtaposition. When a string is given for this argument, it is going to be placed between scalar factors and between the amplitude and the vectors. In LaTeX output of tensors with terms with many factors, special command `\invismult` can be used, which just makes a small space but enables the factors to be automatically split by the `breqn` package.

report (*filename*, *title*)

Make a report for results.

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

Parameters

- **filename** (*str*) – The name of the report file, whose extension gives the type of the report. Currently, `.html` gives reports in HTML format, where the `MathJAX` library is used for rendering the math. `.tex` gives reports in LaTeX format, while `.pdf` will automatically compile the LaTeX source by program `pdflatex` in the path. Normally for LaTeX output, finer tuning of the display environment in *Report.add()* is required, especially for long equations.
- **title** – The title to be printed in the report.

Examples

```
>>> dr = Drudge(SparkContext())
>>> tensor = dr.sum(IndexedBase('x')[Symbol('a')])
>>> with dr.report('report.html', 'A simple tensor') as report:
...     report.add('Simple tensor', tensor)
```

pickle_env ()

Prepare the environment for unpickling contents with tensors.

Pickled contents containing tensors cannot be directly unpickled by the functions from the pickle module directly. They should be used within the context created by this function. Note that the content does not have to have a single tensor object. Anything containing tensor objects needs to be loaded within the context.

Warning: All tensors read within this environment will have the current drudge as their drudge. No checking is, or can be, done to make sure that the tensors are sensible for the current drudge. Normally it should be the same drudge as the drudge used for their creation be used.

Examples

```
>>> dr = Drudge(SparkContext())
>>> tensor = dr.sum(IndexedBase('x')[Symbol('a')])
>>> import pickle
>>> serialized = pickle.dumps(tensor)
>>> with dr.pickle_env():
...     res = pickle.loads(serialized)
>>> print(tensor == res)
True
```

memoize(*comput*, *filename*, *log=None*, *log_header='Memoize:'*)

Preserve/lookup result of computation into/from pickle file.

When the file with the given name exists, it will be opened and attempted to be unpickled, with the deserialized content returned and the given computation skipped. When the file is absent or does not contain valid pickle, the given computation will be performed, with the result both pickled into a file created with the given name and returned.

Parameters

- **comput** – The callable giving the computation to be performed. To be called with no arguments.
- **filename** – The name of the pickle file to read from or write to.
- **log** – The file object to write log information to. `None` if no logging is desired, `True` if they are to be written to the standard output, or any writable file object can be given.
- **log_header** – The header to be prepended to lines of the log texts.

Returns

- *The result of the computation, either read from existing file or newly*
- *computed.*

Examples

```
>>> dr = Drudge(SparkContext())
>>> res = dr.memoize(lambda: 10, 'intermediate.pickle')
>>> res
10
>>> dr.memoize(lambda: 10, 'intermediate.pickle')
10
```

Note that in the second execution, the number 10 should be read from the file rather than being computed again. Normally, rather than a trivial number, expensive intermediate results can be memoized in this way so that the script can be restarted readily.

inside_drs

If we are currently inside a drudge script.

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

`exec_drs`(*src*, *filename='<unknown>'*)
Execute the drudge script.

Drudge script are Python scripts tweaked to be executed in special environments. This domain-specific language is made for the convenience users for simple tasks, especially for users unfamiliar with Python.

Being a Python script executed inside the current interpreter, drudge script differs from normal Python scripts by

1. All integer literal are resolved into SymPy symbolic integers.
2. Global names are resolved in the order of,
   - the name archive in the current drudge,
   - the special drudge script functions in the drudge,
   - the drudge package exported names,
   - the gristmill package exported names (if installed),
   - the SymPy exported names,
   - built-in Python names.
3. All unresolved names are created as a special kind of symbolic object, which behaves basically like SymPy `Symbol`, but with differences,
   (a) They are be directly subscripted, like `IndexedBase`.
   (b) `def_as` method can be used to make a tensor definition with such symbols or its indexing on the left-hand side, the other operand on its right-hand side. The resulted definition is also added to the name archive of the drudge.
   (c) <= operator can be used similar to `def_as`, except the definition is not added to the archive. The result can be put into local variables.
4. All left-shift augmented assignment <<= operations are replaced by `def_as` method calling.
5. Some operations could have slightly different behaviour more suitable inside drudge scripts. For developers, the *`inside_drs()`* property can be used to query if the function is called inside a drudge script.

For a non-technical introduction to drudge script, please see *Drudge scripts*.

`static simplify`(*arg*, ***kwargs*)
Make simplification for both SymPy expressions and tensors.

This method is mostly designed to be used in drudge scripts. The actual simplification is dispatched based on the type of the given argument. Simple SymPy simplification for SymPy expressions, drudge simplification for drudge tensors or tensor definitions.

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.

__repr__()
Get the machine string representation.

In normal execution environment, only the memory address is displayed, since the tensor may or may not be evaluated yet. In drudge scripts, the readable string representation is returned.

__str__()
Get the string representation of the tensor.

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

latex(***kwargs*)
Get the latex form for the tensor.

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

All keyword arguments are forwarded to the `Drudge.format_latex()` method.

display(*if_return=True*, ***kwargs*)
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.
- **kwargs** – All the rest of the keyword arguments are forwarded to the `Drudge.format_latex()` method.

__getstate__()
Get the current state of the tensor.

Here we just have the local terms. Other cached information are discarded.

__setstate__(*state*)
Set the state for the new tensor.

This function reads the drudge to use from the module attribute, which is set in the `Drudge.pickle_env()` method.

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.

`simplify_sums`()
Simplify the summations in the tensor.

Currently, only bounded summations with dummies not involved in the term will be replaced by a multiplication with its size.

`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. Inside drudge scripts, it also make eager evaluation and repartition the terms among the Spark workers, with the result cached. This is for the ease of users unfamiliar with the Spark lazy execution model.

__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.

__neg__()

Negate the current tensor.

The result will be equivalent to multiplication with -1.

__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.

__truediv__(*other*)

Divide tensor by a scalar quantity.

`__rtruediv__`(*other*)
Make division over a tensor.

`subst`(*lhs*, *rhs*, *wilds=None*, *full_balance=False*, *excl=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.

Examples

For instance, we can have a very simple tensor, the outer product of the same vector,

```
>>> dr = Drudge(SparkContext())
>>> r = Range('R')
>>> a, b = dr.set_dumms(r, symbols('a b c d e f'))[:2]
>>> dr.add_default_resolver(r)
>>> x = IndexedBase('x')
>>> v = Vec('v')
>>> tensor = dr.einst(x[a] * x[b] * v[a] * v[b])
>>> str(tensor)
'sum_{a, b} x[a]*x[b] * v[a] * v[b]'
```

We can replace the indexed base by the product of a matrix with another indexed base,

```
>>> o = IndexedBase('o')
>>> y = IndexedBase('y')
>>> res = tensor.subst(x[a], dr.einst(o[a, b] * y[b]))
>>> str(res)
'sum_{a, b, c, d} y[c]*y[d]*o[a, c]*o[b, d] * v[a] * v[b]'
```

We can also make substitution on the vectors,

```
>>> w = Vec('w')
>>> res = tensor.subst(v[a], dr.einst(o[a, b] * w[b]))
>>> str(res)
'sum_{a, b, c, d} x[a]*x[b]*o[a, c]*o[b, d] * w[c] * w[d]'
```

After the substitution, we can always make a simplification, at least to make the naming of the dummies more aesthetically pleasing,

```
>>> res = res.simplify()
>>> str(res)
'sum_{a, b, c, d} x[c]*x[d]*o[c, a]*o[d, b] * w[a] * w[b]'
```

`subst_all`(*defs*, *simplify=False*, *full_balance=False*, *excl=None*)
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.

`rewrite`(*vecs*, *new_amp*)
Rewrite terms with the given vectors in terms of the new amplitude.

This method will rewrite the terms whose vector part patches the given vectors in terms of the given new amplitude. And all terms rewritten into the same form will be aggregated into a single term.

Parameters

- **`vecs`** – A vector or a product of vectors. They should be written in terms of SymPy wild symbols when they need to be matched against different actual vectors.
- **`new_amp`** – The amplitude that the matched terms should have. They are usually written in terms of the same wild symbols as the wilds in the vectors.

Returns

- *rewritten* – The tensor with the requested terms rewritten in term of the given amplitude.
- *defs* – The actual definitions of the rewritten amplitude. One for each rewritten term in the result.

`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.

A tensor definition is basically a tensor with a name. In additional to being a tensor, a tensor definition also has a left-hand side. When the tensor is zero-order, the left-hand side is simply a symbol. When it has external indices, the base and external indices for the it are both stored. Explicit storage of a left-hand side can be convenient in many cases.

`__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.

Parameters

- **`base`** – The base for the definition. It should be a *`Vec`* instance for tensors with vector part. Or it should be SymPy IndexedBase or Symbol instance for scalar tensors, depending on the presence or absence of external indices.
- **`exts`** – The iterable for external indices. They can be either symbol/range pairs for external indices with explicit range, or they can also be a plain symbol for generic definitions.
- **`tensor`** – The RHS of the definition.

`rhs`

Get the right-hand-side of the definition.

The result is the definition itself. Kept here for backward compatibility.

`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.

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

The external indices will take higher precedence over the summed indices inside the right-hand side.

__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. The left-hand side is put into consideration only for comparison with another tensor definition.

__str__ ()
Form simple readable string for a definition.

latex (***kwargs*)
Get the latex form for the tensor definition.

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

Parameters kwargs – All keyword parameters are forwarded to the `Drudge.format_latex()` method.

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

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

act (*tensor*, *wilds=None*, *full_balance=False*)
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.

__getstate__ ()
Get the current state of the definition.

__setstate__ (*state*)
Set the state for the new definition.

4.1.4 Miscellaneous utilities

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

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

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

Similar to the summation utility function `sum_()`, here the initial value for the reduction is the first element. Different from the summation, here a SymPy 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`(*total=False*)
Reset the timer.

Parameters **`total`** – If the total beginning time is going to be reset as well.

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

The formatted timestamp will be given to the callback of the current stamper. The wall time elapsed since the last `tick()` will be printed.

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.

`tock_total`()
Make a timestamp for the total time.

The total time will be the time elapsed since the **total** time was last reset.

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

class `drudge.`**`Report`**(*filename: str*, *title*)
Simple report for output drudge results.

This class helps to write symbolic results to files for batch processing jobs. It is not recommended to be used directly. Users should use the method provided by drudge class instead in `with` statements.

`__init__`(*filename: str*, *title*)
Initialize the report object.

`add`(*title=None*, *content=None*, *description=None*, *env='['*, ***kwargs*)
Add a section to the result.

Parameters

- **`title`** – The title of the equation. It will be used as a section header. When it is given as a None, the section header will not be added.
- **`content`** – The actual tensor or tensor definition to be printed. It can be given as a None to skip any equation rendering.
- **`description`** – A verbal description of the content. It will be typeset before the actual equation as normal text. A None value will cause it to be suppressed.
- **`env`** – The environment to put the equation in. A value of `'['` will use `\[` and `\]` as the deliminator of the math environment. Other values will be put inside the common `\begin{}` and `\end{}` tags of LaTeX.
- **`kwargs`** – All the rest of the keyword arguments are forwarded to the `Drudge.format_latex()` method.

> **Note: For long equations in LaTeX environments,** normally `env='align'` and `sep_lines=True` can be set to allow each term to occupy separate lines, automatic page break will be inserted, or `env='dmath'` and `sep_lines=False` can be used to use `breqn` package to automatically flow the terms.

write()
Write the report.

Note that this method also closes the output file.

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

class drudge.**ScalarLatexPrinter**(*settings=None*)
Specialized LaTeX printers for usage in drudge.

Basically this class tries to fix some problems with using the original LaTeX printer from SymPy in common drudge tasks.

Specifically, for indexed objects, if the base already contains a subscript, it will be raised into a superscript wrapped inside a pair of parenthesis.

4.2 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.

4.2.1 Abstract Wick alegbra

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.

4.2.2 Concrete Wick algebras

4.2.2.1 Fermion-boson CCR/CAR algebra

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.

normal_order(*terms: pyspark.rdd.RDD*, ***kwargs*)
Normal order the field operators.

Here the normal-ordering operation of general Wick drudge will be invoked twice to ensure full simplification.

static dagger(*tensor: drudge.drudge.Tensor*, *real=False*)
Get the Hermitian adjoint of the given operator.

This method is also set to be a tensor method with the same name.

Parameters

- **tensor** – The operator to take the Hermitian adjoint for.
- **real** – If the amplitude is assumed to be real. Note that this need not be set if the amplitude is concrete real numbers.

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.

4.2.2.2 Clifford algebra

`class` `drudge.`**`CliffordDrudge`**(*ctx, inner: typing.Callable[[drudge.term.Vec, drudge.term.Vec], sympy.core.expr.Expr] = <function inner_by_delta>, **kwargs*)

Drudge for Clifford algebras.

A Clifford algebra over a inner product space V is an algebraic system with

$$uv + vu = 2\langle u, v\rangle$$

for all $u, v \in V$.

This drudge should work for any Clifford algebra with given inner product function.

`Inner`

alias of `Callable`

`__init__`(*ctx, inner: typing.Callable[[drudge.term.Vec, drudge.term.Vec], sympy.core.expr.Expr] = <function inner_by_delta>, **kwargs*)

Initialize the drudge.

Parameters

- **`ctx`** – The context for Spark.
- **`inner`** – The callable to compute the inner product of two vectors. By default, the inner product of vectors of the same base and the same number of indices will be computed to be the delta, or `ValueError` will be raised.
- **`kwargs`** – All other keyword arguments will be forwarded to the base class *`WickDrudge`*.

`phase`

The phase for Clifford algebra, negative unity.

`comparator`

Comparator for Clifford algebra.

Here we just compare vectors by the default sort key for vectors.

`contractor`

Contractor for Clifford algebra.

The inner product function will be invoked.

`normal_order`(*terms: pyspark.rdd.RDD, **kwargs*)

Put vectors in Clifford algebra in normal-order.

After the normal-ordering by Wick expansion, adjacent equal vectors will be collapsed by rules of Clifford algebra.

4.2.3 Abstract quadratic algebra

`class` `drudge.`**`GenQuadDrudge`**(*ctx, full_balance=False, **kwargs*)

Drudge for general quadratic algebra.

This abstract base class encompasses a wide range of algebraic systems. By a quadratic algebra, we mean any algebraic system with commutation rules

$$ab = \phi ba + \kappa$$

for any two elements a and b in the algebra with ϕ a scalar and κ any element in the algebra. This includes all Lie algebra systems by fixing ϕ to plus unity. Other algebra systems with other ϕ can also be treated as long as it is a scalar.

For the special case of $\phi = \pm 1$ and κ being a scalar, `WickDrudge` should be used, which utilizes the special structure and has much better performance.

`__init__`(*ctx*, *full_balance=False*, ***kwargs*)
Initialize the drudge.

`full_balance`
If full load-balancing is to be performed during normal-ordering.

`Swapper`
alias of `Callable`

`swapper`
The function to be called with two operators to commute.

It is going to be called with two vectors. When they are already in desired order, a `None` should be returned. Or the phase of the commutation should be returned as a SymPy expression, along with the commutator, which can be anything that can be interpreted as terms.

It is named as `swapper` is avoid any confusion about the established meaning of the word `commutator` in mathematics.

`normal_order`(*terms: pyspark.rdd.RDD*, ***kwargs*)
Normal order the terms in the RDD.

4.2.4 Concrete quadratic algebras

class drudge.**SU2LatticeDrudge**(*ctx*, *cartan=Vec('J^z', ())*, *raise_=Vec('J^+', ())*, *lower=Vec('J^-', ())*, *root=1*, *norm=2*, ***kwargs*)
Drudge for a lattice of SU(2) algebras.

This drudge has the commutation rules for SU(2) algebras in Cartan-Weyl form (Ladder operators). Here both the shift and Cartan operators can have additional *lattice indices*. Operators on different lattice sites always commute.

The the normal-ordering operation would try to put raising operators before the Cartan operators, which come before the lowering operators.

`__init__`(*ctx*, *cartan=Vec('J^z', ())*, *raise_=Vec('J^+', ())*, *lower=Vec('J^-', ())*, *root=1*, *norm=2*, ***kwargs*)
Initialize the drudge.

Parameters

- **`ctx`** – The Spark context for the drudge.
- **`cartan`** – The basis operator for the Cartan subalgebra (J^z operator for spin problem). It is registered in the name archive by the first letter in its label followed by an underscore.
- **`raise`** – The raising operator. It is also also registered in the name archive by the first letter in its label followed by `_p`.
- **`lower`** – The lowering operator, registered by the first letter followed by `_m`.

- **root** – The coefficient for the commutator between the Cartan and shift operators.
- **norm** – The coefficient for the commutator between the raising and lowering operators.
- **kwargs** – All other keyword arguments are given to the base class `GenQuadDrudge`.

swapper
The swapper for the spin algebra.

4.3 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.

Also Einstein summation convention is assumed for this drudge in drudge scripts.

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. None if spin values are not given.

spin_range
The symbolic range for spin values. None if it is not given.

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 explicit spin quantum number. It can be an empty sequence to disable explicit spin. Or it can be a sequence of SymPy expressions to give explicit spin values, or a range and dummies pair for symbolic spin.
- **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, 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, 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, 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, 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', 0, nv), (a, b, c, d, 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', 0, no), (i, j, k, l, 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)), spin=(SpinOneHalf.UP, SpinOneHalf.DOWN), **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, which are set to *UP* and *DOWN* by default. 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', 0, nv), (a, b, c, d, 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', 0, no), (i, j, k, l, 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)), spin=(SpinOneHalf.UP, SpinOneHalf.DOWN), **kwargs*)
Initialize the particle-hole drudge.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX