

# Tensor Contraction Engine Manual



written by Alexander A. Auer

## Tensor Contraction Engine v.1.0

Author : So Hirata

©All rights reserved by Battelle & Pacific Northwest Nat'l Lab  
(2002)

*Sanibel Meeting beta version, March 2004*

# Contents

<b>1</b>	<b>Authors and Collaborators</b>	<b>2</b>
<b>2</b>	<b>Introduction</b>	<b>3</b>
2.1	Basic usage of python interface . . . . .	5
2.2	Usage of the Graphical User Interface (GUI) for the TCE . . .	5
2.3	Usage of the Algebraic Expression Engine AEE . . . . .	8
2.4	Usage of the Operator Contraction Engine . . . . .	13
2.5	Usage of the Tensor Contraction Engine . . . . .	15
<b>3</b>	<b>Structure of the generated code</b>	<b>18</b>
<b>4</b>	<b>CCSD example</b>	<b>23</b>
4.1	OCE input syntax . . . . .	24
4.2	CCSD python script . . . . .	25
4.3	OCE procedures . . . . .	26
4.4	TCE procedures . . . . .	27
<b>5</b>	<b>TCE related www links</b>	<b>28</b>
<b>6</b>	<b>Python literature</b>	<b>29</b>

# 1 Authors and Collaborators

So Hirata

William R Wiley Environmental Molecular Sciences Laboratory, Battelle,  
Pacific Northwest National Laboratory, So.Hirata@pnl.gov

Marcel Nooijen, Alexander Auer

Department of Chemistry, University of Waterloo, nooijen@uwaterloo.ca,  
auer@uwaterloo.ca

P. Sadayappan, Gerald Baumgartner, Alina Bibireata, Daniel Cociorva,  
Xiaoyang Gao, Sriram Krishnamoorthy, Sandhya Krishnan,  
Chi-Chung Lam, Qingda Lu, Alexander Sibiryakov

Computer Science, Ohio State University, saday@cis.ohio-state.edu

David E. Bernholdt, Robert J. Harrison, Venkatesh Choppella

Oak Ridge National Laboratory, P O Box 2008, Bldg 6012, MS 6367, Oak  
Ridge, TN 37831-6367, Fax: 865-574-0680, bernholdtde@ornl.gov

Russell M. Pitzer

Department of Chemistry, Ohio State University

J. Ramanujam

Computer Science, Louisiana State University

## 2 Introduction

The time determining step in the development of electronic structure methods from the idea for a new algorithm (like a parallel implementation), a new approximation (like local methods) or a new theory (like multi-reference Coupled Cluster) to the actual application of the latter is often the computational implementation of the algebraic expressions.

Furthermore, while various ideas are implemented in different program packages the attempt to combine methods often requires large merging or recoding efforts.

As most of the common correlated methods like Many Body Perturbation Theory or Coupled Cluster Theory have a complicated but systematic structure an automated scheme for generating an efficient implementation from the algebraic expressions presents an alternative to the time consuming and error prone implementation by hand.

The basic idea of the TCE is, that any method which can be formulated as a series of tensor contractions can be implemented automatically starting from the algebraic expressions, converting them to working equations which are then factorized and translated to Fortran code.

This generated code incorporates :

- use of permutational, spatial and spin symmetry
- architecture adjusted use of RAM
- parallel evaluation of tensor contractions

This guide is supposed to give a basic outline of the following points :

- How to use the code generator ?
  - tools in the toolbox
  - input and output files
- What does it do ?
  - capabilities of the generator
  - procedures of the modules
- How does the synthetic code look like ?
  - tensor contractions
  - use of symmetry
  - interface issues

These are the basic modules included in the TCE package :

- Algebraic Expression Engine (AEE)
  - generate algebraic expressions as input for the OCE
- Graphical User Interface (GUI) to the TCE
  - graphical front end to OCE and TCE modules
- Operator Contraction Engine (OCE)
  - evaluate Wick's theorem
  - delete disconnected terms
- Tensor Contraction Engine (TCE)
  - strength reduction of equations
  - factorization of term
  - generation of computer code

## 2.1 Basic usage of python interface

When modules are loaded and used in the interactive mode, python has to be started first. In the terminal, type :

```
user@computer> python
```

to start the interactive mode of python :

```
Python 2.1.1 (#1, Oct 25 2001, 09:53:13)
[GCC 2.95.3 20010315 (SuSE)] on linux2
Type "copyright", "credits" or "license" for more information.
>>>
```

the interactive mode can be quitted by pressing Ctrl-D . All modules can be loaded by using the import command, for example

```
>>> import tce
```

will load the module tce from the corresponding tce.py file and all functionality inside tce will be available from thereon. Another useful feature is the "show" function that can be used on an object

```
>>> object.show()
```

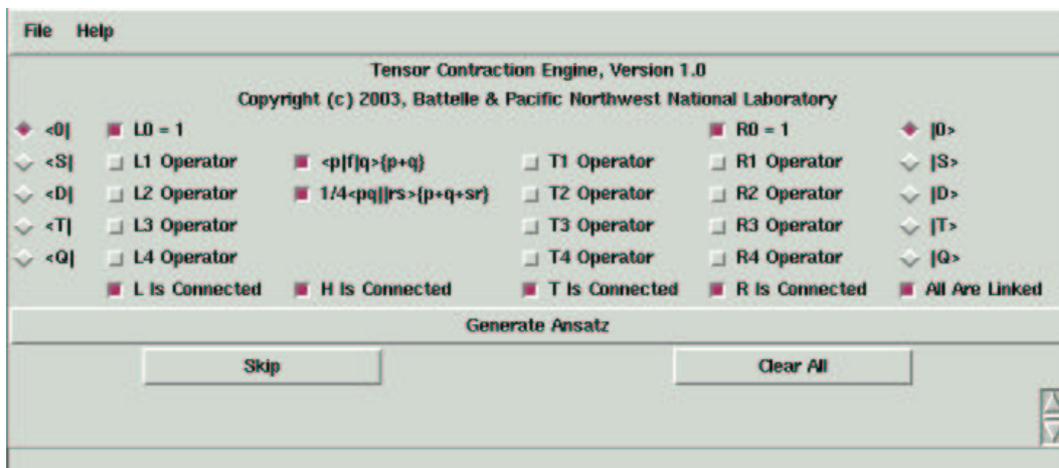
and will pretty-print its content.

## 2.2 Usage of the Graphical User Interface (GUI) for the TCE

The GUI provides an easy and intuitive way of using the existing functionality of the TCE without needing to know the commands and procedures involved. After starting the GUI most of it is self explaining :

```
user@computer> python ccc.py
```

Everything the GUI does can be monitored in the window at the bottom of the GUI. You will find a screenshot on the next page -



One can immediately generate the CCSD energy equations by executing the GUI after checking the box for the T1 and T2 operator. All that needs to be done is clicking the "Generate Ansatz" button through the different steps until the Fortran code is generated.

The basic expression for an Ansatz defined in the GUI is :

$$\langle bra | \hat{L} \hat{H} e^{\hat{T}} \hat{R} | ket \rangle$$

$\hat{R}, \hat{T}$  are excitation operators,  $\hat{R}$  is a de-excitation operator,  $\hat{H}$  is the Hamiltonian. Here are two examples of how different methods could be defined using the GUI :

$$\langle D | L0 [ H e^{T2} ]_{connected} R0 | 0 \rangle \quad \text{CCD doubles equations}$$

$$\langle 0 | (L1 + L2) [ H e^{(T1+T2)} ]_{connected} R0 | S \rangle \quad \begin{array}{l} \text{singles component} \\ \text{of left hand EOM - CCSD} \\ \text{/ CCSD gradient } \Lambda \text{ equation} \end{array}$$

The steps the GUI will execute are precisely those outlined in the OCE and TCE sections to follow. Additionally, the GUI generates the initial algebraic expressions for a given type of Ansatz.

Below is a list of the different options in the GUI :

- **File / Help**  
exits the GUI and provides author information

- **Bra  $\langle 0|$ ,  $\langle 1|$  ....**  
check the buttons at the very left and right to chose the bra and ket state (0 = ground state, S= singly excited state etc.)
- **Left and right hand operators L1, L2 ... and R1, R2 ...**  
EOM right and left hand eigenvectors (L0=1, R0=1 in standard CC theory)
- **Hamiltonian**  
One and two electron part of the Hamilton operator can be checked here to compose the Hamiltonian.
- **T-Operator T1, T2 ...**  
Coupled Cluster excitation operators in the exponential expansion, just as for bra, ket, R, L, T and the Hamiltonian the last box at the bottom can be checked to define the operators as connected.
- **Generate Ansatz button**  
This is the main button that executes the next step to be carried out by the code generator, the label on the button indicates the next step the generator will carry out.
- **Skip button**  
Skip the next step of the generator, useful for skipping the deletion of disconnected terms in some theories.
- **Clear All button**  
Restart function.
- **Status window**  
In the window at the bottom of the GUI the output of the running generator and intermediate Results will be displayed.

As the GUI executes the TCE modules the user will be prompted to save intermediate results and what filename the final Fortran routines should have (the extension .F will be added by the GUI).

## 2.3 Usage of the Algebraic Expression Engine AEE

The AEE is intended to generate the algebraic expression from Coupled Cluster or Many Body Perturbation Theory using only operators and their expansions as input. It is meant to be an open ended, simple toolbox that is easy to modify, extend and customize and can be used as an alternative to running the GUI or writing the OCE input by hand.

The AEE will generate an input file for the OCE module. A typical line of OCE input is a sum of operators in second quantization with a prefactor looking like this :

```
1.0/1.0 Sum( g5 g6 ) f( g5 g6 ) { h1+ h2+ p4 p3 } { g5+ g6 }
```

All lines compose an operator sequence  $\hat{O}$  and will be processed by the OCE as  $\langle 0 | \hat{O} | 0 \rangle$ , so the bra and ket have to be defined through their creation and annihilation operators and included in  $\hat{O}$ .

In order to generate this first level of working equations one needs to go through three steps

- define operators using the intrinsic AEE types
- carry out summations or expansions using the AEE functions
- store the intermediate objects in order to pass them to the OCE

Operators, their sums or commutators must be defined as "Terms", which is an intrinsic object of the AEE. A "Term" is defined as :

```
Term(numerator,denominator,"name of operator")
```

where the first two entries are the prefactor and the last entry is the name of the operator. Defining the object t2amp as the CC T2 amplitudes (1/4 T2) would look like this

```
>>> t2amp=aee.Term(1.0,4.0,["T2"])
>>> t3amp.show()
" +1.0/4.0 ['T2'] "
```

note that "aee." has to be included to indicate that "Term" is an object defined inside the AEE module, the ".show()" sends the object to the function show() that will print its content.

Let's define all operators needed for the CCSD equations

```

>>> amp1=aee.Term(1.0,["T1"])
>>> amp2=aee.Term(1.0,4.0,["T2"])
>>> one=aee.Term(1.0,1.0,["F"])
>>> two=aee.Term(1.0,4.0,["V"])

```

How does the AEE know what a T2 amplitude is or what F or V are ? There is a database of operators inside the AEE which names the most important operators with is creation and annihilation operators. These are the supported operators :

```

"V"    : 2e- part of H
"F"    : fock matrix element
"T"    : cc-amplitude of given order (T1, T2, T3, etc)
"bra"  : bra state represented by operators acting on <0|
        of given excitation rank (bra1, bra2, bra3, ...)

```

In order to construct the Hamilton operator as F+V and the CCSD amplitudes as T1 + T2 the "add" function has to be used :

```

>>> h=aee.Term.add(one,two)
>>> h.show()
+1.0/['F'] []
+1.0/['V'] []
>>> ccscdamplitudes=aee.Term.add(amp1,amp2)
>>> ccscdamplitudes.show()
+1.0/['T1'] []
+1.0/['T2'] []

```

this has been done by calling the "add" function of the "Term" object from the AEE module, aee.Term.add(" ", " "). All Objects now become "Sums" which are then processed.

*(The objects in the AEE are usually defined as "Terms" and processed as "Sums", note that this is rather a data structure than algebraic terminology)*

In order to use these operators to generate the input for the OCE a small set of functions can be used. The functions for the object "Sum" are :

### **multiply**

multiplies two Sum objects :

```

>>> test=aee.Sum.multiply(h,h)
>>> test.show()
+1.0/1.0 ['F', 'F']
+1.0/4.0 ['F', 'V']

```

```
+1.0/4.0 ['V', 'F']
+1.0/16.0 ['V', 'V']
```

### **add**

adds two Sum objects

```
>>> test=aee.Sum.add(h,h)
>>> test.show()
+1.0/1.0 ['F']
+1.0/4.0 ['V']
+1.0/1.0 ['F']
+1.0/4.0 ['V']
```

### **commutator**

form commutator of two Sum objects

```
>>> amp1=aee.Term(1.0,1.0,["T1"])
>>> amp2=aee.Term(1.0,4.0,["T2"])
>>> t1=aee.Sum([amp1])
>>> t2=aee.Sum([amp2])
>>> test=aee.Sum.commutator(t1,t2)
>>> test.show()
+1.0/4.0 ['T1', 'T2']
-1.0/4.0 ['T2', 'T1']
```

*(Note that the Terms amp1 and amp2 have to be converted to "Sum" objects before the operation Sum.commutator() can be carried out !)*

### **exponential(object,order)**

evaluates exponential expansion of operator to a given order

```
>>> x=aee.Sum.exponential(t1,1)
>>> x.show()
+1.0/1.0 ['T1']
>>> x=aee.Sum.exponential(t1,2)
>>> x.show()
+1.0/1.0 ['T1']
+1.0/2.0 ['T1', 'T1']
```

### **bch**

evaluates the Baker Campbell Hausdorff expansion to a given order

```
>>> ccd=aee.Sum.bch(h,t2,2)
>>> ccd.show()
```

```

+1.0/1.0 ['F']
+1.0/4.0 ['V']
+1.0/4.0 ['F', 'T2']
+1.0/16.0 ['V', 'T2']
-1.0/4.0 ['T2', 'F']
-1.0/16.0 ['T2', 'V']

```

In order to generate the CCSD T2 equations  $\langle_{ij}^{ab} | e^{-T} \hat{H} e^T | 0 \rangle = 0$  the following steps have to be carried out :

- make  $h=f+v$
- make  $t=t1+t2$ ,
- set up bch expansion to fourth order (terminates after fourth)
- multiply with doubly excited bra state

this looks like the following :

```

>>> import aee
>>> amp1=aee.Term(1.0,1.0,["T1"])
>>> amp2=aee.Term(1.0,4.0,["T2"])
>>> tccsd=aee.Term.add(amp1,amp2)
>>> one=aee.Term(1.0,1.0,["F"])
>>> two=aee.Term(1.0,4.0,["V"])
>>> h=aee.Term.add(one,two)
>>> doub=aee.Term(1.0,1.0,["bra2"])
>>> doubles=aee.Sum([doub])
>>> ccsd=aee.Sum.bch(h,tccsd,4)
>>> ccsdd=aee.Sum.multiply(doubles,ccsd)

```

note that the object "doub" has to be converted to a "Sum" again and that the Sum.multiply is used to multiply with the bra state.

In order to give the generated expressions "the final touch", that is to complete the objects by adding their creation and annihilation operators, the function aee.Sum.addindices() has to be called

```

>>> h.show()
+1.0/1.0 ['F']
+1.0/4.0 ['V']
>>> out=aee.Sum.addindices(h)
>>> out.show()

```

```
+1.0/1.0 [[' Sum', 'g1', 'g2'], [['f', ['g1', 'g2']], [['g1+', 'g2']]]
+1.0/4.0 [[' Sum', 'g1', 'g2', 'g3', 'g4'], [['v', ['g1', 'g2', 'g3', 'g4']], [['g1+', 'g2+', 'g4', 'g3']]]
```

finally, the "Sum" object will be written to a file using the function :

```
aee.Sum.writefullformula(object,"filename")
```

The output file will be read by the OCE, which evaluates Wick's theorem etc. In the CCSD example the previous two steps would read like :

```
>>> out=aee.Sum.addindices(ccsdd)
>>> aee.Sum.writefullformula(out,"test")
```

the final example illustrates the generation of the CCD T2 equations

```
>>> import aee
>>> amp2=aee.Term(1.0,4.0,["T2"])
>>> tccd=aee.Sum([amp2])
>>> one=aee.Term(1.0,1.0,["F"])
>>> two=aee.Term(1.0,4.0,["V"])
>>> h=aee.Term.add(one,two)
>>> doub=aee.Term(1.0,1.0,["bra2"])
>>> doubles=aee.Sum([doub])
>>> ccd=aee.Sum.bch(h,tccd,2)
>>> ccdd=aee.Sum.multiply(doubles,ccd)
>>> ccdd.show()
+1.0/1.0 ['bra2', 'F']
+1.0/4.0 ['bra2', 'V']
+1.0/4.0 ['bra2', 'F', 'T2']
+1.0/16.0 ['bra2', 'V', 'T2']
-1.0/4.0 ['bra2', 'T2', 'F']
-1.0/16.0 ['bra2', 'T2', 'V']
>>> out=aee.Sum.addindices(ccdd)
>>> aee.Sum.writefullformula(out,"test")
```

note again that some "Term" objects have to be converted to "Sum" objects and that in this case the bch expansion only has to be carried out to second order. The output file will look like this :

```
1.0/1.0 Sum( g5 g6 ) f( g5 g6 ) { h1+ h2+ p4 p3 } { g5+ g6 }
1.0/4.0 Sum( g5 g6 g7 g8 ) v( g5 g6 g7 g8 ) { h1+ h2+ p4 p3 } { g5+ g6+ g8 g7 }
1.0/4.0 Sum( g5 g6 p7 p8 h9 h10 ) f( g5 g6 ) t( p7 p8 h9 h10 ) { h1+ h2+ p4 p3 } { g5+ g6 } { p7+ p8+ h10 h9 }
1.0/16.0 Sum( g5 g6 g7 g8 p9 p10 h11 h12 ) v( g5 g6 g7 g8 ) t( p9 p10 h11 h12 ) { h1+ h2+ p4 p3 } { g5+ g6+ g8 g7 } { p9+ p10+ h12 h11 }
-1.0/4.0 Sum( p5 p6 h7 h8 g9 g10 ) t( p5 p6 h7 h8 ) f( g9 g10 ) { h1+ h2+ p4 p3 } { p5+ p6+ h8 h7 } { g9+ g10 }
-1.0/16.0 Sum( p5 p6 h7 h8 g9 g10 g11 g12 ) t( p5 p6 h7 h8 ) v( g9 g10 g11 g12 ) { h1+ h2+ p4 p3 } { p5+ p6+ h8 h7 } { g9+ g10+ g12 g11 }
```

*(note that the OCE is a little picky as far as the input format is concerned, the OCE input file may not contain a last blank line and each line must end with a single blank space)*

## 2.4 Usage of the Operator Contraction Engine

The OCE is intended to evaluate an expression in second quantization to arrive at algebraic expressions which only contain tensor contractions. These are the successive steps carried out by the OCE :

- read an input file in special format  
(as generated by the AEE or GUI or written by hand)
- perform all contractions of the operators in second quantization  
(evaluate Wick's Theorem for an expression  $\langle 0| \textit{operator sequence} |0\rangle$  recursively)
- delete disconnected terms if needed (CC or MBPT)
- write final tensor contractions into output file to be read by TCE

Note that the general structure of any tensor contraction looks like this :

$$U_{x_1ext, x_2ext, \dots}^{y_1ext, y_2ext, \dots} = \sum_{x_{int}, y_{int}} V_{x_{int}, x_1ext}^{y_{int}, y_1ext} * W_{y_{int}, x_2ext}^{x_{int}, y_2ext} * \dots$$

The basic principle is that any method which can be formulated as a series of tensor contractions can be implemented automatically starting from the algebraic expressions. The OCE is the module that will generate the actual working equations. These are then factorized and translated to Fortran code by the next module, the TCE (see below).

To continue with the example above, we load the OCE into python and read the input file given above

```
>>> import oce
>>> ccdd=oce.readfromfile("test")
```

The objects and classes that the OCE uses are the basic quantities that are needed to describe tensor contractions :

- Operator (second quantized operator with type and index),
- Summation (summation indices),
- Amplitude (amplitude type and indices),
- Factor (coefficient and (!!)) permutations)

but also two joined classes that describe the contractions :

- OperatorSequence (sequence of normal ordered second quantized operators with factor and summation)
- ListOperatorSequences (a list of a number of operator sequences)

in order to proceed we declare a list of operator sequences

```
>>> ccd_t2=oce.ListOperatorSequences()
>>> ccd_t2.join(ccdd.performfullcontraction().deletedisconnected().simplify())
```

and pass the input through the procedures that evaluate the contractions recursively and delete the disconnected terms. The simplify procedure will identify and consolidate identical contractions. The output is dumped to a file using a similar command as in the AEE :

```
>>> ccd_t2.writetofile("ccd_t2.out")
```

The output generated in this way looks like this :

```
[ - 1.0 ] * v ( p4 p3 h1 h2 )
[ - 1.0 + 1.0 * P( p3 p4 h1 h2 => p3 p4 h2 h1 ) ] * Sum ( h5 ) * f ( h5 h1 ) * t ( p3 p4 h5 h2 )
[ + 1.0 - 1.0 * P( p4 p3 h2 h1 => p3 p4 h2 h1 ) ] * Sum ( p5 ) * f ( p4 p5 ) * t ( p5 p3 h2 h1 )
[ + 0.5 ] * Sum ( h5 h6 ) * t ( p3 p4 h5 h6 ) * v ( h5 h6 h1 h2 )
[ + 1.0 - 1.0 * P( p3 p4 h2 h1 => p4 p3 h2 h1 ) - 1.0 * P( p3 p4 h2 h1 => p3 p4 h1 h2 ) + 1.0 * P( p3 p4 h2 h1 => p4 p3 h1 h2 ) ]
* Sum ( h6 p5 ) * t ( p5 p3 h6 h2 ) * v ( h6 p4 h1 p5 )
[ + 0.5 ] * Sum ( p5 p6 ) * t ( p5 p6 h2 h1 ) * v ( p4 p3 p5 p6 )
[ - 0.5 + 0.5 * P( p4 p3 h2 h1 => p3 p4 h2 h1 ) ] * Sum ( h7 h8 p5 p6 ) * t ( p5 p4 h2 h1 ) * t ( p6 p3 h7 h8 ) * v ( h7 h8 p5 p6 )
[ - 0.25 ] * Sum ( h7 h8 p5 p6 ) * t ( p5 p6 h2 h1 ) * t ( p3 p4 h7 h8 ) * v ( h7 h8 p5 p6 )
[ - 0.5 + 0.5 * P( p3 p4 h1 h2 => p3 p4 h2 h1 ) ] * Sum ( h5 h8 p6 p7 ) * t ( p3 p4 h5 h1 ) * t ( p6 p7 h8 h2 ) * v ( h5 h8 p6 p7 )
[ - 1.0 + 1.0 * P( p4 p3 h1 h2 => p3 p4 h1 h2 ) ] * Sum ( h6 h8 p5 p7 ) * t ( p5 p4 h6 h1 ) * t ( p7 p3 h8 h2 ) * v ( h6 h8 p5 p7 )
```

Note that all second quantized operators are resolved and the permutational symmetry for the contractions has been introduced. The factors are expressed as :

```
[factor for given contraction +
 factors for permuted contraction
 * P( indices of given contraction => indices of permuted contraction ) ]
```

for example :

```
[ - 0.5 + 0.5 * P( p4 p3 h2 h1 => p3 p4 h2 h1 ) ]
```

This output can directly be used as input for the TCE

## 2.5 Usage of the Tensor Contraction Engine

The TCE module incorporates a lot of functionality, everything that is needed to turn the raw set of tensor contractions into computer code :

- read the OCE output
- factorize (breakdown) the separate tensor contractions creating intermediates
- factorize the complete list of tensor contractions creating an operation tree
- conversion of the operation tree to Fortran

In the first step the TCE module is loaded into python and the output from the OCE is read

```
>>> import tce
>>> ccddt=tce.readfromfile("ccd_t2.out")
```

There is one thing to note when discussing the factorization of the tensor contractions. A large number of tensor contractions leads to an exponential growth in the number of possible factorizations. This inhibits doing an exhaustive search (trying all possibilities, that is). The solution used here is to generate a non optimal factorization with minimal scaling of overall computational effort by factorizing every contraction by itself. More optimal schemes of finding efficient factorizations would use Simulated Annealing or Genetic Algorithms and are currently under development.

Let's continue with our example :

Next, the different contractions are processed, and the best breakdown of the for each one is found separately :

```
>>> ccddt=ccddt.breakdown()
```

The output of this procedure might look like this :

```
... canonicalizing permutation operator expressions
[ - 1.0 ] * v ( p4 p3 h1 h2 )
... there are 1 ways of breaking down the sequence into elementary tensor contractions
... the best breakdown is [1] with operation cost=N0 00 V0, memory cost=N0 00 V0
... canonicalizing permutation operator expressions
[ - 1.0 + 1.0 * P( h1 h2 p3 p4 => h2 h1 p3 p4 ) ] * Sum ( h5 ) * f ( h5 h1 ) * t ( p3 p4 h5 h2 )
... there are 2 ways of breaking down the sequence into elementary tensor contractions
... the best breakdown is [1, 2] with operation cost=N0 03 V2, memory cost=N0 02 V2
... the suggested decomposition of the permutation operator is valid
... canonicalizing permutation operator expressions
[ + 1.0 - 1.0 * P( h1 h2 p3 p4 => h1 h2 p4 p3 ) ] * Sum ( p5 ) * f ( p4 p5 ) * t ( p5 p3 h2 h1 )
... there are 2 ways of breaking down the sequence into elementary tensor contractions
... the best breakdown is [1, 2] with operation cost=N0 02 V3, memory cost=N0 02
... the suggested decomposition of the permutation operator is valid
```

which nicely shows that the terms are factorized separately following the scheme :

- canonicalize the indices according to internal standard
- find the factorization with the lowest scaling in operation cost by exhaustive search for that specific term
- create and check the permutational symmetry of the resulting contraction

Each contraction has been broken down like this :

```
i0 ( p3 p4 h1 h2 ) + = [ - 0.5 + 0.5 * P( h1 h2 p3 p4 => h1 h2 p4 p3 ) ]
                        * Sum ( p5 ) * i1 ( p3 p5 ) * t ( p4 p5 h1 h2 )
i1 ( p3 p5 ) + = [ - 1.0 ] * Sum ( p6 h7 h8 ) * t ( p3 p6 h7 h8 ) * v ( h7 h8 p5 p6 )
...
...
```

Next, all contractions are factorized together by identifying common intermediates in the procedure :

```
>>> ccd=ccdt.fullyfactorize()
```

Now the operation tree for the CCD- $T_2$  looks like this :

```
i0 ( p3 p4 h1 h2 ) + = [ + 1.0 ] * v ( p3 p4 h1 h2 )
i0 ( p3 p4 h1 h2 ) + = [ - 1.0 + 1.0 * P( h1 h2 p3 p4 => h2 h1 p3 p4 ) ] * Sum ( h5 ) * t ( p3 p4 h1 h5 ) * i1 ( h5 h2 )
i1 ( h5 h1 ) + = [ + 1.0 ] * f ( h5 h1 )
i1 ( h5 h1 ) + = [ + 0.5 ] * Sum ( p6 p7 h8 ) * t ( p6 p7 h1 h8 ) * v ( h5 h8 p6 p7 )
i0 ( p3 p4 h1 h2 ) + = [ + 1.0 - 1.0 * P( h1 h2 p3 p4 => h1 h2 p4 p3 ) ] * Sum ( p5 ) * t ( p3 p5 h1 h2 ) * i1 ( p4 p5 )
i1 ( p3 p5 ) + = [ + 1.0 ] * f ( p3 p5 )
i1 ( p3 p5 ) + = [ - 0.5 ] * Sum ( p6 h7 h8 ) * t ( p3 p6 h7 h8 ) * v ( h7 h8 p5 p6 )
i0 ( p3 p4 h1 h2 ) + = [ - 0.5 ] * Sum ( h9 h7 ) * t ( p3 p4 h7 h9 ) * i1 ( h7 h9 h1 h2 )
i1 ( h7 h9 h1 h2 ) + = [ - 1.0 ] * v ( h7 h9 h1 h2 )
i1 ( h7 h9 h1 h2 ) + = [ - 0.5 ] * Sum ( p5 p6 ) * t ( p5 p6 h1 h2 ) * v ( h7 h9 p5 p6 )
i0 ( p3 p4 h1 h2 ) + = [ - 1.0 + 1.0 * P( h1 h2 p3 p4 => h2 h1 p3 p4 ) + 1.0 * P( h1 h2 p3 p4 => h1 h2 p4 p3 )
                    - 1.0 * P( h1 h2 p3 p4 => h2 h1 p4 p3 ) ] * Sum ( p5 h6 ) * t ( p3 p5 h1 h6 ) * i1 ( h6 p4 h2 p5 )
i1 ( h6 p3 h1 p5 ) + = [ + 1.0 ] * v ( h6 p3 h1 p5 )
i1 ( h6 p3 h1 p5 ) + = [ - 0.5 ] * Sum ( p7 h8 ) * t ( p3 p7 h1 h8 ) * v ( h6 h8 p5 p7 )
i0 ( p3 p4 h1 h2 ) + = [ + 0.5 ] * Sum ( p5 p6 ) * t ( p5 p6 h1 h2 ) * v ( p3 p4 p5 p6 )
```

This operation tree is the outline of the final algorithm. All that remains to be done is to turn this outline into Fortran code. For this purpose, different procedures are available :

- fortran90
- fortran77
- tex

While the fortran77 and fortran90 functions will generate Fortran90 or Fortran77 code, the tex feature shows how convenient the usage of the generator can be, even parts of the documentation can be generated :

```
>>> tce.writetofile(ccd.tex(),"test.tex")
```

will dump something like this to the file test.tex :

$$\begin{aligned}
& \chi_{h_1 h_2}^{p_3 p_4} = \chi_{h_1 h_2}^{p_3 p_4} + v_{h_1 h_2}^{p_3 p_4} \\
\chi_{h_1 h_2}^{p_3 p_4} &= \chi_{h_1 h_2}^{p_3 p_4} - \left( +1 - P_{h_2 h_1 p_3 p_4}^{h_1 h_2 p_3 p_4} \right) t_{h_1 h_5}^{p_3 p_4} \xi_{h_2}^{h_5} \\
& \xi_{h_1}^{h_5} = \xi_{h_1}^{h_5} + f_{h_1}^{h_5} \\
& \xi_{h_1}^{h_5} = \xi_{h_1}^{h_5} + \frac{1}{2} t_{h_1 h_8}^{p_6 p_7} v_{p_6 p_7}^{h_5 h_8} \\
\chi_{h_1 h_2}^{p_3 p_4} &= \chi_{h_1 h_2}^{p_3 p_4} + \left( +1 - P_{h_1 h_2 p_3 p_4}^{h_1 h_2 p_4 p_3} \right) t_{h_1 h_2}^{p_3 p_5} \xi_{p_5}^{p_4} \\
& \xi_{p_5}^{p_3} = \xi_{p_5}^{p_3} + f_{p_5}^{p_3} \\
& \xi_{p_5}^{p_3} = \xi_{p_5}^{p_3} - \frac{1}{2} t_{h_7 h_8}^{p_3 p_6} v_{p_5 p_6}^{h_7 h_8} \\
\chi_{h_1 h_2}^{p_3 p_4} &= \chi_{h_1 h_2}^{p_3 p_4} - \frac{1}{2} t_{h_7 h_9}^{p_3 p_4} \xi_{h_1 h_2}^{h_7 h_9} \\
& \xi_{h_1 h_2}^{h_7 h_9} = \xi_{h_1 h_2}^{h_7 h_9} - v_{h_1 h_2}^{h_7 h_9} \\
& \xi_{h_1 h_2}^{h_7 h_9} = \xi_{h_1 h_2}^{h_7 h_9} - \frac{1}{2} t_{h_1 h_2}^{p_5 p_6} v_{p_5 p_6}^{h_7 h_9} \\
\chi_{h_1 h_2}^{p_3 p_4} &= \chi_{h_1 h_2}^{p_3 p_4} - \left( +1 - P_{h_2 h_1 p_3 p_4}^{h_1 h_2 p_3 p_4} - P_{h_1 h_2 p_3 p_4}^{h_1 h_2 p_4 p_3} + P_{h_2 h_1 p_3 p_4}^{h_1 h_2 p_4 p_3} \right) t_{h_1 h_6}^{p_3 p_5} \xi_{h_2 p_5}^{h_6 p_4} \\
& \xi_{h_1 p_5}^{h_6 p_3} = \xi_{h_1 p_5}^{h_6 p_3} + v_{h_1 p_5}^{h_6 p_3} \\
& \xi_{h_1 p_5}^{h_6 p_3} = \xi_{h_1 p_5}^{h_6 p_3} - \frac{1}{2} t_{h_1 h_8}^{p_3 p_7} v_{p_5 p_7}^{h_6 h_8} \\
\chi_{h_1 h_2}^{p_3 p_4} &= \chi_{h_1 h_2}^{p_3 p_4} + \frac{1}{2} t_{h_1 h_2}^{p_5 p_6} v_{p_5 p_6}^{p_3 p_4}
\end{aligned}$$

To create a Fortran subroutine that can be interfaced to a program package the following command can be used :

```
>> ccd.fortran77("ccd_t2").writetofile("ccd_t2_tce")
```

which will dump the file ccd\_t2\_tce.F containing a subroutine called ccd\_t2 with several calls to contraction routines.

### 3 Structure of the generated code

The generated Fortran code is always one main subroutine, which includes all the contraction routines. It is called with the amplitudes, integrals, Fock matrix, and the residual as arguments. Every tensor is passed with file handle (or a starting address in memory) and an offset array which determines the start address of a specific tile (see below) in the file.

Every contraction is done in a separate subroutine and the files/memory and offset arrays for the intermediate quantities are created/allocated and deleted/de-allocated in the main routine.

A schematic representation of the main routine looks like this :

```

SUBROUTINE ccsd_t2(file_handle_t1,offset_array_t1 ..... )

C   header :
C   $Id: ccsd_t2.F,v 1.8 2002/12/01 21:37:29 sohirata Exp $
C   This is a Fortran77 program generated by Tensor Contraction Engine v.1.0

C comments describing all contractions in this routine
C   ....
C i0 ( p3 p4 h1 h2 ) + = -1.0 * P( 2 ) * Sum ( p5 ) * t ( p5 h1 ) * i1 ( p3 p4 h2 p5 )
C   i1 ( p3 p4 h1 p5 ) + = 1.0 * v ( p3 p4 h1 p5 )
C   i1 ( p3 p4 h1 p5 ) + = -0.5 * Sum ( p6 ) * t ( p6 h1 ) * v ( p3 p4 p5 p6 )
C   ....

C   variables used in this routine :
      IMPLICIT NONE
#include "global.fh"
#include "mafdecls.fh"
#include "util.fh"
#include "tce.fh"
      INTEGER d_i0
      INTEGER k_i0_offset
      INTEGER d_v2
      INTEGER k_v2_offset
      ....

C call to contraction routine with intermediates
C create offset array with addresses of blocks (mem alloc done inside)
      CALL OFFSET_ccsd_t2_3_1(d_i1,l_i1_offset,k_i1_offset,size_i1)
C create the intermediate file
      CALL TCE_FILENAME('ccsd_t2_3_1_i2',filename)
```

```

        CALL CREATEFILE(filename,d_i1,size_i1)
C do the contraction v -> i1
        CALL ccsd_t2_3_1(d_v2,k_v2_offset,d_i1,k_i1_offset)
C second contraction v2 * t1 -> i0
        CALL ccsd_t2_3_2(d_t1,k_t1_offset,d_v2,k_v2_offset,d_i1,k_i1_offse
        &t)
C communicate result
        CALL RECONCILEFILE(d_i1,size_i1)
C consume i1 * t1 -> i0
        CALL ccsd_t2_3(d_t1,k_t1_offset,d_i1,k_i1_offset,d_i0,k_i0_offset)
C delete i1 file and de-allocate offset array
        CALL DELETEFILE(d_i1)
        IF (.not.MA_POP_STACK(l_i1_offset)) CALL ERRQUIT('ccsd_t2',-1)

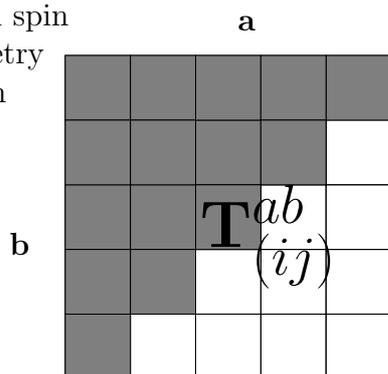
        ....

```

(note that in contrast to the example above the generator does not comment the code except for the basic description of the contractions, and there are no blank lines included)

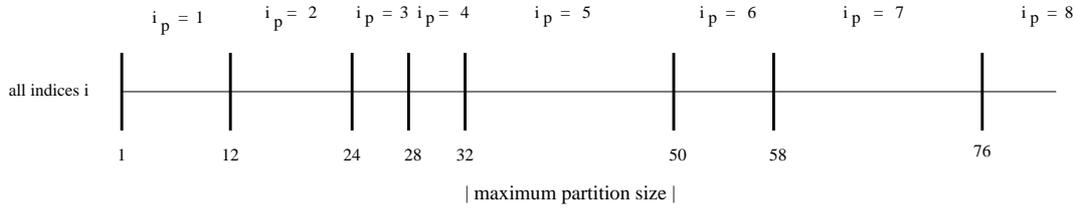
The actual work is done in the contraction routines shown above. The features of these include various ideas to overcome common bottlenecks of electronic structure theory codes. The basic principle in all contractions is the tiling of all tensors that are processed in the TCE generated code :

stored with spin  
and symmetry  
information



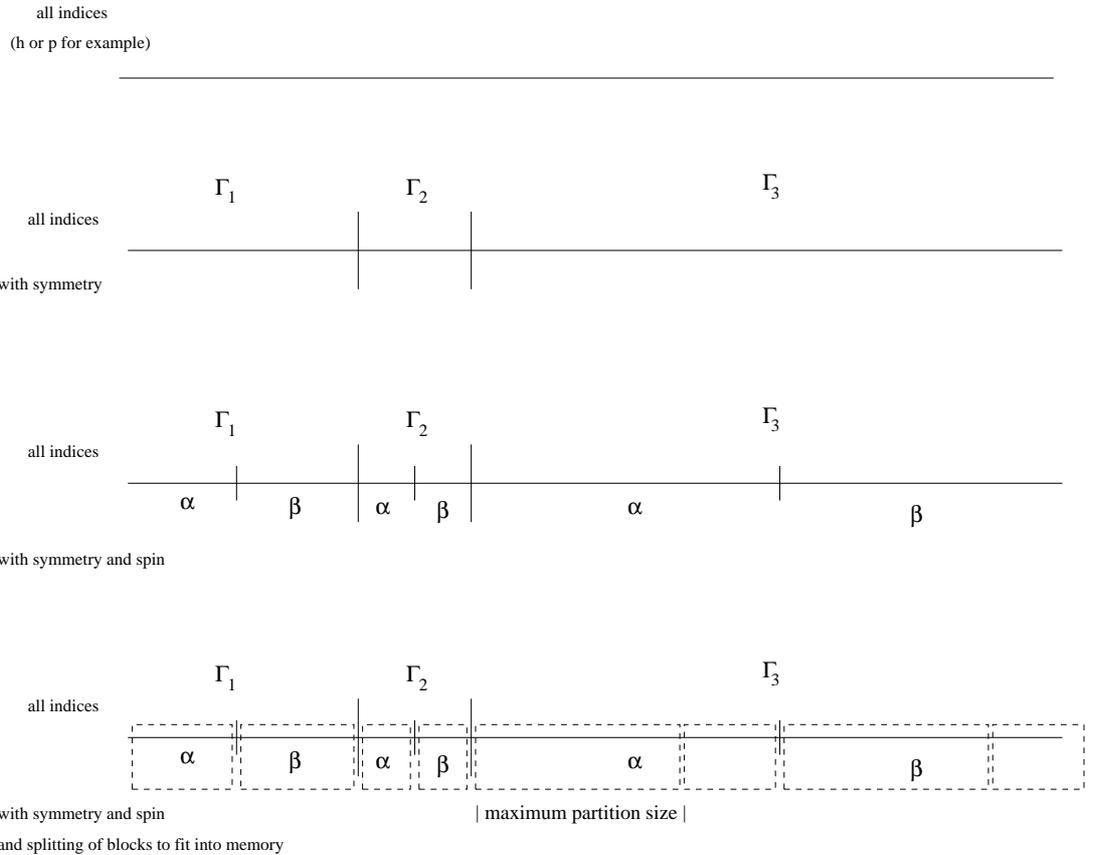
not stored

The tiling of a tensor can be described as follows. The hole and particle indices  $i$  and  $a$  are split into partitions, indicated as  $i_p$  and  $a_p$  :



In the example above  $i_p = 2$  refers to the occupied orbitals 12 to 23 (likewise for the virtual orbitals). A tiled multidimensional tensor  $X(a, b, \dots, i, j, \dots)$  is indicated as  $X(a_p, b_p, \dots, i_p, j_p, \dots)$ . In the picture of the  $T(i, j, a, b)$  array on the previous page, these are the grey and white squares. A tile refers to a dense block of the tensor where the indices are given by  $a \in a_p, b \in b_p$  etc. Each of these rectangular blocks of amplitudes can be accessed as a (in this case) 4 dimensional array.

A partition of an index range is always associated with its spatial and spin symmetry information. The structure in the example above stems from :



For antisymmetric tensors like  $T(a < b, i < j)$  only the tiles of the diagonal and upper diagonal part are stored  $T(a_p < b_p, i_p < j_p)$  ( grey tiles in  $T(i, j, a, b)$  picture ).

For the diagonal tiles this means that elements are stored twice which is accounted for in the TCE generated code. This structure at hand, many problems in electronic structure codes can be addressed :

- **Memory bottleneck** : Full tensors might be too big to keep in memory. To avoid keeping full tensors in memory all index ranges are partitioned. The partition sizes can be adjusted such that a tile will always fit into memory.
- **Exploit spatial, spin and permutational symmetry** : Every index partition of a quantity is stored with a set of properties like spin and spatial symmetry. In every contraction if-statements avoid contracting zero tiles. The right loop structure is generated regarding permutational symmetry.
- **Resorting of tensors** : The dense rectangular form of the tiles makes it easy to resort indices as they are needed in the DGEMM calls of the contractions.

To summarize, the basic principle is the partitioning of index ranges and the resulting tiling of the tensors, which is designed regarding permutational, spatial and spin symmetry. The tile size is determined by the memory available, spin and spatial symmetry information of the tiles are contained in a set of functions.

The general structure of any tensor contraction routine that is generated by the tce will look like this (*note that below block is used as equivalent to tile*):

```
loop over target_tiles
  if (spin_target_tiles okay) then
    if (symmetry_target_tiles okay) then

      distribute task to nodes

      loop over contraction_tiles
        if (spin_contraction_tiles okay) then
          if (symmetry_target_tiles okay) then

            assign_memory(tensor1)
            get_block(tensor1)
            resort_block(tensor1)

            assign_memory(tensor1)
            get_block(tensor2)
            resort_block(tensor2)

            assign_memory(target)

            call dgemm(tensor1,tensor2,target)

            resort_block(target)
            add_block(target)

          endif
        endif
      end loop contraction_tiles

      end distributed process

    endif
  endif
end loop target_tiles
```

## 4 CCSD example

Coupled Cluster Equations :

$$\begin{aligned} |\Psi_{CC}\rangle &= e^{\hat{T}} |\Psi_0\rangle \\ \hat{H} e^{\hat{T}} |\Psi_0\rangle &= E e^{\hat{T}} |\Psi_0\rangle \end{aligned}$$

Amplitude Equations :

$$\langle \Psi | e^{-\hat{T}} \hat{H} e^{\hat{T}} |\Psi_0\rangle = 0$$

Truncation of the Cluster operator  $\hat{T}$  for CCSD:

$$\hat{T} = \hat{T}_1 + \hat{T}_2$$

$\hat{T}_2$  Amplitude Equations :

$$\langle \Psi_{ij}^{ab} | e^{-\hat{T}} \hat{H} e^{\hat{T}} |\Psi_0\rangle = 0$$

First step : list all terms contributing to the CCSD  $\hat{T}_2$  equations in second quantization using OCE syntax :

- write input by hand
- use AEE
- use GUI

## 4.1 OCE input syntax

One line for every contribution :

```
1.0/4.0 Sum( g5 g6 g7 g8 )
      v( g5 g6 g7 g8 )
      { h1+ h2+ p4 p3 } { g5+ g6+ g8 g7 }

1.0/4.0 Sum( g5 g6 g7 g8 p9 h10 )
      v( g5 g6 g7 g8 ) t( p9 h10 )
      { h1+ h2+ p4 p3 } { g5+ g6+ g8 g7 } { p9+ h10 }

1.0/4.0 Sum( g5 g6 p7 p8 h9 h10 )
      f( g5 g6 ) t( p7 p8 h9 h10 )
      { h1+ h2+ p4 p3 } { g5+ g6 } { p7+ p8+ h10 h9 }

1.0/16.0 Sum( g5 g6 g7 g8 p9 p10 h11 h12 )
      v( g5 g6 g7 g8 ) t( p9 p10 h11 h12 )
      { h1+ h2+ p4 p3 } { g5+ g6+ g8 g7 } { p9+ p10+ h12 h11 }

1.0/8.0 Sum( g5 g6 g7 g8 p9 h10 p11 h12 )
      v( g5 g6 g7 g8 ) t( p9 h10 ) t( p11 h12 )
      { h1+ h2+ p4 p3 } { g5+ g6+ g8 g7 } { p9+ h10 } { p11+ h12 }

1.0/4.0 Sum( g5 g6 p7 h8 p9 p10 h11 h12 )
      f( g5 g6 ) t( p7 h8 ) t( p9 p10 h11 h12 )
      { h1+ h2+ p4 p3 } { g5+ g6 } { p7+ h8 } { p9+ p10+ h12 h11 }

1.0/16.0 Sum( g5 g6 g7 g8 p9 h10 p11 p12 h13 h14 )
      v( g5 g6 g7 g8 ) t( p9 h10 ) t( p11 p12 h13 h14 )
      { h1+ h2+ p4 p3 } { g5+ g6+ g8 g7 } { p9+ h10 } { p11+ p12+ h14 h13 }

1.0/128.0 Sum( g5 g6 g7 g8 p9 p10 h11 h12 p13 p14 h15 h16 )
      v( g5 g6 g7 g8 ) t( p9 p10 h11 h12 ) t( p13 p14 h15 h16 )
      { h1+ h2+ p4 p3 } { g5+ g6+ g8 g7 } { p9+ p10+ h12 h11 } { p13+ p14+ h16 h15 }

1.0/24.0 Sum( g5 g6 g7 g8 p9 h10 p11 h12 p13 h14 )
      v( g5 g6 g7 g8 ) t( p9 h10 ) t( p11 h12 ) t( p13 h14 )
      { h1+ h2+ p4 p3 } { g5+ g6+ g8 g7 } { p9+ h10 } { p11+ h12 } { p13+ h14 }

1.0/32.0 Sum( g5 g6 g7 g8 p9 h10 p11 h12 p13 p14 h15 h16 )
      v( g5 g6 g7 g8 ) t( p9 h10 ) t( p11 h12 ) t( p13 p14 h15 h16 )
      { h1+ h2+ p4 p3 } { g5+ g6+ g8 g7 } { p9+ h10 } { p11+ h12 } { p13+ p14+ h16 h15 }

1.0/96.0 Sum( g5 g6 g7 g8 p9 h10 p11 h12 p13 h14 p15 h16 )
      v( g5 g6 g7 g8 ) t( p9 h10 ) t( p11 h12 ) t( p13 h14 ) t( p15 h16 )
      { h1+ h2+ p4 p3 } { g5+ g6+ g8 g7 } { p9+ h10 } { p11+ h12 } { p13+ h14 } { p15+ h16 }
```

*(note that in the real input file every contribution above is a single line of input, there must be no blank lines in between)*

## 4.2 CCSD python script

```
# AEE input generation for ccSD t2 equations
import aee
amp1=aee.Term(1.0,1.0,["T1"])
amp2=aee.Term(1.0,4.0,["T2"])
tccsd=aee.Term.add(amp1,amp2)
tccsd.show()
one=aee.Term(1.0,1.0,["F"])
two=aee.Term(1.0,4.0,["V"])
h=aee.Term.add(one,two)
h.show()
ccsd=aee.Sum.bch(h,tccsd,5)
ccsd.show()
doub=aee.Term(1.0,1.0,["bra2"])
doubles=aee.Sum([doub])
ccsdd=aee.Sum.multiply(doubles,ccsd)
ccsdd.show()
out=aee.Sum.addindices(ccsdd)
out.show()
aee.Sum.writefullformula(out,"ccsd_t2_aee.out")
```

```
import oce
#alternative input file :
# ccsdd=oce.readfromfile("ccsd_t2.in")
ccsdd=oce.readfromfile("ccsd_t2_aee.out")
ccsd_t2a=oce.ListOperatorSequences()
ccsd_t2a.join(ccsdd.performfullcontraction())
ccsd_t2=oce.ListOperatorSequences()
ccsd_t2.join(ccsd_t2a.deletedisconnected().simplify())
ccsd_t2.writetofile("ccsd_t2_oce.out")
```

```
import tce
ccsd=tce.readfromfile("ccsd_t2_oce.out")
ccsd=ccsd.breakdown()
ccsd=ccsd.fullyfactorize()
ccsd.fortran77("ccsd_t2").writetofile("ccsd_t2_tce")
```

### 4.3 OCE procedures

#### **readfromfile**

read equations into OCE

```
[ + 0.25 ] * Sum ( g5 g6 g7 g8 ) * v ( g5 g6 g7 g8 )  
            * <0|{ h1+ h2+ p4 p3 } { g5+ g6+ g8 g7 }|0>  
  
[ + 1.0 ] * Sum ( g5 g6 p7 h8 ) * f ( g5 g6 ) * t ( p7 h8 )  
            * <0|{ h1+ h2+ p4 p3 } { g5+ g6 } { p7+ h8 }|0>
```

↓

#### **performfullcontraction**

perform contractions according to Wick's theorem

```
[ + 1.0 ] * v ( p3 p4 h1 h2 )  
[ - 1.0 ] * f ( p4 h1 ) * t ( p3 h2 )  
[ + 1.0 ] * f ( p3 h1 ) * t ( p4 h2 )  
[ + 1.0 ] * f ( p4 h2 ) * t ( p3 h1 )  
[ - 1.0 ] * f ( p3 h2 ) * t ( p4 h1 )
```

↓

#### **deletedisconnected**

delete disconnected terms for MBPT/CC  
(can be skipped !)

```
[ '[ + 1.0 ] * v ( p3 p4 h1 h2 ) ]'
```

↓

#### **writetofile**

dump to outputfile to be read by TCE

## 4.4 TCE procedures

**readfromfile**  
read equations into TCE

```
[ + 1/2 - 1/2 * P( p4 p3 h1 h2 => p3 p4 h1 h2 ) ] * Sum ( h5 h6 )
  * t ( p4 h5 ) * t ( p3 h6 ) * v ( h6 h5 h1 h2 )

[ + 1/2 - 1/2 * P( p4 p3 h2 h1 => p3 p4 h2 h1 ) - 1/2 * P( p4 p3 h2 h1 => p4 p3 h1 h2 )
  + 1/2 * P( p4 p3 h2 h1 => p3 p4 h1 h2 ) ] * Sum ( h5 h6 p7 )
  * t ( p4 h5 ) * t ( p3 h6 ) * t ( p7 h2 ) * v ( h6 h5 h1 p7 )
```

↓

**breakdown**  
strength reduction, term by term

```
i0 ( p3 p4 h1 h2 ) += [ + 1/2 - 1/2 * P( h1 h2 p3 p4 => h1 h2 p4 p3 ) ] * Sum ( h5 )
  * i1 ( h5 p3 h1 h2 ) * t ( p4 h5 )
i1 ( h5 p3 h1 h2 ) += [ - 1 ] * Sum ( h6 )
  * v ( h5 h6 h1 h2 ) * t ( p3 h6 )

i0 ( p3 p4 h1 h2 ) += [ + 1/2 - 1/2 * P( h1 h2 p3 p4 => h1 h2 p4 p3 ) ] * Sum ( h5 )
  * i1 ( h5 p3 h1 h2 ) * t ( p4 h5 )
i1 ( h5 p3 h1 h2 ) += [ + 1 ] * Sum ( h6 )
  * i2 ( h5 h6 h1 h2 ) * t ( p3 h6 )
i2 ( h5 h6 h1 h2 ) += [ - 1 + 1 * P( h1 h2 p3 p4 => h2 h1 p3 p4 ) ] * Sum ( p7 )
  * v ( h5 h6 h1 p7 ) * t ( p7 h2 )
```

↓

**fullyfactorize**  
do global factorization by joining terms with common intermediates, obtain  
operation tree

```
i0 ( p3 p4 h1 h2 ) += [ - 1/2 + 1/2 * P( h1 h2 p3 p4 => h1 h2 p4 p3 ) ]
  * Sum ( h5 ) * t ( p3 h5 ) * i1 ( h5 p4 h1 h2 )
i1 ( h5 p3 h1 h2 ) += [ - 1 ] * Sum ( h6 ) * t ( p3 h6 ) * i2 ( h5 h6 h1 h2 )
i2 ( h5 h6 h1 h2 ) += [ + 1 ] * v ( h5 h6 h1 h2 )
i2 ( h5 h6 h1 h2 ) += [ - 1 + 1 * P( h1 h2 p3 p4 => h2 h1 p3 p4 ) ]
  * Sum ( p7 ) * t ( p7 h1 ) * v ( h5 h6 h2 p7 )
```

↓

**fortran77,writetofile**  
generate Fortran77 code from operation tree

## 5 TCE related www links

TCE group home page (members, projects, references) :

<http://www.cis.ohio-state.edu/gb/TCE/>

report on TCE :

<http://www.e4engineering.com/item.asp?id=49806>

NWChem home page :

<http://www.emsl.pnl.gov/docs/nwchem/nwchem.html>

Global Arrays home page :

<http://www.emsl.pnl.gov/docs/global/>

Python interpreted, interactive, object-oriented programming language :

<http://www.python.org>

## 6 Python literature

Making Use of Python, Rashi Gupta  
John Wiley & Sons; 1st edition (June 15, 2002)  
ISBN: 0471219754

Learning Python, David Ascher, Mark Lutz  
O'Reilly & Associates; 2nd edition (December 2003)

ISBN: 0596002815  
Python in a Nutshell, Alex Martelli  
O'Reilly & Associates; 1st edition (March 2003)  
ISBN: 0596001886

# Index

TeXoutput, 16

AEE usage, 8

antisymmetry of tensors, 21

Authors, 2

BCH expansion, 10

breakdown (TCE), 15

CCSD example, 23

Collaborators, 2

contraction routine, 19, 22

Coupled Cluster equations, 23

deletedisconnected (OCE), 14

email addresses, 2

EOM-CCSD example (GUI), 6

exit python, 5

factorization, 15

Fortran comments, 19

Fortran77, 16

Fortran90, 16

fullyfactorize (TCE), 16

Genetic Algorithms, 15

Global Arrays, 28

GUI functions, 6

GUI input, 6

GUI screenshot, 6

GUI usage, 5

import command (python), 5

Introduction, 3

main routine, generated, 18

memory bottleneck, 21

modules overview, 4

NWChem, 28

OCE input, 12, 24

OCE output, 14

OCE usage, 13

operation tree (TCE), 16

operator sequence, 8, 13

partitioning of indexes, 20

performfullcontraction (OCE), 14

python homepage, 28

python interactive mode, 5

python literature, 29

python usage, 5

resorting of tensors, 21

show function (python), 5

Simulated Annealing, 15

structure of generated code, 18

Sum object (AEE), 9

symmetry, 21

TCE input, 14

TCE links, 28

TCE python script, CCSD, 25

TCE usage, 15

tensor contraction, 13

Term object (AEE), 9

tiling scheme, 19

Wick's Theorem, 4, 12, 13, 26