Flexible matrix realignment

In this document, we will discuss a convenient (and hopefully efficient) way of realigning the matrix elements of the different tensors entering the CCD amplitude equation, so that all diagrams may be calculated by matrix multiplication.

In its current implementations (in python/numpy and c++/armadillo), I utilize sparse matrices to store all elements. This has some advantage when it comes to memory consumption, since only nonzero elements are stored. The current drawback seems to be the efficiency of matrix multiplications, since this process consumes notably more time for sparse matrices than the block diagonalization approach. This is due to the way the elements are stored.

The CCD amplitude equation

The T2 amplitude for the CCD truncation is

$$t_{ij}^{ab}\epsilon_{ij}^{ab} = \frac{1}{2}\sum_{cd}\langle ab||cd\rangle t_{ij}^{cd} + \frac{1}{2}\sum_{kl}\langle kl||ij\rangle t_{kl}^{ab} + \hat{P}(ij|ab)\sum_{kc}\langle kb||cj\rangle t_{ik}^{ac} + \frac{1}{4}\sum_{klcd}\langle kl||cd\rangle t_{ij}^{cd}t_{kl}^{ab} + \hat{P}(ij)\sum_{klcd}\langle kl||cd\rangle t_{ik}^{ac}t_{jl}^{bd} - \frac{1}{2}\hat{P}(ij)\sum_{klcd}\langle kl||cd\rangle t_{ik}^{dc}t_{lj}^{ab} - \frac{1}{2}\hat{P}(ab)\sum_{klcd}\langle kl||cd\rangle t_{lk}^{ac}t_{ij}^{db}$$

This equation is commonly solved iteratively by guessing some initial amplitude, typically the ones corresponding to the MBPT(2) energy, then solving for the RHS to yield the new approximation. The iterative process is halted when some convergence criteria is fulfilled, possibly when the change in amplitudes between the iterations subpasses some threshold or when the resulting energy converges towards some value.

The structure of this equation motivates a subdivision into *linear* and *quadratic* terms in order of the unknown amplitudes t_{ij}^{ab}.

By denoting linear terms by L_n and quadratic terms by Q_n, we may then express the equation as

$$t_{ij}^{ab}\epsilon_{ij}^{ab} = L_1(t_{ij}^{ab}) + L_2(t_{ij}^{ab}) + L_3(t_{ij}^{ab}) + Q_1(t_{ij}^{ab}) + Q_2(t_{ij}^{ab}) + Q_3(t_{ij}^{ab}) + Q_4(t_{ij}^{ab})$$

Each term on the RHS above corresponds to a one of the diagram formed by contracting the similarity transformed hamiltonian with the exponential ansatz.

In the following sections, we will discuss how these terms may be calculated as matrix multiplications.

Setting up the matrices

Every factor in the terms above is a tensor of rank 4. To utilize the advantages of matrix multiplications, we need to represent these tensors as matrices (tensors of rank 2). In effect, we need a unambiguous mapping from rank 4 to rank 2.

One very straightforward such mapping is derived by considering the indexes and their associated length;

$$p = [0, 1, 2, 3, \ldots, N_p]$$

A matrix will have only two indices, so we then map the amplitudes and interactions onto matrices by for example

$$\langle pq||rs\rangle = \langle p + qN_p||r + sN_r\rangle$$

Now we have the rows of the matrix contained in the bra part of the RHS above, and the columns is contained in the ket.

Example usage

Since a high accuracy calculation will involve a large number of particle states, the L1 term is computationally expensive (it is a sum over particle states). A naive implementation will typically consist of four nested for-loops (for a,b,c,d), and will require a lot of flops.

If we instead map it onto matrices, so that

$$\sum_{cd}\langle ab||cd\rangle t_{ij}^{cd} = \sum_{cd}\langle a + bN_a||c + dN_c\rangle t_{i+jN_i}^{c+dN_c} = \sum_{\gamma}\langle \alpha||\gamma\rangle t_{\beta}^{\gamma}$$

We easily see that the whole diagram may be calculated as a matrix multiplication

$$L_1 = V_{\gamma}^{\alpha} * T_{\beta}^{\gamma}$$

Flexible indexing

Because of the occurances of indices in the contributions L_1, L_2 and Q_1, these diagrams are especially straighforward to set up as matrix multiplications. In the remaining diagrams, we will need to realign the matrix elemenent before the multiplication is performed.

In the following we will utilize the following matrix notation:

$$t_{ij}^{cd} \equiv |cd\rangle T\langle ij|$$

The L_1 term is then calculated as

$$L_1 = (|a + bN_a\rangle V\langle c + dN_c|)(|c + dN_c\rangle T\langle i + jN_i|)$$

The alignment problem is related to diagrams where indices does not occur in corresponding bra and kets as for L_1 above. For example, we have for the L_3 term (prefactors and permutations left out)

$$L_3 = \sum_{kc} \langle kb||cj \rangle t_{ik}^{ac}$$

To set up a matrix for V and T, we need the indices k and c to coincide in the bra and ket of V and T respectively. In effect, we want

$$\tilde{L}_3 = (|b + jN_b\rangle V \langle k + cN_k|)(|k + cN_k\rangle T \langle a + iN_a|)$$

Where the indices of $\tilde{L}_3$ does not align with the indices of L_3, but since the elements are the same we only need to perform a simple realignment back to the original column- and row indexing.

$$\tilde{L}_3 \rightarrow L_3$$

The process is maybe most easily understood as a *generalized transpose* for tensors with $rank > 2$.

Subdivision of the interaction matrix

As may be seen from the CCD equation, only certain elements from the interaction matrix actually enters the calculation.

This makes it convenient to subdivide the interaction matrix into blocks in

V_{pp}^{pp}
V_{hh}^{hh}
V_{pp}^{hh}
V_{hh}^{pp}
V_{ph}^{hp}

In these matrices h and p denotes hole and particle states. Their location in the full interaction matrix is visualized below.

	hh	**hp**	**ph**	**pp**
hh	V_{hh}^{hh}			V_{pp}^{hh}
hp			V_{ph}^{hp}	
ph				
pp	V_{hh}^{pp}			V_{pp}^{pp}

A lot of symmetries is also involved in these elements, so we do not need to actually calculate all quantities involved prior to the CCD calculation.

The actual alignments

We will now derive the various realignments that we need to perform to solve the full CCD amplitude equation by matrix multiplications. These expressions are meant to correspond with the actual implementation in fermicc, so instead of having direct corresponding indices, we use p, q, r, s for the first, second, third and fourth index respectively (as they occur in the tensors "native" representation.)

The realignments/permutations are listed in the table below. Note that any prefactors or permutations is omitted from the expressions.

Table: Realignment of matrix elements in interaction and amplitudes

Diagram	**Expression**	**Aligned multiplication**	**Interaction**	**Amplitude(1)**
L_1	$\sum_{cd} \langle ab \Vert cd \rangle t_{ij}^{cd}$	$(\vert ab\rangle V \langle cd\vert)(\vert cd\rangle T \langle ij\vert)$		
L_2	$\sum_{kl} \langle kl \Vert ij \rangle t_{kl}^{ab}$	$(\vert ab\rangle T \langle kl\vert)(\vert kl\rangle V \langle ij\vert)$		
L_3	$\sum_{kc} \langle kb \Vert cj \rangle t_{ik}^{ac}$	$(\vert jb\rangle \tilde{V} \langle ck\vert)(\vert ck\rangle \tilde{T} \langle ai\vert)$	$V_{rs}^{pq} = \tilde{V}_{rp}^{sq}$	$T_{rs}^{pq} = \tilde{T}_{pr}^{qs}$
Q_1	$\sum_{klcd} \langle kl \Vert cd \rangle t_{ij}^{cd} t_{kl}^{ab}$	$(\vert ab\rangle T \langle kl\vert)(\vert kl\rangle V \langle cd\vert)(\vert cd\rangle T \langle ij\vert)$		
Q_2	$\sum_{klcd} \langle kl \Vert cd \rangle t_{ik}^{ac} t_{jl}^{bd}$	$(\vert ai\rangle T \langle kc\vert)(\vert kc\rangle V \langle ld\vert)(\vert ld\rangle T \langle bj\vert)$	$V_{rs}^{pq} = \tilde{V}_{qs}^{pr}$	$T_{rs}^{pq} = \tilde{T}_{sq}^{pr}$
Q_3	$\sum_{klcd} \langle kl \Vert cd \rangle t_{ik}^{dc} t_{lj}^{ab}$	$(\vert abj\rangle T \langle l\vert)(\vert l\rangle V \langle kcd\vert)(\vert kcd\rangle T \langle i\vert)$	$V_{rs}^{pq} = \tilde{V}_{prs}^{q}$	$T_{rs}^{pq} = \tilde{T}_{r}^{pqs}$
Q_4	$\sum_{klcd} \langle kl \Vert cd \rangle t_{lk}^{ac} t_{ij}^{db}$	$(\vert a\rangle T \langle klc\vert)(\vert klc\rangle V \langle d\vert)(\vert d\rangle T \langle bij\vert)$	$V_{rs}^{pq} = \tilde{V}_{s}^{pqr}$	$T_{rs}^{pq} = \tilde{T}_{srq}^{p}$

Implementation

A standard format for representing sparse matrices is the COOrdinate format. In this format the matrix elements are stored in three arrays, one for the actual values, and two for each coordinate in the matrix. All elements not stored in these arrays are by default zero.

For example, the matrix below

	0	**1**	**2**	**3**
0		1		3
1				
2				

3	2			

May be represented in the following way

element	0	1	2
values	1	3	2
row	0	0	3
column	1	3	0

This is however not the most optimal (least space consuming) representation, so formats such as CSC (compressed sparse column) or CSR (compressed sparse row) is commonly utilized. Armadillos sp_mat object is of the first type, meaning that the array containing the column indices is compressed by replacing it with a pointer to the first element occuring in each column. This way, the index in the column pointer array denotes the actual column in which the following elements occur (up until the pointer to the next element). To access the actual column index of any element in the matrix, we therefore have to unpack this array first.

http://netlib.org/linalg/html_templates/node92.html

One such unpacking procedure written for the python/numpy implementation of the CCD calculation is given below

In [3]:
```
def unpack_indptr(indptr):
    #Unpack compressed indices
    #indptr is the array to be uncompressed
    I =zeros(indptr[-1], dtype = int)
    for i in range(len(indptr)-1):
        I[indptr[i]:indptr[i+1]] = i
    return I
```

When the matrices are represented in this way, the realignment procedure becomes trivial. Given elements, row- and columnindices, we may easily obtain the rank 4 indices, as is done in the following python example:

In [4]:
```
from numpy import *
Np = 14
Nq = 14
Nr = 14
Ns = 14

row = arange(Np*Nq)
col = arange(Nr*Ns)

p = row//Np
q = row%Np
r = col//Nr
s = col%Nr
```

Next up, an example of how to recast the elements to a "realigned" matrix

In [5]:
```
row = q + r*Nq
col = s + p*Ns

#New matrix dimensions: (Nq*Nr), (Ns*Np)
#Matrix elements "val" remain the same
```

Density of the interactions

Due to the many kroenecker deltas occuring in the HEG interaction, we find that the interaction matrix has a denisty of approx < 1%, meaning that this is a case where utilizing sparse storage is reasonable.

Pros and cons of the sparse scheme

The main drawback of the sparse approach will however be the efficiency of the matrix multiplications when compared to the block diagonalization scheme. When block diagonalized, the calculations may be performed in mere seconds.

The main advantage of this approach is, to my expection, that the implementation of the CCSDT truncation will be very straightforward codewise, low on memory consumption, and possible to perform also as serial computations. (unless computational cost exceeds the expected progression).

There is a number of routes that may possibly speed up the sparse matrix multiplications. I could continue optimizing the COO-scheme, find a replacement for armadillo, or possibly try a CUDA im

One is to implement a similar subdivision of the matrix elements that is done for the block diagonal scheme. By simply skipping casting the elements to a sparse matrix in the first place, I attempted to just perform the multiplication directly on the elements stored in a COOrdinate format with a subdivision similar to the block diagonal scheme. This successfully decreased the time spent on multiplication with a factor of about 0.007 when compared to the naive implementation, but the native scipy.sparse multiplication operation still outperformed this algorithm by a factor of about 0.001 when compared to the optimized implementation. (meaning a factor of about 0.00007 compared to the naive scheme).

On the other hand, Armadillo's native sparse multiplication did not compare well to the scipy algorithm, as it was only about a factor of 0.1 faster than the optimized algorithm.

Benchmarking sparse multiplication

Library/Algorithm	**Time (s)**	**Fractional time**
Naïve	2.07011318512	1
Optimized	0.042023196118	0.02029995094957284
Scipy	0.000879618572071	0.00042491327449808525
Armadillo	0.025932	0.012526851278664156

The table lists the time spent on performing a multiplication of the V_{pp}^{pp} matrix with itself, with a dimension of 576x567 and a density of about 0.75 % . (ratio of nonzero elements)

It should be noted that the results from armadillo does not directly compare as they were calculated in a virtual ubuntu environment. (Oracle VM VirtualBox, no cap on CPU and plenty of RAM).

Considering alternative libraries

Mainly three libraries for the C++ implementation are up for consideration:

1. Eigen (http://eigen.tuxfamily.org/index.php?title=Main_Page)
2. CSparse (http://www.cise.ufl.edu/research/sparse/CSparse/)
3. CUSparse https://developer.nvidia.com/cuSPARSE

The first two seems to have optimized support for sparse matrix multiplication (and a lot of other functionality). The third certainly meets this criteria, but it is also dependent on the GPU/CUDA. Scaling should also be taken into account, as libraries such as CUDA may or may not (https://www.cs.fsu.edu/research/projects/aseshpande.pdf https://www.cs.fsu.edu/research/projects/rose_report.pdf) perform slower for small systems compared to libraries utilizing the CPU. Another consideration in this matter is that the upcoming CCSDT implementation should be expected to require parallellization (in MPI), so it will be of great advantage if the implementation is easily extended to parallel computation.

```
In [ ]:
```