Research project #2. October 30

Topics:  *second quantization (normal order and expectation values)*  
        *Wick’s theorem*

| An alternative to the homework: a research project. |

Write a computer program which will implement Wick’s theorem using any computer language you know or Maple/Mathematica.

What will this program do?

Given input (in any form you want) of the type \( a_j a_i a_j a_i \) or any other product of creation or annihilation operators it will return the sum of the normal-ordered terms. In this example the output is

\[
a_j a_i a_j a_i = \delta_{ii} \delta_{jj} - \delta_{jk} \delta_{il} - a_j^\dagger a_j : a_i^\dagger a_i : + a_i^\dagger a_j : a_j^\dagger a_i : + a_i^\dagger a_i a_j a_j : \]

YOU ARE NOT EXPECTED TO INTERCHANGE OPERATORS INSIDE THE NORMAL-ORDERED TERMS: YOU CAN KEEP THE ORIGINAL ORDER OF THE OPERATORS AND MAKE THE CONTRACTIONS.

The output

\[
a_j a_i a_j a_i = a_j a_i a_j a_i : + a_j a_i : a_i a_j : : : \delta_{ii} - \delta_{jk} : a_i a_j : = - a_j a_i : a_i a_j : + \delta_{ii} : a_i a_j : + \delta_{ll} \delta_{jj} - \delta_{jk} \delta_{il} \]

is also fine as it is exactly identical to the result above.

The code must work for any product of annihilation and creation operators up to 10 operators. It does not matter how output looks as long as it is correct and I can recognize it in terms of operators and delta functions. You may want to attach an explanation of your designations if they are not obvious. You must verify that the code works properly and be reasonably sure that it will continue working properly for more complex cases.

What you are supposed to submit:

DUE: November 13
1) source code (program itself)
2) printouts (input/output) of at least three correct examples. At least two must contain four or more operators.
3) I will give you a test case and you will e-mail me (or print it out and put in my mailbox) the output for this case. If it did not work properly you will have a week to fix it. I will give you a correct result.

NOTE: this research project carries bonus points: total is 15 instead of 9 in this homework.
You may chose to return both research project and homework. The points for BOTH research projects and homework will be counted. Note: bonus points are added to the total homework grade. If you exceed maximum number of points for the homework the remainder will carry over between homework/exams breakdown with the corresponding 50% contribution.

HINT: if you do the research project it will do most of the homework for you (#1 and #2).

**HOW TO PROCEED: STEP-BY-STEP INSTRUCTIONS**

**NOTE:** since I use FORTRAN I included some examples in FORTRAN. The general algorithm should be applicable to any language. I will e-mail you my source code after all of you who decided to attempt research project have returned the final version of your work to me. I will include detailed comments into my source code. I have used only very basic FORTRAN commands when writing this code so it should run fine with g77 compiler.

You do not have to follow the algorithm below either in part or in its entirety. You are most welcome to do it differently. There are certainly other ways to pursue this project.

**STEP 1.** Write input in a form which can be understood (read in) by a computer.

**Example:** instead of $a_ia_ia_j^ia_j^i$ write input as $-j-i+k+l$, i.e. put “−” for the annihilation operator, “+” for creation operator and write operator’s index after it.

**STEP 2.** Read in this input to your program.
**Example:** store signs “–” and “+” into array $a1(i)$ and indexes into array $a2(i)$ where $i=1,\ldots,N$ for $N$ operators. In FORTRAN these arrays will be of `character*1` type. Make sure that you have

```
character*1 a1,a2
```

statement in the first line of any subroutine where $a1,a2$ are used. Store the number of operators in the input as well (you can count then when you read them in).

In our example $\begin{array}{c} a_j a_i a_k a_l \end{array}$ $(-j-i+k+l)$

```
a1(1)=a1(2)='-', a1(3)=a1(4)='+'
a2(1)='i', a2(2)='j', a2(3)='k', a2(4)='l'
```

**Note:** you may chose to read in “fixed input” where no spaces are allowed and you always write them in the same way (sign first and index second) or you may write more elaborate version which will skip the space and will understand $-j-i+k+l$ or $j-i-k+1$ just as well as $-j-i+k+l$. You are not required, of course, to write an elaborate version. I have included the second version in my code.

In general, more elaborate read in version is useful in codes which you expect other people to use or which you expect to use for a long time as it minimizes input errors.

**STEP 3.** Set up the general structure of each TERM.

Now your program has the input information so it has enough information to write out first term with zero contractions, i.e. just re-write the input (you may or may not write normal product signs “::” before and after the operators as everything on the right-hand side of Wick’s theorem is normal-ordered anyway).

**Example:** in the case of $a_j a_i a_k a_l$ the first term is just $:: a_j a_i a_k a_l ::$.

However, I recommend defining the general term structure before proceeding any further.

Lets look at our terms in our example:

```
a_j a_i a_k a_l = a_i a_j a_k a_l = a_j a_i a_k a_l = a_i a_j a_k a_l = a_j a_i a_k a_l = a_i a_j a_k a_l = a_j a_i a_k a_l = a_i a_j a_k a_l
```

First, lets note that it is exactly

```
a_j a_i a_k a_l = a_i a_j a_k a_l + a_j a_i a_k a_l = a_j a_i a_k a_l - a_i a_j a_k a_l
```

This expression is still normal ordered and is easier to program since we can keep
original operator order.

YOU ARE NOT EXPECTED TO INTERCHANGE OPERATORS INSIDE THE
NORMAL ORDERED TERMS: YOU CAN KEEP THE ORIGINAL ORDER OF
THE OPERATORS AND MAKE THE CONTRACTIONS.

This expression contains 7 terms: 1 with zero contractions, 4 terms resulting from
making one contraction, and 2 resulting from making 2 contractions. Each term can be
separated into three parts: its sign, delta functions, and operators. Some terms may
contain no delta functions or no operators. Such definition is very general: no matter
how elaborate is the initial expression your terms will still only contain sign, delta
functions, and normal-ordered operators.

Example: I used the following description for each term:

**********************************************************
* STORAGE OF THE RESULT: TERM (i) IS DEFINED BY: *
* * 1) sign: defined by one array ns(i) with values +1 or -1 *
* * 2) deltas: defined by three arrays *
* * ndelta(i): number of deltas in this term *
* d1(i,j) : first delta index, j=1,ndelta(i) *
* d2(i,j) : second delta index, j=1,ndelta(i) *
* * 3) operators: defined by three arrays *
* nopt(i): number of operators in this term *
* o1(i,j): defines if it is a creation or an annihilation *
* operator. It has the value '+' or '-'. *
* j=1,nopt(i) *
* o2(i,j): index of the operator, j=1,nopt(i) *

* Example: TERM #10: \(-\delta_{ij}\delta_{kl}:a_d^\dagger a_b:\)

* ns(10)=-1 ndelta(10)=2 d1(10,1)='i' d2(10,1)='j'
I store all the current terms in the common block /output/:  
parameter (nn=10,nt=10000)  
common /output/ o1(nt,nn),o2(nt,nn),d1(nt,nn),d2(nt,nn),nopt(nt),  
*ndelta(nt),ns(nt),iterm  

All of the arrays were introduced above. The last variable iterm is the number of the terms.

**STEP 4.** Define the first term: no contractions. Just copy the input into Term # 1 in our term format defined in step 3.  
**Example:** iterm=1, ns(1)=1, number of operators nopt(1)=N, number of delta functions ndelta(1)=0 and the operators arrays o1(1,i) and o2(1,i) are our input a1(i) and a2(i) arrays.

**STEP 5.** Decide on algorithm you are going to use to do multiple contractions. Clearly, there are different ways to do it. Here is the algorithm that I come up with:  
A) Take our initial Term#1 which we defined in Step 4. Make a copy of it to the /work_in/ common block (I added letter “w” to all arrays to distinguish them from the block /output/ arrays):  
common /work_in/ o1w(nt,nn),o2w(nt,nn),d1w(nt,nn),d2w(nt,nn),noptw(nt),  
*ndeltaw(nt),nsw(nt),itw  
The last variable “itw” is the number of terms in our /work_in/ block.  
B) Write a subroutine “contraction” which is capable of making all possible one contractions for all of the terms in the common block /work_in/.  
**Example:** if input in /work_in/ is : $a_j a_k a_l a_m$ : it will produce four terms:  
Term#1 = $a_j a_k : \delta_{jl}$, Term#2 = $-\delta_{jl} : a_j a_k$ ; Term#3 = $- : a_j a_k : \delta_{jl}$, Term#4 = $\delta_{jl} : a_j a_k$ :  
C) All new terms are written to /work_out/ common block:
Note that this routine makes one contraction only. The great thing about this structure is that you do not need to write code to do multiple contractions; you can simply use contraction subroutine over and over again until there are no more contractions to make.

D) Loop this process over: 1) copy /work_out/ to output common block

2) copy /work_out/ block to /work_in/ block

3) erase /work_in/ block

4) call contraction subroutine again (it always makes one contraction in terms from /work_in/ and writes them to /write_out/.

5) keep doing it until step 4) produced no contractions.

E) Second call to contraction subroutine produces terms with two contractions. There is one problem with this scheme: this procedure will clearly produce duplicate terms. 

Example: making one contraction with terms

Term#1 = \(a_j a^l \delta_{il}\), Term#2 = \(-\delta_{jk} a_j a^l \delta_{il}\), Term#3 = \(-a_j a^l \delta_{jl}\), Term#4 = \(\delta_{jl} a_j a^l \delta_{il}\);

will produce terms

Term#1 = \(\delta_{jl} \delta_{ik}\), Term#2 = \(-\delta_{jk} \delta_{il}\), Term#3 = \(-\delta_{jk} \delta_{il}\), Term#4 = \(\delta_{jl} \delta_{ik}\).

So we get four terms instead of two. It can be simply corrected: each time we run contraction subroutine we search for the same terms and “erase” them. It is best to put this procedure in a separate subroutine. The easiest way to do it is: take term 1, look for the same term. Suppose term 6 is the same as term 1. Set \(ns(6)\) to zero: this will clearly identify this term. Repeat until there are no identical terms. Now make a correction to the loop over procedure:

1) copy /work_out/ to output common block, omit all terms with ns(i)=0

2) copy /work_out/ block to /work_in/ block, omit all terms with ns(i)=0

3) erase /work_in/ block

4) call contraction subroutine again

4a) call search_same routine. Mark all identical terms by setting ns(i)=0

5) keep doing it until step 4) produced no contractions.
STEP 6. Write out the result which is in block /output/. There are a lot of options of how to do it.

First (and the simplest one) is to just write out terms line by line.

**Example:** Term\#2 = $-\delta^i_j : a^\dagger_j : a^\dagger_l :$ write out $- jk -i+l$

Second, we can add some extra characters to the output.

**Example:** Term\#2 = $-\delta^i_j : a^\dagger_j :$ write out $- \delta(jk) : a(i) a+(l) :$

and either write terms line by line or write the output in complete lines.

Third, we can write output into the tex file.

**Example:**

```latex
\documentclass[12pt]{article}
\begin{document}
\begin{eqnarray*}
&& a^\dagger_j \ a_i \ a^\dagger_k \ a^\dagger_l = : a^\dagger_j \ a_i \ a^\dagger_k \ a^\dagger_l : - \delta_{jk} : a_i \ a^\dagger_l : + \delta_{jl} : a_i \ a^\dagger_k : + \delta_{ik} : a^\dagger_l : - \delta_{il} : a^\dagger_k : \\
&& - \delta_{jk} \delta_{il} + \delta_{jl} \delta_{ik} \\
\end{eqnarray*}
\end{document}
```

You can design any output you like. You will need to attach an explanation if your output is not self-explanatory.

**ADDITIONAL NOTES: HOW TO WRITE A “CONTRACTION” SUBROUTINE.**

The only non-zero contractions come from $: a^\dagger_j : a^\dagger_l :$ pairs, with annihilation operator on the left and creation operator on the right. Therefore, we carry out the following steps for the each term #it in the /work_in/ block to make a contraction:

**STEP 1.** Find the first operator with $o1(it,i) ='-'$.

**STEP 2.** Starting from index $i+1$ find the first operator with $o1(it,j) =$’+’. We found our non-zero contraction.

**STEP 3.** Define the new term. We will call it #1.

Find the sign of the contraction ($-$ if $j-i$ is even and $+$ if $j-i$ is odd).

Multiply it with the sign of the term #it.
Define the number of operators $n_{optz(1)}$ which is $n_{optw(it)}-2$.
Define the number of deltas $n_{deltaz(1)}$ which is $n_{deltaw(it)}+1$.
Define deltas: copy the original deltas from term #it and define the new one from the indices of the $(i,j)$ operators.
Define operators: copy the original operators from term #it erasing the $(i,j)$ operators.

**STEP 4.** Loop over steps 1-3 until you find all the contractions. Count the number of new terms.

**STEP 5.** Carry over steps 1-4 for all terms in block /work_in/.