```
(*******DiracQ_Package*******)

(********August 2015**********)

(*
Authors:   John Wright
           University of California, Santa Cruz
           Santa Cruz, CA
           95060
           jgwright1986@gmail.com

           B. Sriram Shastry
           University of California, Santa Cruz
           Santa Cruz, CA

Any comments, bug reports, suggestions etc welcomed. Contact us at DiracQ@gmail.com

Context: DiracQ

History: Created in 2013 by John Wright and B. Sriram Shastry. Revised in 2015

© Copyright: John Wright and Sriram Shastry, 2013.

Distributed under GNU General Public License, version 2
Read the GNU General Public License at:

http://www.gnu.org/copyleft/gpl.html

*)

BeginPackage["DiracQ`"];
```

Usage Statements

```
Commutator::usage =
"Commutator is used to calculate the commutators of expressions involving operators with known commutation relations. Commutator[A,B] is defined as AB-BA.";
AntiCommutator::usage =
"AntiCommutator is used to calculate the anticommutators of
    expressions involving operators with known anticommutation relations. AntiCommutator[A,B] is defined as AB+BA.";

CommuteParts::usage =
"CommuteParts[A,B,C] will reverse the order of the noncommuting objects specified by the lists B and C. B and C are lists of
    consecutive noncommutative objects found in the expression A specified by numerical ordering of the noncommutative objects found in
    A. Therefore, to permute the second and third NCOs found in A with the fourth NCO found in A, B would be {2,3}, and C would be {4}.";

ProductQ::usage =
  "ProductQ gives the product of two expressions involving terms that are noncommutative objects. ProductQ should be used in place of the standard Mathematica
    function NonCommutativeMultiply for combining expressions. ProductQ can be called as a function with two arguments or
    as the CircleTimes symbol ESCc*ESC used between two expressions. Operator product definitions will be applied by default.";
```

```
AddOperator::usage =
  "AddOperator allows users to expand the number of symbols that can be specified as noncommutative objects. The argument of AddOperator is the symbol that
    represents the new operator. Algebraic relations for new operators such as basic commutators, anticommutators, and products must be defined
    by the user. To define a basic commutator for two operators's α and β, input CommutatorDefinition[α,β]:=_ , where the blank is the definition.
    AntiCommutatorDefinition[α,β] is the equivelant function for anticommutators and OperatorProduct[α,β] is the equivelant function for definition of operator
    products. Any number of such definitions can be input. If a function calls a definition that has not been input by the user the output will read 'Null'.";

DeleteOperator::usage = "DeleteOperator will remove a user defined operator from
    the list of possible operators. The argument of delete operators is the symbol by which the operator represented.";

Organize::usage =
  "Organize is the function that enables the DiracQ package to understand user input. Organize takes a mathematical expression as input and yields a nested
    list that contains the atoms of the input ordered according to their properties. Numbers, summed indices, c numbers, and q numbers
    are separated into groups. Each term of the input separated by plus sign constitutes a separate list of items in the output. Example:
Organize[(#) \!\(\*UnderscriptBox[\"∑\", \"index\"]\)(c #)*(q #)]
={{#,{index},{c #},{q #}}}

Organize[(\!\(\*SubscriptBox[\"#\", \"1\"]\)) \!\(\*UnderscriptBox[\"∑\", SubscriptBox[\"index\",
    \"1\"]]\)(c \!\(\*SubscriptBox[\"#\", \"1\"]\))*(q \!\(\*SubscriptBox[\"#\", \"1\"]\))+(\!\(\*SubscriptBox[\"#\", \"2\"]\))
    \!\(\*UnderscriptBox[\"∑\", SubscriptBox[\"index\", \"2\"]]\)(c \!\(\*SubscriptBox[\"#\", \"2\"]\))*(q \!\(\*SubscriptBox[\"#\", \"2\"]\))]
={{\!\(\*SubscriptBox[\"#\", \"1\"]\),{\!\(\*SubscriptBox[\"index\", \"1\"]\)},{c \!\(\*SubscriptBox[\"#\", \"1\"]\)},{q
    \!\(\*SubscriptBox[\"#\", \"1\"]\)}},{\!\(\*SubscriptBox[\"#\", \"2\"]\),{\!\(\*SubscriptBox[\"index\",
    \"2\"]\)},{c \!\(\*SubscriptBox[\"#\", \"2\"]\)},{q \!\(\*SubscriptBox[\"#\", \"2\"]\)}}}
For a more in depth explanation see the DiracQ writeup notebook.";

FullOrganize::usage = "FullOrganize is a function that will organize and
    simplify and expression. This function is identical to SimplifyQ except that output is left in the organized format.";

Humanize::usage =
  "Humanize is the functional opposite of Organize. Humanize takes a nested list of terms organized according to the method of the package and yields output
    of familiar mathematical forms. Humanize only reconizes input that is the output of the Organize function. Example:
Humanize[{{#,{index},{c #},{q #}}}]
=(#) \!\(\*UnderscriptBox[\"∑\", \"index\"]\)(c #)*(q #)";

SimplifyQ::usage = "SimplifyQ is analogous to the existing Mathematica Simplify function for expressions that contain noncommutative objects.";

StandardOrderQ::usage =
  "StandardOrderQ will order the operators of an expression according to operator type, operator species, and site index respectively. Furthermore
    this function will place creation operators to the left of annihiliation operators of the same type, accounting
    for the commutator of the two operators.Operator product definitions are applied by default, and can be turned off
    by specifying ApplyDefinition->False.Operators are sorted in the following order: {Bra,b†,b,f†,f,J,X,σ,p,q,Ket}.";

TakeQPart::usage =
  "TakeQPart will scan input and remove only q number terms (operators). Output is returned as a list. Each entry in the list is the operators of a single
    term found within the input expression where terms are taken as components seperated by addition.";

TakeCPart::usage =
  "TakeCPart will scan input and remove only c number terms (numbers and other constants). Output is returned as a list. Each entry in the list is the
    operators of a single term found within the input expression where terms are taken as components seperated by addition.";

TakeSummand::usage =
  "TakeSummand will yield only the summand of an input expression of the form Sum[Summand,Indice(s)]. Input of other forms will yield error messages.";
```

```
QCoefficient::usage =
  "QCoeffficient[expression,form] will scan the expression for terms containing a string of operators that match 'form'. The function will output the
    coefficient of the operator(s) specified by 'form'. Only exact matches of the string of operators are found. If several terms are
    found in expression containing terms that match 'form' the output will be a sum of the coefficients of the specified operators.";

Operators::usage = "Operators is the list of symbols that are currently being recognized as operators. The population
    of the list should correspond with the operators selected using the DiracQ palette as well as any user defined operators.";

Decomposition::usage = "Decomposition specifies whether composite commutators will be decomposed into basic commutators
    of anticommutators. Not all combinations of operators can be decomposed entirely into basic commutators or anticommutators only.";

ApplyDefinition::usage = "ApplyDefinition is an option of several functions that specifies whether or not!@$#@%$@#";

SecondaryOperators::usage =
  "SecondaryOperators is a list of all operators that are created by some combination of basic operators. The default list is {\!\(\*SubscriptBox[\"n\",
    \"f\"]\),\!\(\*SubscriptBox[\"n\", \"b\"]\),\!\(\*SuperscriptBox[\"σ\", \"Plus\"]\),\!\(\*SuperscriptBox[\"σ\", \"Minus\"]\),\!\(\*SuperscriptBox[\"J\",
    \"Plus\"]\),\!\(\*SuperscriptBox[\"J\", \"Minus\"]\),\!\(\*OverscriptBox[\"p\", \"⇀\"]\),\!\(\*OverscriptBox[\"q\", \"⇀\"]\)}.";

f::usage =
  "f is the fermionic annihilation operator. This operator requires one index denoting site, and a second optional index can be used to denote spin. Also included
    is the fermionic number operator, represented by \!\(\*SubscriptBox[\"n\", \"f\"]\). The argument scheme for
    the number operator is identical to that of the annihilation operator.";

f†::usage =
  "f† is the fermionic creation operator. This operator requires one index denoting site, and a second optional index can be used to denote spin. Also included
    is the fermionic number operator, represented by \!\(\*SubscriptBox[\"n\", \"f\"]\). The argument scheme
    for the number operator is identical to that of the creation operator.";

b::usage =
  "b is the bosonic annihilation operator. This operator requires one index denoting site, and a second optional index can be used to denote spin. Also included
    is the bosonic number operator, represented by \!\(\*SubscriptBox[\"n\", \"b\"]\). The argument scheme for
    the number operator is identical to that of the annihilation operator.";

b†::usage =
  "b† is the bosonic creation operator. This operator requires one index denoting site, and a second optional index can be used to denote spin. Also included
    is the bosonic number operator, represented by \!\(\*SubscriptBox[\"n\", \"b\"]\). The argument scheme
    for the number operator is identical to that of the creation operator.";

n::usage = "n is the number operator. To specify the number operator
    for bosons use \!\(\*SubscriptBox[\"n\", \"b\"]\) and for fermions use \!\(\*SubscriptBox[\"n\", \"f\"]\).";

X::usage = "X is the Hubbard Operator. Three arguments are required. The first argument represents site. The second
    argument is taken to be the  direction of the 'Ket' spin and the third argument is taken to be the direction of the 'Bra spin";

σ::usage =
  "σ is the Pauli spin matrix. This operator requires two arguments. The first is site index and the second is coordinate direction. An optional third argument
    is used to denote different spin species. Also included are the Pauli raising and lowering operators, denoted by
    \!\(\*SuperscriptBox[\"σ\", \"Plus\"]\) and \!\(\*SuperscriptBox[\"σ\", \"Minus\"]\) respectively. The raising and lowering
    operators require only one argument corresponding to site. A second argument (optional) will be taken to represent spin species.";

p::usage = "p is the canonical momentum operator. This operator can be called with one argument,
    taken to be site index, or two arguments. The second argument will be taken to be coordinate direction. Also included
    is the 3 dimensional canonical momentum vector, represented by OverVector[p], or \!\(\*OverscriptBox[\"p\", \"⇀\"]\).";

q::usage = "q is the canonical position operator. This operator can be called with one argument,
    taken to be site index, or two arguments. The second argument will be taken to be coordinate direction. Also included
    is the 3 dimensional canonical position vector, represented by OverVector[q], or \!\(\*OverscriptBox[\"q\", \"⇀\"]\).";
```

```
J::usage =
  "J is the canonical angular momentum operator. This operator requires two arguments. The first is site index and the second is coordinate direction. An optional
    third argument is used to denote different  species. Also included are the angular momentum raising and lowering operators, denoted
    by \!\(\*SuperscriptBox[\"J\", \"Plus\"]\) and \!\(\*SuperscriptBox[\"J\", \"Minus\"]\) respectively. The raising and lowering
    operators accept only one argument corresponding to site. A second (optional) argument will be taken to represent species.";

Bra::usage = "Bra[x] represents a bra vector x using Dirac notation";

Ket::usage = "Ket[x] represents a ket vector x using Dirac notation";

Vacuum::usage = "Vacuum is the symbol used to represent the vacuum state. In general different operators are taken
    to act on different basis and therefore Vacuum represents the direct product of the vacuum state of several different basis";

x::usage = "x is the x coordinate direction";

y::usage = "y is the y coordinate direction";

z::usage = "z is the z coordinate direction";

ℏ::usage = "ℏ is the reduced Planck's constant";

δ::usage = "δ is the Kronecker. For symbols that do not have numerical definitions this delta will not evaluate unless the option Evaluation is set to Identical";

ϵ::usage =
  "ϵ is the Levi-Civita symbol. In this package the ϵ is only used for coordinate directions x,y,z. Any permutation of these symbols that follows from the right
    hand rule will yield one, any permutation opposite to the right hand rule yields -1, and any argument that involves repeated symbols will yield zero.";

OrganizeQ::usage =
  "OrganizeQ is a function used within the package that is not relevant to most users. OrganizeQ takes organized input and rearranges the operators according
    to a standardized order. OrganizeQ is a subfunction of the FullOrganize function. Output is also organized.";

OrganizedProduct::usage = "OrganizedProduct is a function used within the package that is not relevant to most users.
    OranizedProduct takes organized input and simplifies the operators by evaluating products if possible. Output is also organized.";

†::usage = " The dagger symbol, †, is used in the representation of creation operators. The dagger symbol is not used
    as a superscript but is rather placed directly following the symbol 'f' for fermionic operators and 'b' for bosonic operators.";

AllSymbols::usage = "AllSymbols is a option setting which specifies that every non-numerical symbol will be viewed as a noncommutative object.";

CommutatorRule::usage = "CommutatorRule is an option setting which speficies to decompose composite commutators into basic commutators.";

Evaluation::usage =
  "Evaluation is an option for the Kronecker-δ function. If Evaluation is set to Identical the δ will evaluate to zero unless both arguments are indentical.";

Identical::usage = "Identical is an option setting for the Kronecker-δ
    option Evaluation.If Evaluation is set to Identical the δ will evaluate to zero unless both arguments are indentical.";

NCcross::usage = "Non Commutative cross product of two 3dimensional vectors retaining the order of the operators.";

PositionQ::usage = "PositionQ[expr, pattern] gives a list of the positions of an operator
    matching pattern appear in expr. The position given is the position of the operator relative to other operators in expr only.";

PushOperatorRight::usage = "PushOperatorRight[expr,pattern] will move the
    operator matching pattern to the right of all other operators in every term in expr. Commutators are accounted for.";
```

```
PushOperatorLeft::usage = "PushOperatorRight[expr,pattern] will move the
    operator matching pattern to the left of all other operators in every term in expr. Commutators are accounted for.";

DropQ::usage =
  "DropQ[expr,n] gives expr with the operators specified by n dropped. The specification of operators is identical to that used by the Drop function to specify
    which elements of a list to drop. The operators are specified by the order in which they appear in expr.";

anticommutator::usage = "anticommutator is the function used to specify an unknown or otherwise unevaluated anticommutator of only two elements.";

commutator::usage = "commutator is the function used to specify an unknown or otherwise unevaluated commutator of only two elements.";

function::usage =
  "function is a label used by the Organize function to denote a function of operators that can not be decomposed simply. The first argument is the function,
    and the second argument is the operators on which the function depends. This function is not intended to be manipulated by the user.";

CommutatorDefinition::usage = "CommutatorDefinition is the function through which the commutators of symbols are defined." ;

AntiCommutatorDefinition::usage = "AntiCommutatorDefinition is the function through which the anticommutators of symbols are defined." ;

OperatorProduct::usage = "OperatorProduct is the function through which the products of symbols are defined." ;

DiracQPalette::usage = "DiracQPalette will open the DiracQ Palette.";

OrganizedExpression::usage =
  "OrganizedExpression is an option of several DiracQ functions which allows a user to input a preorganized expression into a function which normally accepts
    standard form input. This is normally done in the interest of time saving. To use preorganized input include the option setting OrganizedExpression->True.";

StandardReordering::usage = "StandardReording is and option of the Commutator function that dictates whether the result will be placed in Standard Order or not.";
```

■

```
Begin["`Private`"];

(*Making some slight alterations to NCM function to make it more generally useful.*)
Unprotect[NonCommutativeMultiply,Times];
    a_**(b_+c_)=a**b+a**c;
    a_**0=0;
    0**a_=0;
    (a_+b_)**c_=a**c+b**c;1**a_=a;
    1**a_=a;
    a_**1=a;
    (-1)**a_=-a;
    a_**(-1)=-a;
Protect[NonCommutativeMultiply,Times];
```

■ Palette

```
DiracQPalette := CreatePalette[
    Column[{
        OpenerView[
                    {"Function Options",
                    TabView[
                                {Apply Definition -> RadioButtonBar[Dynamic[ApplyDefinition], {True, False}],

        "Decomposition" -> RadioButtonBar[Dynamic[Decomposition], {CommutatorRule -> "Commutator", AntiCommutatorRule -> "AntiCommutator"}]
```

```
                        }
                    ]
            },
            True, Alignment -> Left],
    OpenerView[{"Operator Controls",
            Column[{Framed[Labeled[
                Text["b and b†: Bosonic Annihilation and Creation Operators
                    f and f†: Fermionic Annihilation and Creation Operators
                    J: Canonical Angular Momentum Operators
                    X: Hubbard Operators
                    σ: Pauli Spin Matrices
                    q and p: Canonical Position and Momentum Operators
                    Bra and Ket: Dirac Notation Vectors"],
                    "Included Operators", Top, LabelStyle -> Bold], RoundingRadius -> 2, FrameStyle -> {Gray}],
                    Manipulate[
                        Row[
                            {Labeled[
                                If[
                                    allsymbols == True,
                                        Operators = {AllSymbols},
                                            Operators =
                                                Extract[{b, b†, f, f†, J, X, σ, q, p, Bra, Ket},
                                                    Position[
              {bosebut, bosebut, fermibut, fermibut, Jbut, Xbut, σbut, qandpbut, qandpbut, braketbut, braketbut}, True]
                                                    ]
                                    ],
                                    "Primary Operators", Top, LabelStyle -> Bold],
                                    Labeled[
                                        If[allsymbols == True,
                                            Operators = {AllSymbols},
                                            SecondaryOperators = Extract[

             {Subscript[n, b], Subscript[n, f], J^Plus, J^Minus, σ^Plus, σ^Minus, OverVector[q], OverVector[p]},
                                                        Position[{bosebut, fermibut, Jbut, Jbut, σbut, σbut, qandpbut, qandpbut}, True]
                                                        ]
                                            ],
                                    "Secondary Operators", Top, LabelStyle -> Bold]
                                },
                                    "|"
                            ],
                    {{bosebut, False, "b and b†"}, {True, False}},
                    {{fermibut, False, "f and f†"}, {True, False}},
                    {{Jbut, False, "J"}, {True, False}},
                    {{Xbut, False, "X"}, {True, False}},
                    {{σbut, False, "σ"}, {True, False}},
                    {{qandpbut, False, "q and p"}, {True, False}},
                    {{braketbut, False, "Bra and Ket"}, {True, False}},
                    {{allsymbols, False, "All Symbols"}, {True, False}},
                    {{Clearall, Button["Return Default Palette Settings",
                        ApplyDefinition = True;
                        Decomposition = CommutatorRule;
                        Operators = {};
                        {bosebut, fermibut, Jbut, Xbut, σbut, qandpbut, braketbut, allsymbols} = Table[False, {8}]], ""
                        }
                    },
                    FrameMargins -> 0,
```

```
                              FrameLabel -> {"", "", Style["Active Operators", Bold]},
                              Alignment -> Center,
                              ControlPlacement -> {Bottom, Bottom, Bottom, Bottom, Bottom, Bottom, Bottom}]]]},
                              True
                         ],
             OpenerView[{"Typesetting",
                         Grid[{{PasteButton[σ, Appearance -> Automatic, ImageSize -> {43, 43}],
                              PasteButton[Defer[†], Appearance -> Automatic, ImageSize -> {43, 43}],
                              PasteButton[δ, Appearance -> Automatic, ImageSize -> {43, 43}],
                              PasteButton[ℏ, Appearance -> Automatic, ImageSize -> {43, 43}],
                              PasteButton[I, Appearance -> Automatic, ImageSize -> {43, 43}],
                              PasteButton[E, Appearance -> Automatic, ImageSize -> {43, 43}]},
                              {PasteButton["⊗", Defer[Placeholder[] ⊗ Placeholder[]], Appearance -> Automatic, ImageSize -> {43, 43}],
                                 PasteButton["\!\(\*OverscriptBox[\"□\", \"⇀\"]\)",
                                 OverVector[Placeholder[]],
                                 Appearance -> Automatic,
                                 ImageSize -> {43, 43}],

           PasteButton["\!\(\*SuperscriptBox[\"□\", \"□\"]\)", Placeholder[] ^ Placeholder[], Appearance -> Automatic, ImageSize -> {43, 43}],
                                 PasteButton["\!\(\*SubscriptBox[\"□\", \"□\"]\)", Subscript[Placeholder[], Placeholder[]],
            Appearance -> Automatic, ImageSize -> {43, 43}],
                                 PasteButton["\!\(\*UnderoverscriptBox[\"∑\",RowBox[{\"□\", \"=\", \"□\"}], \"□\"]\)□",

            Defer[Sum[Placeholder[], {Placeholder[], Placeholder[], Placeholder[]}]], Appearance -> Automatic, ImageSize -> {43, 43}
                                 ],
                                 PasteButton["\!\(\*FractionBox[\"□\", \"□\"]\)", Defer[Placeholder[] / Placeholder[]], Appearance -> Automatic, ImageSize -> {43, 43}]
                              }
                           },
                           Spacings -> {0, 0}
                           ]
                         },
                     True]
             }
          ] ,
     WindowTitle -> "DiracQ Palette"
]

DiracQPalette;
```

■ Functions that Combine or Manipulate Expressions

```
CommuteParts[a_, bb_, c_, OptionsPattern[]] := Module[{d, ff, g, h, i, j, r, nn, m},
    If[OptionValue[OrganizedExpression] == True,
        d = a,
        d = Organize[a]
    ];
    ff = 1;
    g = 1;
    If[Length[bb] == 1,
        ff = ff ** d[[1]][[4]][[bb[[1]]]],
        Do[ff = ff ** d[[1]][[4]][[bb[[1]] + nn]],
            {nn, 0, bb[[2]] - bb[[1]]}
        ]
    ];
    If[Length[c] == 1,
        g = g ** d[[1]][[4]][[c[[1]]]],
        Do[g = g ** d[[1]][[4]][[c[[1]] + nn]],
            {nn, 0, c[[2]] - c[[1]]}
        ]
    ];
    h = Organize[Commutator[ff, g, StandardReordering -> False]];
    d[[1]][[4]] = Drop[d[[1]][[4]], {bb[[1]], c[[-1]]}];
    d = Table[d[[1]], {Length[h] + 1}];
    For[nn = 1, nn <= Length[h], nn++,
        d[[nn]][[4]] = Flatten[Insert[d[[nn]][[4]], h[[nn]][[4]], bb[[1]]]];
        d[[nn]][[1]] *= h[[nn]][[1]];
        d[[nn]][[3]] = d[[nn]][[3]] h[[nn]][[3]]
    ];
    d[[nn]][[4]] = Insert[d[[nn]][[4]], g ** ff, bb[[1]]];
    r = Organize[Humanize[d]];
    For[nn = 1, nn <= Length[r], nn++,
        r[[nn]] = sumreduce[r[[nn]]]
    ];
    If[! OptionValue[OrganizedExpression] == True,
        r = Humanize[r]
    ];
    Return[r]
]

Commutator[f_, g_, OptionsPattern[]] := Module[{l, m, r, nn, s, o, u, t, a, bb, numterms, lengtha, lengthbb, expr1, expr2, final, i, end},
    o = 1;
    If[OptionValue[OrganizedExpression],
        a = f;
        bb = g,
        If[Head[f] === List,
            If[Head[g] === List,

    Print["Requesting the commutator of an array A, with another array B gives an array C consisting of the Mathematica output C =
       Outer[Commutator, A, B]. For example if A is an array of length 'a' and B is a scalar the result is an array of length 'a'."];
                final = Outer[Commutator, f, g];
                Goto[end],

    Print["Requesting the commutator of an array A, with another array B gives an array C consisting of the Mathematica output C =
       Outer[Commutator, A, B]. For example if A is an array of length 'a' and B is a scalar the result is an array of length 'a'."];
                final = Outer[Commutator, f, {g}];
```

```
                        Goto[end]
                    ],
                    If[Head[g] === List,
  Print["Requesting the commutator of an array A, with another array B gives an array C consisting of the Mathematica output C =
     Outer[Commutator, A, B]. For example if A is an array of length 'a' and B is a scalar the result is an array of length 'a'."];
                        final = Outer[Commutator, {f}, g];
                        Goto[end]
                    ]
                ];
                a = OrganizeQ[Organize[f]];
                bb = OrganizeQ[Organize[g]];
            ];
            lengtha = Length[a];
            lengthbb = Length[bb];
            Do[expr1[nn] = a[[nn]],
                {nn, lengtha}
            ];
            Do[expr2[nn] = bb[[nn]],
                {nn, lengthbb}
            ];
            For[nn = 1, nn <= lengtha, nn ++,
                For[s = 1, s <= lengthbb, s ++,
                    r[o] = combine[expr1[nn], expr2[s]];
                    If[Length[expr1[nn][[4]]] == 0 || Length[expr2[s][[4]]] == 0,
                        r[o] = {ReplacePart[r[o], 1 -> 0]};
                        Goto[skip]
                    ];
 r[o] = ReplacePart[r[o], 4 -> {commutate[expr1[nn][[4]], expr2[s][[4]]]}]; (*commutate is just a place holder head that is recognized by other functions*)
                    r[o] = commute[{r[o]}]; (*commute is the function which actually evaluates the commutator of the operators*)
                    Label[skip];
                    If[ApplyDefinition,
                        If[OptionValue[StandardReordering],
                            r[o] = StandardOrderQ[r[o], OrganizedExpression -> True]
                        ]
                    ];
                    o ++
                ]
            ];
            If[OptionValue[StandardReordering],
                final = StandardOrderQ[final, OrganizedExpression -> True]
            ];
            If[OptionValue[OrganizedExpression] == True,
                final = Flatten[Table[r[v], {v, o - 1}], 1],
                final = Sum[Humanize[r[v]], {v, o - 1}]
            ];
            Label[end];
            Return[final]
        ]
AntiCommutator[f_, g_, OptionsPattern[]] := Module[{l, m, pp, r, nn, s, o, u, t, a, bb, lengtha, lengthbb, end, expr1, expr2, final, i},
            o = 1;
            If[OptionValue[OrganizedExpression],
                a = f;
                bb = g,
```

```
                                If[Head[f] === List,
                                    If[Head[g] === List,

 Print["Requesting the commutator of an array A, with another array B gives an array C consisting of the Mathematica output C =
    Outer[Commutator, A, B]. For example if A is an array of length 'a' and B is a scalar the result is an array of length 'a'."];
                                        final = Outer[Commutator, f, g];
                                        Goto[end],

 Print["Requesting the commutator of an array A, with another array B gives an array C consisting of the Mathematica output C =
    Outer[Commutator, A, B]. For example if A is an array of length 'a' and B is a scalar the result is an array of length 'a'."];
                                        final = Outer[Commutator, f, {g}];
                                        Goto[end]
                                    ],
                                    If[Head[g] === List,

 Print["Requesting the commutator of an array A, with another array B gives an array C consisting of the Mathematica output C =
    Outer[Commutator, A, B]. For example if A is an array of length 'a' and B is a scalar the result is an array of length 'a'."];
                                        final = Outer[Commutator, {f}, g];
                                        Goto[end]
                                    ]
                                ];
                                a = OrganizeQ[Organize[f]];
                                bb = OrganizeQ[Organize[g]];
                            ];
                            lengtha = Length[a];
                            lengthbb = Length[bb];
                            Do[expr1[nn] = a[[nn]],
                                {nn, lengtha}
                            ];
                            Do[expr2[nn] = bb[[nn]],
                                {nn, lengthbb}
                            ];
                            For[nn = 1, nn <= Length[a], nn ++,
                                For[s = 1, s <= Length[bb], s ++,
                                    r[o] = combine[expr1[nn], expr2[s]];
                                    If[Length[expr1[nn][[4]]] == 0 || Length[expr2[s][[4]]] == 0,
                                        r[o] = {ReplacePart[r[o], 1 -> r[o][[1]] * 2]};
                                        Goto[skip]
                                    ];
                                    r[o] = ReplacePart[r[o], 4 -> {anticommutate[expr1[nn][[4]], expr2[s][[4]]]}];
(*anticommutate is just a place holder head that is recognized by other functions*)
                                    r[o] = anticommute[{r[o]}];  (*anticommute is the function which actually evaluates the commutator of the operators*)
                                    Label[skip];
                                    If[ApplyDefinition,
                                        r[o] = StandardOrderQ[r[o], OrganizedExpression -> True]
                                    ];
                                    o ++
                                ]
                            ];
                            If[OptionValue[StandardReordering],
                                            final = StandardOrderQ[final, OrganizedExpression -> True]
                            ];
                            If[OptionValue[OrganizedExpression] == True,
                                final = Flatten[Table[r[v], {v, o - 1}], 1],
                                final = Sum[Humanize[r[v]], {v, o - 1}]
```

```
                    ];
                    Label[end];
                    Return[final]
                ]

a_⊗b_ := ProductQ[a, b];
ProductQ[A_, B_] := Module[{expr1, expr2, i, j, k, term, final, org1, org2, length1, length2, time, end},
                If[Head[A] === List,
                    If[Head[B] === List,

    Print["Requesting the product of an array A, with another array B gives an array C consisting of the Mathematica output C = Outer[ProductQ,
      A, B]. For example if A is an array of length 'a' and B is a scalar the result is an array of length 'a'."];
                        final = Outer[ProductQ, A, B];
                        Goto[end],

    Print["Requesting the product of an array A, with another array B gives an array C consisting of the Mathematica output C = Outer[ProductQ,
      A, B]. For example if A is an array of length 'a' and B is a scalar the result is an array of length 'a'."];
                        final = Outer[ProductQ, A, {B}];
                        Goto[end]
                    ],
                    If[Head[B] === List,

    Print["Requesting the product of an array A, with another array B gives an array C consisting of the Mathematica output C = Outer[ProductQ,
      A, B]. For example if A is an array of length 'a' and B is a scalar the result is an array of length 'a'."];
                        final = Outer[ProductQ, {A}, B];
                        Goto[end]
                    ]
                ];
                org1 = Organize[A];
                org2 = Organize[B];
                length1 = Length[org1];
                length2 = Length[org2];
                Do[expr1[i] = org1[[i]],
                    {i, length1}
                ];
                Do[expr2[i] = org2[[i]],
                    {i, length2}
                ];
                k = 1;
                For[i = 1, i <= length1, i ++,
                    For[j = 1, j <= length2, j ++,
                        If[ApplyDefinition,
                            term[k] = OrganizedProduct[OrganizeQ[{combine[expr1[i], expr2[j]]}]],
                            term[k] = {combine[expr1[i], expr2[j]]}
                        ];
                        term[k] = StandardOrderQ[term[k], OrganizedExpression -> True];
                        k ++
                    ]
                ];
                final = Sum[Humanize[term[k]], {k, length1 length2}];
                Label[end];
                Return[final]
            ]

NCcross[a_, b_] := {a[[2]] ** b[[3]] - a[[3]] ** b[[2]], a[[3]] ** b[[1]] - a[[1]] ** b[[3]], a[[1]] ** b[[2]] - a[[2]] ** b[[1]]}
```

```
PositionQ[a_, bb_, OptionsPattern[]] := Module[{c, lengthc, list, j},
                                            If[OptionValue[OrganizedExpression],
                                                c = a,
                                                c = Organize[a]
                                            ];
                                            lengthc = Length[c];
                                            list = {};
                                            For[i = 1, i <= lengthc, i ++,
                                                If[MemberQ[c[[i]][[4]], bb],
                                                    position = Position[c[[i]][[4]], bb, 1];
                                                    For[j = 1, j <= Length[position], j ++,
                                                        list = Append[list, {i, position[[j]][[1]]}]
                                                    ]
                                                ]
                                            ];
                                            Return[list]
                                        ]

PushOperatorRight[a_, bb_] := Module[{c, result, position, termnum, opnum, i, j, start, correctterm},
                                c = Organize[a];
                                Label[start];
                                correctterm = 0;
                                position = PositionQ[c, bb, OrganizedExpression -> True];
                                For[i = 1, i <= Length[position], i ++,
                                    termnum = position[[i]][[1]];
                                    opnum = position[[i]][[2]];

    If[opnum != Length[c[[termnum]][[4]]] && ! (Length[Drop[c[[termnum]][[4]], opnum]] == Count[Drop[c[[termnum]][[4]], opnum], c[[termnum]][[4]][[opnum]]]),
                                        result = CommuteParts[{c[[termnum]]}, {opnum}, {opnum + 1, Length[c[[termnum]][[4]]]}, OrganizedExpression -> True];
                                        c = Drop[c, {termnum}];
                                        Do[c = Insert[c, result[[j]], termnum],
                                            {j, Length[result]}
                                        ];
                                        Goto[start],
                                        correctterm ++;
                                        If[correctterm == Length[position],
                                            Return[Humanize[c]]
                                        ]
                                    ]
                                ];
                                Return[Humanize[c]]
                            ]
```

```
PushOperatorLeft[a_, bb_] := Module[{c, result, position, termnum, opnum, i, j, start, correctterm},
                                    c = Organize[a];
                                    Label[start];
                                    correctterm = 0;
                                    position = PositionQ[c, bb, OrganizedExpression -> True];
                                    For[i = 1, i <= Length[position], i ++,
                                        termnum = position[[i]][[1]];
                                        opnum = position[[i]][[2]];
                                        If[opnum != 1 && ! (Length[Drop[c[[termnum]][[4]], {opnum, Length[c[[termnum]][[4]]]}]] ==
         Count[Drop[c[[termnum]][[4]], {opnum, Length[c[[termnum]][[4]]]}], c[[termnum]][[4]][[opnum]]]),
                                            result = CommuteParts[{c[[termnum]]}, {1, opnum - 1}, {opnum}, OrganizedExpression -> True];
                                            c = Drop[c, {termnum}];
                                            Do[c = Insert[c, result[[j]], termnum],
                                                {j, Length[result]}
                                            ];
                                            Goto[start],
                                            correctterm ++;
                                            If[correctterm == Length[position],
                                                Return[Humanize[c]]
                                            ]
                                        ]
                                    ];
                                    Return[Humanize[c]]
                                ]
```

```
TakeQPart[a_, OptionsPattern[]] := Module[{bb, lengthexpr, nn, m, c, opterm, lengthopterm, term},
    error = False;
    If[OptionValue[OrganizedExpression],
        bb = a,
        bb = Organize[a]
    ];
    lengthexpr = Length[bb];
    Do[term[i] = bb[[i]],
        {i, lengthexpr}
    ];
    For[nn = 1, nn <= lengthexpr, nn ++,
        If[Length[term[nn][[2]]] > 0,

Print["Error: TakeQPart cannot operate on terms containing the Sum function. To operate on only the summand of a summed term first use TakeSummand"];
            error = True
        ];
        opterm = term[nn][[4]];
        lengthopterm = Length[opterm];
        For[m = 1, m <= lengthopterm, m ++,
            If[Head[opterm[[m]]] === function,
                opterm[[m]] = opterm[[m]][[1]]
            ]
        ];
        If[Length[opterm] == 0,
            c[nn] = {0},
            c[nn] = NonCommutativeMultiply @@ Join[{1}, opterm]
        ]
    ];
    If[error,
        Return[a],
        Return[Table[c[nn], {nn, lengthexpr}]]
    ]
]
```

```
TakeCPart[a_, OptionsPattern[]] := Module[{bb, lengthexpr, nn, m, c, term, error},
                                   error = False;
                                   If[OptionValue[OrganizedExpression],
                                       bb = a,
                                       bb = Organize[a]
                                   ];
                                   lengthexpr = Length[bb];
                                   Do[term[i] = bb[[i]],
                                       {i, lengthexpr}
                                   ];
                                   For[nn = 1, nn <= lengthexpr, nn ++,
                                       If[Length[term[nn][[2]]] > 0,

    Print["Error: TakeCPart cannot operate on terms containing the Sum function. To operate on only the summand of a summed term first use TakeSummand"];
                                           error = True
                                       ];
                                       c[nn] = Times @@ Join[{term[nn][[1]]}, term[nn][[3]]]
                                   ];
                                   If[error,
                                       Return[a],
                                       Return[Table[c[nn], {nn, lengthexpr}]]
                                   ]
                               ]

QCoefficient[A_, B_, OptionsPattern[]] := Module[{term1, term2, list1},
                                           If[OptionValue[OrganizedExpression],
                                               term1 = A,
                                               term1 = Organize[A]
                                           ];
                                           term2 = Organize[B];
                                           If[Length[term2] > 1,

   Print["Error: Coefficient of more than one term requested. Limit second argument to single terms containing only operators."]
                                           ];
                                           If[term2[[1]][[3]] != 1,

   Print["Error: Coefficient of expression including c numbers requested. Limit second argument to terms containing only operators."]
                                           ];
                                           If[Length[term2[[1]][[2]]] > 0,

   Print["Error: Coefficient of expression including sums requested. Limit second argument to terms containing only operators."]
                                           ];
                                           If[term2[[1]][[1]] != 1,

   Print["Error: Coefficient of expression including numbers requested. Limit second argument to terms containing only operators."]
                                           ];
                                           list1 = Select[term1, #[[4]] === term2[[1]][[4]] &];
                                           list1 = ReplacePart[#, 4 -> {}] & /@ list1;
                                           Do[If[Length[list1[[i]][[2]]] > 0,
                                               Print["Coefficient contains sum. The q numbers in expression are assumed to be contained in the sum."];
                                               ],
                                               {i, Length[list1]}
                                           ];
                                           Return[Humanize[list1]]
                                       ]
```

```
TakeSummand[a_, OptionsPattern[]] := Module[{bb, lengthexpr, nn, m, c, opterm, lengthopterm, term},
                                        If[OptionValue[OrganizedExpression],
                                            bb = a,
                                            bb = Organize[a]
                                        ];
                                        If[Length[bb] > 1,
                                            Print["Error: Input is not in the form Sum[Summand,Indice(s)]."],
                                            If[Length[bb[[1]][[2]]] < 1,
                                                Print["Error: Input is not in the form Sum[Summand,Indice(s)]."],
                                                bb = ReplacePart[bb, {1, 2} -> {}]
                                            ]
                                        ];
                                        Return[Humanize[bb]]
                                    ]

DropQ[a_, bb_] := Module[{c},
                    c = Organize[a];
                    Return[Humanize[ReplacePart[c, {1, 4} -> Drop[c[[1]][[4]], bb]]]]
                ]

coordinateset = {x, y, z};

commuteset = {commutate, anticommutate, anticommutator, commutator};
```

Main Package Functions (includes the set of functions that organizes input and humanizes output)

```
(*Organize works itteratively, identifying the head of expressions and applying the corresponding procedures. operationsymbols
 is a list of all the types of heads that it can understand. Considered adding the ability to do Product, but have not yet.*)

operationsymbols = {Times, Sum, Plus, Power, NonCommutativeMultiply};

Organize[A_] := Module[{bb, c, nn, m, B, F, i, d, lengthB, deferpos},
                    B = Expand[A];
                    deferpos = Position[B, Defer[x_]];
                    While[Length[Position[B, Defer[x_]]] > 0,
                        deferpos = Position[B, Defer[x_]];
                        B = ReplacePart[B, deferpos[[1]] -> Extract[B, deferpos[[1]]][[1]]]
                    ];
                    If[Head[B] === Plus,
                        lengthB = Length[B];
                        Do[F[m] = B[[m]], {m, lengthB}],
                            lengthB = 1;
                            F[1] = B
                    ];
                    For[i = 1, i <= lengthB, i ++,
                        bb[i] = {{{F[i]}}, {{1, {}, {}, {placeholder}}}};
                        For[nn = 1, nn <= Length[bb[i][[1]]], nn ++,
                            For[m = 1, m <= Length[bb[i][[1]][[nn]]], m ++,
                                bb[i] = Join[{bb[i][[1]]}, {bb[i][[2]]}, {nn}, {m}];
                                While[Length[bb[i][[1]][[nn]]] > 0,
                                    If[Head[bb[i][[1]][[nn]][[m]]] === Times || Head[bb[i][[1]][[nn]][[m]]] === NonCommutativeMultiply,
                                        bb[i] = TimesandNCMProcedure[bb[i][[1]], bb[i][[2]], bb[i][[3]], bb[i][[4]]]
                                    ];
                                    If[Length[bb[i][[1]][[nn]]] == 0,
                                        Goto[end];
                                    ];
```

```
                        If[Head[bb[i][[1]][[nn]][[m]]] === Sum,
                            bb[i] = SumProcedure[bb[i][[1]], bb[i][[2]], bb[i][[3]], bb[i][[4]]]
                        ];
                        If[Length[bb[i][[1]][[nn]]] == 0,
                            Goto[end]
                        ];
                        If[Head[bb[i][[1]][[nn]][[m]]] === Plus,
                            bb[i] = PlusProcedure[bb[i][[1]], bb[i][[2]], bb[i][[3]], bb[i][[4]]]
                        ];
                        If[Length[bb[i][[1]][[nn]]] == 0,
                            Goto[end]
                        ];
                        If[Head[bb[i][[1]][[nn]][[m]]] === Power,
                            bb[i] = PowerProcedure[bb[i][[1]], bb[i][[2]], bb[i][[3]], bb[i][[4]]]
                        ];
                        If[Length[bb[i][[1]][[nn]]] == 0,
                            Goto[end]
                        ];
                        If[! MemberQ[operationsymbols, Head[bb[i][[1]][[nn]][[m]]]],
                            bb[i] = TermProcedure[bb[i][[1]], bb[i][[2]], bb[i][[3]], bb[i][[4]]]
                        ];
                        Label[end]
                    ]
                ]
            ];
            bb[i] = Delete[bb[i], Position[bb[i], placeholder]]
        ];
        d = Flatten[Table[bb[i][[2]], {i, lengthB}], 1];
        Do[d[[nn]][[3]] = {Simplify[Times @@ d[[nn]][[3]]]},
            {nn, Length[d]}
        ];
        Return[d]
    ]

(*several of the functions below use "function" when they find terms of a certain form (eg. cterm^qterm).
Mostly this is used for operators in the exponent but it could have other uses.
Example of format: (a^p[i]) - > function[a^p[i],p[i]]
(if p is an operator)
*)
```

```
TimesandNCMProcedure[A_, C_, n_, p_] := Module[{bb, d, e, g, ff, m, position, counter, nn},
    counter = 0;
    ff = C;
    position = Position[ff[[n]][[4]], placeholder][[p]];
    e = ReplacePart[A, n -> Drop[A[[n]], {p}]];
    bb = Table[A[[n]][[p]][[m]], {m, Length[A[[n]][[p]]]}];
    d = Length[bb];
    For[m = 1, m <= Length[bb], m++,
      If[MemberQ[operationsymbols, Head[bb[[m]]], Infinity],
        e = ReplacePart[e, n -> Insert[e[[n]], bb[[m]], (p + counter)]];
        counter++;
        ff[[n]] = ReplacePart[ff[[n]], 4 -> Insert[ff[[n]][[4]], placeholder, position]];
        position++,
        If[ MemberQ[Operators, Head[bb[[m]]], Infinity] ||
          MemberQ[SecondaryOperators, Head[bb[[m]]], Infinity] ||
          MemberQ[commuteset, Head[bb[[m]]]] ||
          (MemberQ[Operators, AllSymbols] && ! NumberQ[bb[[m]]]) ||
          Head[bb[[m]]] === function,
            ff[[n]] = ReplacePart[ff[[n]], 4 -> Insert[ff[[n]][[4]], bb[[m]], position]];
            position++,
            If[NumberQ[bb[[m]]],
              ff = MapAt[bb[[m]] * # &, ff, {n, 1}],
              ff[[n]] = ReplacePart[ff[[n]], 3 -> Join[ff[[n]][[3]], {bb[[m]]}]]
            ]
        ]
      ]
    ];
    ff[[n]] = ReplacePart[ff[[n]], 4 -> Drop[ff[[n]][[4]], Position[ff[[n]][[4]], placeholder][[p + counter]]]];
    Return[{e, ff, n, p}]
  ]
```

```
TermProcedure[A_, C_, n_, p_] := Module[{e, ff, powerterm, exponent, term},
        e = MapAt[Drop[#, {p}] &, A, {n}];
        ff = C;
        powerterm = False;
        If[Head[A[[n]][[p]]] === power,
            powerterm = True;
            exponent = A[[n]][[p]][[2]];
            term = A[[n]][[p]][[1]],
            term = A[[n]][[p]]
        ];
        If[ MemberQ[Operators, Head[term], Infinity] ||
            MemberQ[SecondaryOperators, Head[term], Infinity] ||
            MemberQ[commuteset, Head[term]] ||
            (MemberQ[Operators, AllSymbols] && ! NumberQ[term]) ||
            Head[A[[n]][[p]]] === function,
                ff[[n]] = ReplacePart[ff[[n]], 4 -> Insert[ff[[n]][[4]], term, Position[ff[[n]][[4]], placeholder][[p]]]],
                If[NumberQ[term],
                    If[powerterm,
                        ff = MapAt[Power[term, exponent] * # &, ff, {n, 1}],
                        ff = MapAt[A[[n]][[p]] * # &, ff, {n, 1}]
                    ],
                    If[powerterm,
                        ff[[n]] = ReplacePart[ff[[n]], 3 -> Join[ff[[n]][[3]], {Power[term, exponent]}]];
                        ff[[n]] = ReplacePart[ff[[n]], 4 -> Drop[ff[[n]][[4]], Position[ff[[n]][[4]], placeholder][[p]]]],
                        ff[[n]] = ReplacePart[ff[[n]], 4 -> Drop[ff[[n]][[4]], Position[ff[[n]][[4]], placeholder][[p]]]];
                        ff[[n]] = ReplacePart[ff[[n]], 3 -> Join[ff[[n]][[3]], {A[[n]][[p]]}]]
                    ]
                ]
        ];
        Return[{e, ff, n, p}]
    ]
```

```
PowerProcedure[A_, C_, n_, p_] := Module[{ff, m, e, term, end, k, org, j, operatorfind, product, functoperators},
    functoperators = {};
    ff = C;
    term = A[[n]][[p]];
    If[Head[term] === Power && Head[term[[2]]] === Plus,
     product = 1;
     Do[product = product ** Power[term[[1]], term[[2]][[k]]],
      {k, Length[term[[2]]]}
     ];
     e = ReplacePart[A, {n, p} -> product];
     Goto[end],
     e = ReplacePart[A, n -> Drop[A[[n]], {p}]];
     If[Head[term[[2]]] === Integer && term[[2]] > 1,
      For[m = 1, m <= term[[2]], m++,
       e = ReplacePart[e, n -> Insert[e[[n]], term[[1]], p]]
      ];
      Do[ff = ReplacePart[ff, {n, 4} -> Insert[ff[[n]][[4]], placeholder, p]],
       {term[[2]] - 1}
      ],
      operatorfind = False;
      For[k = 1, k <= Length[term], k++,
       org[k] = Organize[term[[k]]];
       For[j = 1, j <= Length[org[k]], j++,
        If[Length[org[k][[j]][[4]]] > 0,
         operatorfind = True;
         functoperators = Union[functoperators, org[k][[j]][[4]]]
        ]
       ]
      ];
      If[operatorfind,
       e = ReplacePart[e, n -> Insert[e[[n]], function[term, functoperators], p]],
       e = ReplacePart[e, n -> Insert[e[[n]], power @@ term, p]]
      ]
     ]
    ];
   Label[end];
   Return[{e, ff, n, p}]
  ]

SumProcedure[A_, C_, n_, p_] := Module[{m, ff, e, t},
    ff = C;
    For[m = 2, m <= Length[A[[n]][[p]]], m++,
     ff[[n]] = ReplacePart[ff[[n]], 2 -> Union[ff[[n]][[2]], {A[[n]][[p]][[m]]}]]
    ];
    e = ReplacePart[A, n -> ReplacePart[A[[n]], p -> A[[n]][[p]][[1]]]];
    Return[{e, ff, n, p}]
   ]
```

```
PlusProcedure[A_, C_, n_, p_] := Module[{e, ff, m},
                                   e = A;
                                   For[m = 1, m <= Length[A[[n]][[p]]], m++,
                                       e = Insert[e, ReplacePart[A[[n]], p -> A[[n]][[p]][[m]]], n + m]
                                   ];
                                   e = Drop[e, {n}];
                                   ff = C;
                                   Do[ff = Insert[ff, C[[n]], n],
                                       {Length[A[[n]][[p]]] - 1}
                                   ];
                                   Return[{e, ff, n, p}]
                                ]
```

```
Humanize[a_] :=
  Module[{i, k, nn, m, cnum, o, l, t, r, qnum, w, lengtha},
    lengtha = Length[a];
    r = 0;
    Do[i[nn] = a[[nn]],
      {nn, 1, Length[a]}
    ];
    For[nn = 1, nn <= lengtha, nn ++,
      If[Length[i[nn][[3]]] != 0,
        cnum = Times @@ i[nn][[3]],
        cnum = 1
      ];
      If[Length[i[nn][[4]]] != 0,
        Do[If[Head[i[nn][[4]][[w]]] === function,
          i[nn] = ReplacePart[i[nn], {4, w} -> i[nn][[4]][[w]][[1]]
            ]
          ],
          {w, Length[i[nn][[4]]]}
        ];
        If[Length[i[nn][[4]]] == 1,
          qnum = i[nn][[4]][[1]],
          qnum = NonCommutativeMultiply @@ i[nn][[4]]
        ],
        qnum = 1
      ];
      k[nn] = cnum ** qnum;
      If[Length[i[nn][[2]]] != 0,
        k[nn] = {k[nn]};
        Do[k[nn] = Append[k[nn], i[nn][[2]][[w]]],
          {w, Length[i[nn][[2]]]}
        ];
        k[nn] = Sum @@ k[nn]
      ]
    ];
    For[nn = 1, nn <= lengtha, nn ++,
      r += i[nn][[1]] k[nn]];
      r =
        Replace[r,
          {f†[i__] ** f[i__] -> Defer[Subscript[n, f][i]],
          aa__ ** f†[i__] ** f[i__] -> aa ** Defer[Subscript[n, f][i]],
          f†[i__] ** f[i__] ** bb__ -> Defer[Subscript[n, f][i]] ** bb,
          aa__ ** f†[i__] ** f[i__] ** bb__ -> aa ** Defer[Subscript[n, f][i]] ** bb,
          b†[i__] ** b[i__] -> Defer[Subscript[n, b][i]],
          aa__ ** b†[i__] ** b[i__] -> aa ** Defer[Subscript[n, b][i]],
          b†[i__] ** b[i__] ** bb__ -> Defer[Subscript[n, b][i]] ** bb,
          aa__ ** b†[i__] ** b[i__] ** bb__ -> aa ** Defer[Subscript[n, b][i]] ** bb},
          {0, Infinity}
        ];
  Return[r]
  ]

SimplifyQ[a_, OptionsPattern[]] := If[OptionValue[OrganizedExpression] == True,
                                        OrganizedProduct[OrganizeQ[a]],
                                        Humanize[FullOrganize[a]]
                                      ]
```

```
(*OrganizeQ reorders the Q (operator) component of organized expressions. This function relies on the order specified in QOrderedQ definitions*)
OrganizeQ[A_] := Module[{B, c, D, j, lengthA, lengthDj4, term1, term2, k, i, permutation},
    lengthA = Length[A];
    Do[D[m] = A[[m]], {m, lengthA}]; (*just to speed things up*)
    For[j = 1, j <= lengthA, j ++,
        permutation = 1;
        lengthDj4 = Length[D[j][[4]]]; (*just to speed things up*)
        For[i = 1, i <= (lengthDj4 - 1), i ++,
            For[k = 1, k <= lengthDj4 - i, k ++,
                term1 = D[j][[4]][[k]];
                term2 = D[j][[4]][[k + 1]];
                If[! QOrderedQ[{term1, term2}],
                    If[fermitest[Head[term1]] && fermitest[Head[term2]],
                        permutation *= -1
                    ];
                        D[j] = ReplacePart[D[j], {4, k} -> term2];
                        D[j] = ReplacePart[D[j], {4, k + 1} -> term1]
                ]
            ]
        ];
        D[j] = ReplacePart[D[j], 1 -> D[j][[1]] * permutation]
    ];
    Return[Table[D[j], {j, lengthA}]]
]
```

```
(*OrganizedProduct takes an organized expression and evaluates products between q terms, yielding a simplified organized expression*)
OrganizedProduct[A_] := Module[{i, j, k, B, C, D, oporg, m, lengthA, jump, finalop},
        lengthA = Length[A];
        Do[B[m] = A[[m]],
          {m, lengthA}
        ];
        For[i = 1, i <= lengthA, i ++,
          If[Length[B[i][[4]]] > 1,
            j = 1;
            While[j < (Length[B[i][[4]]]),
              If[Head[B[i][[4]][[j]]] === Head[B[i][[4]][[j + 1]]],
                If[OperatorProduct[B[i][[4]][[j]], B[i][[4]][[j + 1]]] == B[i][[4]][[j]] ** B[i][[4]][[j + 1]],
                  j ++;
                  Goto[jump]
                ];
                oporg = Organize[OperatorProduct[B[i][[4]][[j]], B[i][[4]][[j + 1]]]];
                If[Length[oporg > 1],
                  j ++;
                  Goto[jump]
                ];
                oporg = oporg[[1]];
                finalop = Flatten[Insert[Delete[B[i][[4]], {{j}, {j + 1}}], oporg[[4]], j]];
                B[i] = combine[B[i], oporg];
                B[i] = ReplacePart[B[i], 4 -> finalop];
                j = j + Length[oporg[[4]]];
                Label[jump],
                j ++
              ]
            ]
          ]
        ];
        Return[Table[sumreduce[B[i]], {i, lengthA}]]
      ]

FullOrganize[A_] := Module[{},
        If[ApplyDefinition,
          Return[OrganizedProduct[OrganizeQ[Organize[A]]]],
          Return[OrganizeQ[Organize[A]]]
        ]
      ]

StandardOrderQ[A_, OptionsPattern[]] := Module[{B, term1, term2, i, j, Start, lengthB, lengthFik4, F, k, final, end, commutatorterm, o},
        If[MemberQ[Operators, AllSymbols],
          final = A;
          Goto[end]
        ];
        If[OptionValue[OrganizedExpression] == True,
          If[ApplyDefinition,
            B = OrganizedProduct[OrganizeQ[A]],
            B = OrganizeQ[A]
          ],
          B = FullOrganize[A]
        ];
        lengthB = Length[B];
        Do[F[m] = {B[[m]]},
          {m, lengthB}
```

```
];
For[i = 1, i <= lengthB, i ++,
    Do[F[i] = ReplacePart[F[i], k -> sumreduce[F[i][[k]]]],
        {k, Length[F[i]]}
    ];
    For[k = 1, k <= Length[F[i]], k ++,
        If[ ! (MemberQ[F[i][[k]][[4]], f[a__]] ||
               MemberQ[F[i][[k]][[4]], f†[a__]] ||
               MemberQ[F[i][[k]][[4]], b[a__]] ||
               MemberQ[F[i][[k]][[4]], b†[a__]] ||
               MemberQ[F[i][[k]][[4]], q[a__]] ||
               MemberQ[F[i][[k]][[4]], p[a__]]
              ),
                Goto[End]
            ];
        Label[Start];
        lengthFik4 = Length[F[i][[k]][[4]]];
        For[j = 1, j <= (lengthFik4 - 1), j ++,
            term1 = F[i][[k]][[4]][[j]];
            term2 = F[i][[k]][[4]][[j + 1]];
            If[ (Head[term1] === f && Head[term2] === f†) ||
                (Head[term1] === b && Head[term2] === b†) ||
                (Head[term1] === q && Head[term2] === p),
                    F[i] = Insert[F[i], F[i][[k]], k];
                    F[i] = Delete[F[i], {{k, 4, j}, {k, 4, j + 1}}];
                    F[i] = ReplacePart[F[i], {k, 3} -> Append[F[i][[k]][[3]], δ[term1[[1]], term2[[1]]]]];
                    If[Length[term1] === 2 && Length[term2] === 2,
                        F[i] = ReplacePart[F[i], {k, 3} -> Append[F[i][[k]][[3]], δ[term1[[2]], term2[[2]]]]],
                        If[Length[term1] != Length[term2],

Print["Uneven Argument: Two operators of the same type are used with unequal argument length."]
                        ]
                    ];
                    F[i] = ReplacePart[F[i], {k + 1, 4, j} -> term2];
                    F[i] = ReplacePart[F[i], {k + 1, 4, j + 1} -> term1];
                    If[(Head[term1] === f && Head[term2] === f†),
                        F[i] = ReplacePart[F[i], {k + 1, 1} -> F[i][[k + 1]][[1]] * -1]
                    ];
                    If[Head[term1] === q && Head[term2] === p,
                        F[i] = ReplacePart[F[i], {k, 3} -> Append[F[i][[k]][[3]], I ℏ]]
                    ];
                    Goto[Start]
            ]
        ];
        Label[End]
    ]
];
Do[Do[F[i] = ReplacePart[F[i], k -> sumreduce[F[i][[k]]]],
        {k, Length[F[i]]}
    ];
    F[i] = OrganizeQ[F[i]],
    {i, lengthB}
];
If[OptionValue[OrganizedExpression] == True,
    final = Flatten[Table[F[i], {i, lengthB}], 1],
```

```
                    final = Sum[Humanize[F[i]], {i, lengthB}]
                ];
                Label[end];
                Return[final]
            ]
(*These functions define how operators should be ordered*)
(*Note on Fermionic reordering: fermionic operators are preferentially reordered with Creation operators to the left of annihilation operators.
  Operators are not reordered if indices are undefined variables (eg f[n]**f†[m]). This is because it may later be defined that n=m*)
QOrderedQ[{f†[n__], f[m__]}] := True

QOrderedQ[{f[n__], f†[m__]}] := If[Length[{n}] == 2,
            If[({n}[[2]] == {m}[[2]]) === False,
                False,
                If[{n}[[1]] == {m}[[1]] === False,
                    OrderedQ[{{n}[[1]], {m}[[1]]}],
                        True
                ]
            ],
            If[({n}[[1]] == {m}[[1]]) === False,
                False,
                True
            ]
        ];

QOrderedQ[{f[n__], f[m__]}] := If[Length[{n}] == 2,
            OrderedQ[{{{n}[[2]], {n}[[1]]}, { {m}[[2]], {m}[[1]]}}],
            OrderedQ[{n, m}]
        ];

QOrderedQ[{f†[n__], f†[m__]}] := If[Length[{n}] == 2,
            OrderedQ[{{{n}[[2]], {n}[[1]]}, { {m}[[2]], {m}[[1]]}}],
            OrderedQ[{n, m}]
        ];

QOrderedQ[{b†[n__], b[m__]}] := If[Length[{n}] == 2,
            If[({n}[[2]] == {m}[[2]]) === False,
                OrderedQ[{{n}[[2]], {m}[[2]]}],
                If[{n}[[1]] == {m}[[1]] === False,
                    OrderedQ[{{n}[[1]], {m}[[1]]}],
                    True
                ]
            ],
            If[{n}[[1]] == {m}[[1]] === False,
                OrderedQ[{{n}[[1]], {m}[[1]]}],
                True
            ]
        ];

QOrderedQ[{b[n__], b†[m__]}] := If[Length[{n}] == 2,
            If[({n}[[2]] == {m}[[2]]) === False,
                OrderedQ[{{n}[[2]], {m}[[2]]}],
                If[{n}[[1]] == {m}[[1]] === False,
                    OrderedQ[{{n}[[1]], {m}[[1]]}],
                    True]
                ],
                If[{n}[[1]] == {m}[[1]] === False,
                    OrderedQ[{{n}[[1]], {m}[[1]]}],
                    True
```

```
                                          ]
                                    ];

QOrderedQ[{b[n__], b[m__]}] := If[Length[{n}] == 2,
                                          If[({n}[[2]] == {m}[[2]]) === False,
                                              OrderedQ[{{n}[[2]], {m}[[2]]}],
                                              If[({n}[[1]] == {m}[[1]]) === False,
                                                  OrderedQ[{{n}[[1]], {m}[[1]]}],
                                                  True
                                              ]
                                          ],
                                          OrderedQ[{{n}[[1]], {m}[[1]]}]
                                    ];

QOrderedQ[{b†[n__], b†[m__]}] := If[Length[{n}] == 2,
                                          If[({n}[[2]] == {m}[[2]]) === False,
                                              OrderedQ[{{n}[[2]], {m}[[2]]}],
                                              If[({n}[[1]] == {m}[[1]]) === False,
                                                  OrderedQ[{{n}[[1]], {m}[[1]]}],
                                                  True
                                              ]
                                          ],
                                          OrderedQ[{{n}[[1]], {m}[[1]]}]
                                    ];

QOrderedQ[{σ[n__], σ[m__]}] := If[Length[{n}] == 3,
                                          If[({n}[[3]] == {m}[[3]]) === False,
                                              OrderedQ[{{n}[[3]], {m}[[3]]}],
                                                  If[{n}[[1]] == {m}[[1]] === False,
                                                      OrderedQ[{{n}[[1]], {m}[[1]]}],
                                                      True
                                                  ]
                                          ],
                                          If[{n}[[1]] == {m}[[1]] === False,
                                              OrderedQ[{{n}[[1]], {m}[[1]]}],
                                              True
                                          ]
                                    ];

QOrderedQ[{X[n__], X[m__]}] := If[! (({n}[[1]] == {m}[[1]]) === False),
                                          True,
                                          OrderedQ[{{n}, {m}}]
                                    ];

QOrderedQ[{p[i__], q[j__]}] := If[Length[{i}] === 2,
                                          If[({i}[[1]] == {j}[[1]]) === False && ({i}[[2]] == {j}[[2]]) === False,
                                              OrderedQ[{{i}, {j}}],
                                              True
                                          ],
                                          If[({i}[[1]] == {j}[[1]]) === False,
                                              OrderedQ[{{i}, {j}}],
                                              True
                                          ]
                                    ];

QOrderedQ[{q[i__], p[j__]}] := If[Length[{i}] === 2,
                                          If[({i}[[1]] == {j}[[1]]) === False && ({i}[[2]] == {j}[[2]]) === False,
                                              OrderedQ[{{i}, {j}}],
                                              True
```

```
                                        ],
                                        If[({i}[[1]] == {j}[[1]]) === False,
                                            OrderedQ[{{i}, {j}}],
                                            True
                                        ]
                                    ];

QOrderedQ[{p[i__], p[j__]}] := OrderedQ[{{i}, {j}}];

QOrderedQ[{q[i__], q[j__]}] := OrderedQ[{{i}, {j}}];

(*Bra and Kets are never reordered with any other operators*)

QOrderedQ[{Ket[n__], Bra[m__]}] := True;

QOrderedQ[{Ket[n__], a_[m__]}] := True;

QOrderedQ[{Bra[n__], a_[m__]}] := True;

QOrderedQ[{a_[n__], Bra[m__]}] := True;

QOrderedQ[{a_[n__], Ket[m__]}] := True;

(*The definitions below cover all remaining cases of pairs of operators, including user defined operators.
User defined operators will never be reordered with existing operators or eachother, unless the user adds new QOrderedQ instructions*)

QOrderedQ[{a_[n__], c_[m__]}] := If[(! a === c) &&

    MemberQ[{b, b†, f, f†, J, X, σ, q, p, Bra, Ket, Subscript[n, b], Subscript[n, f], J^Plus, J^Minus, σ^Plus, σ^Minus, OverVector[q], OverVector[p]}, a] &&
                                        MemberQ[{b, b†, f, f†, J, X, σ, q, p, Bra, Ket, Subscript[n, b],
      Subscript[n, f], J^Plus, J^Minus, σ^Plus, σ^Minus, OverVector[q], OverVector[p]}, c] &&
                                        ! (MemberQ[{b, b†}, a] && MemberQ[{b, b†}, c]) &&
                                        ! (MemberQ[{f, f†}, a] && MemberQ[{f, f†}, c]) &&
                                        ! (MemberQ[{Bra, Ket}, a] || MemberQ[{Bra, Ket}, c]),
                                        OrderedQ[{a[n], c[m]}],
                                        True
                                    ];

QOrderedQ[{a_[i__], function[n_, m_]}] := If[Commutator[Plus @@ m, a[i]] === 0, OrderedQ[{a[i], function[n, m]}], True]

QOrderedQ[{function[n_, m_], a_[i__]}] := If[Commutator[Plus @@ m, a[i]] === 0, OrderedQ[{function[n, m], a[i]}], True]

QOrderedQ[{function[a__, b__], p[j_]}] := True;

QOrderedQ[{p[j_], function[a__, b__]}] := True;
```

Smaller Internal Functions

```
(*combine takes two organized units and combines them into one. Sum indices are a union, operators are Joined*)
combine[f_, g_] := Module[{r},
                    r = {Null, Null, Null, Null};
                    r[[1]] = f[[1]] * g[[1]];
                    r[[2]] = Union[f[[2]], g[[2]]];
                    r[[3]] = f[[3]] * g[[3]];
                    r[[4]] = Join[f[[4]], g[[4]]];
                    Return[r]
                  ]

(*This function finds terms with head commutate in an organized expression and evaluates commutators of these terms using CommutatorDefinition's*)
commute[A_] := Module[{a, bb, c, bc, d, g, h, i, j, k, l, m, nn, pp, o, v, yy, test, t, orgterm, vqnum, term, final, lengthA},
                lengthA = Length[A];
                Do[v[i] = {A[[i]]},
                    {i, lengthA}
                ];
                Label[1];
                For[i = 1, i <= lengthA, i ++,
                    For[j = 1, j <= Length[v[i]], j ++,
                        vqnum = v[i][[j]][[4]];
                        For[nn = 1, nn <= Length[vqnum], nn ++,
                            l = 1;
                            test = False;
                            If[Head[vqnum[[nn]]] === commutate,
                                If[(Length[vqnum[[nn]][[1]]] > 2 || Length[vqnum[[nn]][[2]]] > 2) || (Length[vqnum[[nn]][[1]]] == 2 && Length[vqnum[[nn]]] == 2),
                                    If[Length[vqnum[[nn]][[1]]] <= Length[vqnum[[nn]][[2]]],
                                        a = vqnum[[nn]][[1]];
                                        bc = vqnum[[nn]][[2]],
                                        a = vqnum[[nn]][[2]];
                                        bc = vqnum[[nn]][[1]];
                                        l = -1
                                    ];
                                    bb = Table[bc[[pp]], {pp, Round[Length[bc] / 2]}];
                                    c = Table[bc[[pp]], {pp, Round[Length[bc] / 2] + 1, Length[bc]}];

       orgterm = Organize[l (commutate[a, bb] ** NonCommutativeMultiply @@ Join[{1}, c] - NonCommutativeMultiply @@ Join[{1}, bb] ** commutate[c, a])]
                                ];
                                If[Length[vqnum[[nn]][[1]]] == 1 && Length[vqnum[[nn]][[2]]] == 1,
                                    If[ApplyDefinition,
                                        orgterm = OrganizeQ[Organize[CommutatorDefinition[vqnum[[nn]][[1]][[1]], vqnum[[nn]][[2]][[1]]]]],
                                        orgterm = Organize[commutator[vqnum[[nn]][[1]][[1]], vqnum[[nn]][[2]][[1]]]]
                                    ]
                                ];
                                If[Length[vqnum[[nn]][[1]]] == 1 && Length[vqnum[[nn]][[2]]] == 2,
                                    test = True;
                                    a = vqnum[[nn]][[1]][[1]];
                                    bb = vqnum[[nn]][[2]][[1]];
                                    c = vqnum[[nn]][[2]][[2]]
                                ];
                                If[Length[vqnum[[nn]][[2]]] == 1 && Length[vqnum[[nn]][[1]]] == 2,
                                    test = True;
                                    a = vqnum[[nn]][[2]][[1]];
                                    bb = vqnum[[nn]][[1]][[1]];
                                    c = vqnum[[nn]][[1]][[2]];
                                    l = -1
                                ];
```

```
                        If[test,
                            If[Decomposition === AntiCommutatorRule,
                                If[ApplyDefinition,
                                    orgterm = OrganizeQ[Organize[l (AntiCommutatorDefinition[a, bb] ** c - bb ** AntiCommutatorDefinition[c, a])]],
                                    orgterm = Organize[l (anticommutator[a, bb] ** c - bb ** anticommutator[c, a])]
                                ],
                                If[ApplyDefinition,
                                    orgterm = OrganizeQ[Organize[l (CommutatorDefinition[a, bb] ** c - bb ** CommutatorDefinition[c, a])]],
                                    orgterm = Organize[l (commutator[a, bb] ** c - bb ** commutator[c, a])]
                                ]
                            ]
                        ];
                        term[nn] = Table[v[i][[j]], {Length[orgterm]}];
                        For[d = 1, d <= Length[orgterm], d ++,
                            term[nn] = ReplacePart[term[nn], d -> ReplacePart[term[nn][[d]], 1 -> term[nn][[d]][[1]] orgterm[[d]][[1]]]];
                            term[nn] = ReplacePart[term[nn], d -> ReplacePart[term[nn][[d]], 3 -> orgterm[[d]][[3]] term[nn][[d]][[3]]]];

    Do[term[nn] = ReplacePart[term[nn], d -> ReplacePart[term[nn][[d]], 4 -> Insert[term[nn][[d]][[4]], orgterm[[d]][[4]][[m]], nn + m - 1]]],
                                {m, Length[orgterm[[d]][[4]]]}
                            ];
                            term[nn] = ReplacePart[term[nn], d -> ReplacePart[term[nn][[d]], 4 -> Drop[term[nn][[d]][[4]], {nn + Length[orgterm[[d]][[4]]]}]]]
                        ];
                        Do[v[i] = Insert[v[i], term[nn][[d]], j + d - 1],
                            {d, Length[orgterm]}
                        ];
                        v[i] = Drop[v[i], {j + Length[orgterm]}];
                        Goto[1]
                    ]
                ]
            ]
        ];
        final = Flatten[Table[v[i], {i, lengthA}], 1];
        Return[final]
    ]
(*This function finds terms with head anticommutate in an organized expression and evaluates anticommutators of these terms using AntiCommutatorDefinition's*)
anticommute[A_] := Module[{a, bb, c, bc, d, g, h, i, j, k, l, m, nn, pp, o, v, yy, go, t, anticomm, comm, lengthA, vqnum, orgterm, term, final, end},
                lengthA = Length[A];
                Do[v[i] = {A[[i]]},
                    {i, lengthA}
                ];
                Label[1];
                l = 1;
                For[i = 1, i <= lengthA, i ++,
                    For[j = 1, j <= Length[v[i]], j ++,
                        vqnum = v[i][[j]][[4]];
                        For[m = 1, m <= Length[vqnum], m ++,
                            comm = False;
                            anticomm = False;
                            go = False;
                            If[Head[vqnum[[m]]] === anticommutate,

    If[(Length[vqnum[[m]][[1]]] > 2 || Length[vqnum[[m]][[2]]] > 2) || (Length[vqnum[[m]][[1]]] == 2 && Length[vqnum[[m]][[2]]] == 2),
                                go = True;
                                If[Length[vqnum[[m]][[1]]] <= Length[vqnum[[m]][[2]]],
                                    a = vqnum[[m]][[1]];
```

```
                    bc = vqnum[[m]][[2]],
                    a = vqnum[[m]][[2]];
                    bc = vqnum[[m]][[1]]
                ];
                bb = Table[bc[[pp]], {pp, Round[Length[bc] / 2]}];
                c = Table[bc[[pp]], {pp, Round[Length[bc] / 2] + 1, Length[bc]}];

 orgterm = Organize[commutate[a, bb] ** NonCommutativeMultiply @@ Join[{1}, c] + NonCommutativeMultiply @@ Join[{1}, bb] ** anticommutate[a, c]]
            ];
            If[Length[vqnum[[m]][[1]]] == 1 && Length[vqnum[[m]][[2]]] == 1,
                go = True;
                If[ApplyDefinition,
                    orgterm = OrganizeQ[Organize[AntiCommutatorDefinition[vqnum[[m]][[1]][[1]], vqnum[[m]][[2]][[1]]]]],
                    orgterm = Organize[anticommutator[vqnum[[m]][[1]][[1]], vqnum[[m]][[2]][[1]]]]
                ]
            ];
            If[Length[vqnum[[m]][[1]]] == 1 && Length[vqnum[[m]][[2]]] == 2,
                anticomm = True;
                comm = False;
                go = True;
                a = vqnum[[m]][[1]][[1]];
                bb = vqnum[[m]][[2]][[1]];
                c = vqnum[[m]][[2]][[2]]
            ];
            If[Length[vqnum[[m]][[2]]] == 1 && Length[vqnum[[m]][[1]]] == 2,
                anticomm = True;
                comm = False;
                go = True;
                a = vqnum[[m]][[2]][[1]];
                bb = vqnum[[m]][[1]][[1]];
                c = vqnum[[m]][[1]][[2]]
            ]
        ];
        If[Head[vqnum[[m]]] == commutate,

If[(Length[vqnum[[m]][[1]]] > 2 || Length[vqnum[[m]][[2]]] > 2) || (Length[vqnum[[m]][[1]]] == 2 && Length[vqnum[[m]][[2]]] == 2),
                go = True;
                If[Length[vqnum[[m]][[1]]] <= Length[vqnum[[m]][[2]]],
                    a = vqnum[[m]][[1]];
                    bc = vqnum[[m]][[2]],
                    a = vqnum[[m]][[2]];
                    bc = vqnum[[m]][[1]];
                    l = -1
                ];
                bb = Table[bc[[pp]], {pp, Round[Length[bc] / 2]}];
                c = Table[bc[[pp]], {pp, Round[Length[bc] / 2] + 1, Length[bc]}];

 orgterm = Organize[l (commutate[a, bb] ** NonCommutativeMultiply @@ Join[{1}, c] - NonCommutativeMultiply @@ Join[{1}, bb] ** commutate[c, a])]
            ];
            If[Length[vqnum[[m]][[1]]] == 1 && Length[vqnum[[m]][[2]]] == 1,
                go = True;
                If[ApplyDefinition,
                    orgterm = OrganizeQ[Organize[CommutatorDefinition[vqnum[[m]][[1]][[1]], vqnum[[m]][[2]][[1]]]]],
                    orgterm = Organize[commutator[vqnum[[m]][[1]][[1]], vqnum[[m]][[2]][[1]]]]
                ]
```

```
                ];
                If[Length[vqnum[[m]][[1]]] == 1 && Length[vqnum[[m]][[2]]] == 2,
                    comm = True;
                    anticomm = False;
                    go = True;
                    a = vqnum[[m]][[1]][[1]];
                    bb = vqnum[[m]][[2]][[1]];
                    c = vqnum[[m]][[2]][[2]]
                ];
                If[Length[vqnum[[m]][[2]]] == 1 && Length[vqnum[[m]][[1]]] == 2,
                    comm = True;
                    anticomm = False;
                    go = True;
                    a = vqnum[[m]][[2]][[1]];
                    bb = vqnum[[m]][[1]][[1]];
                    c = vqnum[[m]][[1]][[2]];
                    l = -1
                ]
            ];
            If[anticomm,
                If[ApplyDefinition,
                    orgterm = OrganizeQ[Organize[AntiCommutatorDefinition[a, bb] ** c - bb ** CommutatorDefinition[a, c]]],
                    orgterm = Organize[anticommutator[a, bb] ** c - bb ** commutator[a, c]]
                ]
            ];
            If[comm,
                If[Decomposition === AntiCommutatorRule,
                    If[ApplyDefinition,
                        orgterm = OrganizeQ[Organize[l (AntiCommutatorDefinition[a, bb] ** c - bb ** AntiCommutatorDefinition[c, a])]],
                        orgterm = Organize[l (anticommutator[a, bb] ** c - bb ** anticommutator[c, a])]
                    ],
                    If[ApplyDefinition,
                        orgterm = OrganizeQ[Organize[l (CommutatorDefinition[a, bb] ** c - bb ** CommutatorDefinition[c, a])]],
                        orgterm = Organize[l (commutator[a, bb] ** c - bb ** commutator[c, a])]
                    ]
                ]
            ];
            term[m] = Table[v[i][[j]], {Length[orgterm]}];
            If[go,
                For[d = 1, d <= Length[orgterm], d ++,
                    term[m] = ReplacePart[term[m], d -> ReplacePart[term[m][[d]], 1 -> term[m][[d]][[1]] orgterm[[d]][[1]]]];
                    term[m] = ReplacePart[term[m], d -> ReplacePart[term[m][[d]], 3 -> orgterm[[d]][[3]] term[m][[d]][[3]]]];

Do[term[m] = ReplacePart[term[m], d -> ReplacePart[term[m][[d]], 4 -> Insert[term[m][[d]][[4]], orgterm[[d]][[4]][[nn]], m + nn - 1]]],
                        {nn, Length[orgterm[[d]][[4]]]}
                    ];

term[m] = ReplacePart[term[m], d -> ReplacePart[term[m][[d]], 4 -> Drop[term[m][[d]][[4]], {m + Length[orgterm[[d]][[4]]]}]]]
                ];
                    Do[v[i] = Insert[v[i], term[m][[d]], j + d - 1],
                        {d, Length[orgterm]}
                    ];
                    v[i] = Drop[v[i], {j + Length[orgterm]}];
                    Goto[1]
            ]
        ];
```

```
                    Label[2]
                ]
            ];
            final = Flatten[Table[v[i], {i, lengthA}], 1];
            Return[final]
        ]
```

```
(*sumreduce is the function that looks for summed indices and kronecker deltas and collapses summed indices if they are found withing a delta.*)
sumreduce[a_] := Module[{l, qq, nn, m, ff, pp, r, t, dterm1, dterm2, diff, sol},
        ff = a;
        If[Head[ff[[3]][[1]]] == Times,
            ff[[3]] = Flatten[ReplacePart[ff[[3]], 1 -> Apply[List, ff[[3]][[1]]]]]
        ];
        For[m = 1, m <= Length[ff[[3]]], m ++,
            Label[1];
            If[Head[ff[[3]][[m]]] === δ,
                ff[[3]][[m]] = Sort[ff[[3]][[m]]];
                dterm1 = ff[[3]][[m]][[1]];
                dterm2 = ff[[3]][[m]][[2]];
                diff = dterm1 - dterm2;
                For[nn = 1, nn <= Length[ff[[2]]], nn ++,
                    If[MemberQ[diff, ff[[2]][[nn]], Infinity] || ff[[2]][[nn]] === diff,
                        sol = Solve[diff == 0, ff[[2]][[nn]]];
                        ff = Replace[ff, sol[[1]], Depth[ff]];
                        ff[[2]] = Drop[ff[[2]], {nn}];
                        If[m != Length[ff[[3]]],
                            m ++;
                            Goto[1],
                            Goto[2]
                        ]
                    ]
                ]
            ]
        ];
        Label[2];
        ff[[3]] = {Times @@ ff[[3]]};
        Return[ff]
    ]
```

```
(*CoordinateFind takes as arguments any two of the coordinates "x, y, z" and yields as the output the remaining coordinate*)
CoordinateFind[α_, β_] := Module[{r},
            If[α === β,
                Return[{α}],
                r = Drop[coordinateset, Position[coordinateset, α][[1]]];
                r = Drop[r, Position[r, β][[1]]];
                Return[r[[1]]]
            ]
        ]
```

■ Operator Property Definitions

■ (*General*)

```
CommutatorDefinition[a_[i__], c_[j__]] := If[! (a === c) && ! ((MemberQ[{f, f†}, a] && MemberQ[{f, f†}, c]) || (MemberQ[{b, b†}, a] && MemberQ[{b, b†}, c])),
                                        0
                                    ];
```

```
AntiCommutatorDefinition[a_[i__], c_[j__]] := If[! (a === c) && ! ((MemberQ[{f, f†}, a] && MemberQ[{f, f†}, c]) || (MemberQ[{b, b†}, a] && MemberQ[{b, b†}, c])),
                                                2 a[i] c[j]
                                             ];

OperatorProduct[a_[i__], b_[j__]] := a[i] ** b[j]

CommutatorDefinition[a_[n__], function[m__]] := Print["Error: Unknown commutator called!"];
CommutatorDefinition[function[m__], a_[n__]] := Print["Error: Unknown commutator called!"];

CommutatorDefinition[a_, b_] := If[MemberQ[Operators, AllSymbols, Infinity],
                                   a ** b - b ** a
                                ];

AntiCommutatorDefinition[a_, b_] := If[MemberQ[Operators, AllSymbols, Infinity],
                                       a ** b + b ** a
                                    ];

fermitest1[a_, b_] := If[MemberQ[Operators, AllSymbols, Infinity], False];

OperatorProduct[a_, b_] := If[MemberQ[Operators, AllSymbols, Infinity], a ** b];
```

(*X Operators*)

```
indexparity[a_, b_] := Exp[I π (a^2 + b^2)];

CommutatorDefinition[X[i_, σ1_, σ2_], X[j_, σ3_, σ4_]] :=
  δ[i, j] ** δ[σ3, σ2] ** X[i, σ1, σ4] - δ[i, j] ** δ[σ1, σ4] ** X[i, σ3, σ2] + (1 - δ[i, j]) (1 - indexparity[σ1, σ2]) (1 - indexparity[σ3, σ4]) / 2 X[i, σ1, σ2] ** X[j, σ3, σ4];

AntiCommutatorDefinition[X[i_, σ1_, σ2_], X[j_, σ3_, σ4_]] := δ[i, j] ** δ[σ3, σ2] ** X[i, σ1, σ4] +
   δ[i, j] ** δ[σ1, σ4] ** X[i, σ3, σ2] + (1 - δ[i, j]) (1 - (1 - indexparity[σ1, σ2]) (1 - indexparity[σ3, σ4]) / 4) 2 X[i, σ1, σ2] ** X[j, σ3, σ4];

OperatorProduct[X[i_, σ1_, σ2_], X[j_, σ3_, σ4_]] := δ[i, j] δ[σ2, σ3] X[i, σ1, σ4] - δ[i, j] X[i, σ1, σ2] ** X[j, σ3, σ4] + X[i, σ1, σ2] ** X[j, σ3, σ4];

fermitest[X] := False;
```

(*Bosonic Operators*)

```
CommutatorDefinition[b[i_, σ1_], b†[j_, σ2_]] := δ[σ1, σ2] δ[i, j];
CommutatorDefinition[b†[i_, σ1_], b[j_, σ2_]] := δ[σ1, σ2] δ[i, j];
CommutatorDefinition[b[i_, σ1_], b[j_, σ2_]] := 0;
CommutatorDefinition[b†[i_, σ1_], b†[j_, σ2_]] := 0;

CommutatorDefinition[b[i_], b†[j_]] := δ[i, j];
CommutatorDefinition[b†[i_], b[j_]] := -δ[i, j];
CommutatorDefinition[b[i_], b[j_]] := 0;
CommutatorDefinition[b†[i_], b†[j_]] := 0;

AntiCommutatorDefinition[b[i_, σ1_], b†[j_, σ2_]] := δ[σ1, σ2] δ[i, j] + 2 b†[j, σ2] ** b[i, σ1];
AntiCommutatorDefinition[b†[i_, σ1_], b[j_, σ2_]] := δ[σ1, σ2] δ[i, j] + 2 b†[i, σ1] ** b[j, σ2];
AntiCommutatorDefinition[b[i_, σ1_], b[j_, σ2_]] := 2 b[i, σ1] ** b[j, σ2]
AntiCommutatorDefinition[b†[i_, σ1_], b†[j_, σ2_]] := 2 b†[i, σ1] ** b†[j, σ2];

AntiCommutatorDefinition[b[i_], b†[j_]] := δ[i, j] + 2 b†[j] ** b[i];
AntiCommutatorDefinition[b†[i_], b[j_]] := δ[i, j] + 2 b†[i] ** b[j];
AntiCommutatorDefinition[b[i_], b[j_]] := 2 b[i] ** b[j]
AntiCommutatorDefinition[b†[i_], b†[j_]] := 2 b†[i] ** b†[j];
```

```
OperatorProduct[b[i_], b[i_]] := b[i] ** b[i];

OperatorProduct[b†[i_], b†[i_]] := b†[i] ** b†[i];

fermitest[b] := False; fermitest1[b†, a_] := False;

Subscript[n, b][i_] := b†[i] ** b[i];
Subscript[n, b][i_, σ_] := b†[i, σ] ** b[i, σ];

Unprotect[NonCommutativeMultiply];
Unprotect[Times];
b[a__] ** Ket[Vacuum] := 0;
Bra[Vacuum] ** b†[a__] := 0;
b[a__] Ket[Vacuum] := 0;
Bra[Vacuum] b†[a__] := 0;
Protect[NonCommutativeMultiply];
Protect[Times];
```

(*Fermionic Operators*)

```
AntiCommutatorDefinition[f[i_, σ1_], f†[j_, σ2_]] := δ[i, j] δ[σ1, σ2];
AntiCommutatorDefinition[f†[i_, σ1_], f[j_, σ2_]] := δ[i, j] δ[σ1, σ2];
AntiCommutatorDefinition[f[i_, σ1_], f[j_, σ2_]] := 0;
AntiCommutatorDefinition[f†[i_, σ1_], f†[j_, σ2_]] := 0;

CommutatorDefinition[f[i_, σ1_], f†[j_, σ2_]] := δ[σ1, σ2] δ[i, j] - 2 f†[j, σ2] ** f[i, σ1];
CommutatorDefinition[f†[i_, σ1_], f[j_, σ2_]] := δ[σ1, σ2] δ[i, j] - 2 f[j, σ2] ** f†[i, σ1];
CommutatorDefinition[f[i_, σ1_], f[j_, σ2_]] := -2 f[j, σ2] ** f[i, σ1];
CommutatorDefinition[f†[i_, σ1_], f†[j_, σ2_]] := -2 f†[j, σ2] ** f†[i, σ1];

CommutatorDefinition[f[i_], f†[j_]] := δ[i, j] - 2 f†[j] ** f[i];
CommutatorDefinition[f†[i_], f[j_]] := δ[i, j] - 2 f[j] ** f†[i];
CommutatorDefinition[f[i_], f[j_]] := -2 f[j] ** f[i];
CommutatorDefinition[f†[i_], f†[j_]] := -2 f†[j] ** f†[i];

AntiCommutatorDefinition[f[i_], f†[j_]] := δ[i, j];
AntiCommutatorDefinition[f†[i_], f[j_]] := δ[i, j];
AntiCommutatorDefinition[f[i_], f[j_]] := 0;
AntiCommutatorDefinition[f†[i_], f†[j_]] := 0;

Unprotect[NonCommutativeMultiply];
Unprotect[Times];
f†[a__] ** f†[a__] := 0;
f[a__] ** f[a__] := 0;
f[a__] ** Ket[Vacuum] := 0;
Bra[Vacuum] ** f†[a__] := 0;
f†[a__] f†[a__] := 0;
f[a__] f[a__] := 0;
f[a__] Ket[Vacuum] := 0;
Bra[Vacuum] f†[a__] := 0;
f[m__] ** f†[m__] ** f[m__] := f[m];
f[m__] ** f†[m__] ** f[m__] ** f†[m__] := f[m] ** f†[m];
Protect[NonCommutativeMultiply];
Protect[Times];
```

```
OperatorProduct[f[n__], f[m__]] := If[Length[{m}] == 2,
                                      If[({n}[[2]] == {m}[[2]]) === True,
                                          If[({n}[[1]] == {m}[[1]]) === True,
                                              0,
                                              f[n] ** f[m]
                                          ],
                                          f[n] ** f[m]
                                      ],
                                      If[({n}[[1]] == {m}[[1]]) === True,
                                          0,
                                          f[n] ** f[m]
                                      ]
                                  ];

OperatorProduct[f†[n__], f†[m__]] := If[Length[{m}] == 2,
                                          If[({n}[[2]] == {m}[[2]]) === True,
                                              If[({n}[[1]] == {m}[[1]]) === True,
                                                  0,
                                                  f†[n] ** f†[m]
                                              ],
                                              f†[n] ** f†[m]
                                          ],
                                          If[({n}[[1]] == {m}[[1]]) === True,
                                              0,
                                              f†[n] ** f†[m]
                                          ]
                                      ];

OperatorProduct[f[n__], f†[m__]] := f[n] ** f†[m];

OperatorProduct[f†[n__], f[m__]] := f†[n] ** f[m];

OperatorProduct[f[m__], Subscript[n, f][m__]] := f[m];

fermitest[f] := True;
fermitest[f†] := True;

Subscript[n, f][i_] := f†[i] ** f[i];
Subscript[n, f][i_, σ_] := f†[i, σ] ** f[i, σ];
```

(*Canonical Pairs*)

```
CommutatorDefinition[q[i_, α_], p[j_, β_]] := If[MemberQ[{x, y, z}, α] && MemberQ[{x, y, z}, β],
                                                   δ[i, j] δ[α, β, Evaluation -> Identical] I ℏ,
                                                   δ[i, j] δ[α, β] I ℏ
                                               ]

CommutatorDefinition[p[i_, α_], q[j_, β_]] := If[MemberQ[{x, y, z}, α] && MemberQ[{x, y, z}, β],
                                                   -δ[i, j] δ[α, β, Evaluation -> Identical] I ℏ,
                                                   -δ[i, j] δ[α, β] I ℏ
                                               ]

CommutatorDefinition[q[i_], p[j_]] := δ[i, j] I ℏ

CommutatorDefinition[p[i_], q[j_]] := -δ[i, j] I ℏ

CommutatorDefinition[q[i_], q[j_]] := 0
```

```
CommutatorDefinition[q[i_, x_], q[j_, y_]] := 0

CommutatorDefinition[p[i_, x_], p[j_, y_]] := 0

CommutatorDefinition[p[i_], p[j_]] := 0

AntiCommutatorDefinition[q[i_, α_], p[j_, β_]] := If[MemberQ[{x, y, z}, α] && MemberQ[{x, y, z}, β],
                                                     δ[i, j] δ[α, β, Evaluation -> Identical] I ℏ + 2 p[j, β] q[i, α],
                                                     δ[i, j] δ[α, β] I ℏ + 2 p[j, β] q[i, α]
                                                  ]
AntiCommutatorDefinition[p[i_, α_], q[j_, β_]] := If[MemberQ[{x, y, z}, α] && MemberQ[{x, y, z}, β],
                                                     -δ[i, j] δ[α, β, Evaluation -> Identical] I ℏ + 2 q[j, β] p[i, α],
                                                     -δ[i, j] δ[α, β] I ℏ + 2 q[j, β] p[i, α]
                                                  ]
AntiCommutatorDefinition[q[i_], p[j_]] := δ[i, j] I ℏ + 2 p[j] ** q[i]

AntiCommutatorDefinition[p[i_], q[j_]] := δ[i, j] I ℏ + 2 p[i] ** q[j]

AntiCommutatorDefinition[q[i_, α_], q[j_, β_]] := 2 q[i, α] q[j, β]

AntiCommutatorDefinition[q[i_], q[j_]] := 2 q[i] q[j]

AntiCommutatorDefinition[p[i_, α_], p[j_, β_]] := 2 p[i, α] p[j, β]

AntiCommutatorDefinition[p[i_], p[j_]] := 2 p[i] p[j]

CommutatorDefinition[p[j__], function[a__, b__]] := Module[{position, term, final, i, pdir, functdir, dirdelta},
                                                        final = 0;
                                                        position = Position[Head /@ b, q];
                                                        For[i = 1, i <= Length[position], i ++,
                                                            term[i] = b[[position[[i]][[1]]]];
                                                            If[Length[term[i]] == 2,
                                                                pdir = {j}[[2]];
                                                                functdir[i] = term[i][[2]];
                                                                If[MemberQ[{x, y, z}, pdir] && MemberQ[{x, y, z}, functdir[i]],
                                                                    dirdelta[i] = δ[functdir[i], pdir, Evaluation -> Identical],
                                                                    dirdelta[i] = δ[functdir[i], pdir]
                                                                ];
                                                                final += -I ℏ δ[term[i][[1]], {j}[[1]]] dirdelta[i] D[a, term[i]],
                                                                final += -I ℏ δ[term[i][[1]], {j}[[1]]] D[a, term[i]]
                                                            ]
                                                        ];
                                                        Return[final]
                                                     ];

CommutatorDefinition[function[a__, b__], p[j__]] := Module[{position, term, final, i, pdir, functdir, dirdelta},
                                                        final = 0;
                                                        position = Position[Head /@ b, q];
                                                        For[i = 1, i <= Length[position], i ++,
                                                            term[i] = b[[position[[i]][[1]]]];
                                                            If[Length[term[i]] == 2,
                                                                pdir = {j}[[2]];
                                                                functdir[i] = term[i][[2]];
```

```
                        If[MemberQ[{x, y, z}, pdir] && MemberQ[{x, y, z}, functdir[i]],
                            dirdelta[i] = δ[functdir[i], pdir, Evaluation -> Identical],
                            dirdelta[i] = δ[functdir[i], pdir]
                        ];
                        final += I ℏ δ[term[i][[1]], {j}[[1]]] dirdelta[i] D[a, term[i]],
                        final += I ℏ δ[term[i][[1]], j] D[a, term[i]]
                    ]
                ];
                Return[final]
            ];

CommutatorDefinition[function[a__, b__], q[j__]] := Module[{position, term, final, i, qdir, functdir, dirdelta},
                final = 0;
                position = Position[Head /@ b, p];
                For[i = 1, i <= Length[position], i ++,
                    term[i] = b[[position[[i]][[1]]]];
                    If[Length[term[i]] == 2,
                        qdir = {j}[[2]];
                        functdir[i] = term[i][[2]];
                        If[MemberQ[{x, y, z}, qdir] && MemberQ[{x, y, z}, functdir[i]],
                            dirdelta[i] = δ[functdir[i], qdir, Evaluation -> Identical],
                            dirdelta[i] = δ[functdir[i], qdir]
                        ];
                        final += -I ℏ δ[term[i][[1]], {j}[[1]]] dirdelta[i] D[a, term[i]],
                        final += -I ℏ δ[term[i][[1]], j] D[a, term[i]]
                    ]
                ];
                Return[final]
            ];

CommutatorDefinition[q[j__], function[a__, b__]] := Module[{position, term, final, i, qdir, functdir, dirdelta},
                final = 0;
                position = Position[Head /@ b, p];
                For[i = 1, i <= Length[position], i ++,
                    term[i] = b[[position[[i]][[1]]]];
                    If[Length[term[i]] == 2,
                        qdir = {j}[[2]];
                        functdir[i] = term[i][[2]];
                        If[MemberQ[{x, y, z}, qdir] && MemberQ[{x, y, z}, functdir[i]],
                            dirdelta[i] = δ[functdir[i], qdir, Evaluation -> Identical],
                            dirdelta[i] = δ[functdir[i], qdir]
                        ];
                        final += I ℏ δ[term[i][[1]], {j}[[1]]] dirdelta[i] D[a, term[i]],
                        final += I ℏ δ[term[i][[1]], j] D[a, term[i]]
                    ]
                ];
                Return[final]
            ];

CommutatorDefinition[function[a__, b__], function[c__, d__]] := 0;

OverVector[q][i_] := {q[i, x], q[i, y], q[i, z]}

OverVector[p][i_] := {p[i, x], p[i, y], p[i, z]}
```

(*Pauli Matrices*)

```
CommutatorDefinition[σ[n__], σ[m__]] := Module[{u, i, j},
    If[Length[{m}[[1]]] > 0,
        u = 1;
        For[i = 1, i <= Length[{m}[[1]]], i ++,
            u = u ** δ[{n}[[1]][[i]], {m}[[1]][[i]]]
        ],
        u = δ[{n}[[1]], {m}[[1]]]
    ];
    If[Length[{m}] == 3,
        Return[δ[{n}[[3]], {m}[[3]]] ϵ[{n}[[2]], {m}[[2]], CoordinateFind[{n}[[2]], {m}[[2]]]]
  2 I u (1 - δ[{n}[[2]], {m}[[2]], Evaluation -> Identical]) σ[{n}[[1]], CoordinateFind[{n}[[2]], {m}[[2]]], {n}[[3]]]],
        Return[ϵ[{n}[[2]], {m}[[2]], CoordinateFind[{n}[[2]], {m}[[2]]]] 2 I u
  (1 - δ[{n}[[2]], {m}[[2]], Evaluation -> Identical]) σ[{n}[[1]], CoordinateFind[{n}[[2]], {m}[[2]]]]]
    ]
];

AntiCommutatorDefinition[σ[i_, α_], σ[j_, β_]] := OperatorProduct[σ[i, α], σ[j, β]] + OperatorProduct[σ[j, β], σ[i, α]];

AntiCommutatorDefinition[σ[i_, α_, γ_], σ[j_, β_, ϕ_]] = OperatorProduct[σ[i, α, γ], σ[j, β, ϕ]] + OperatorProduct[σ[j, β, ϕ], σ[i, α, γ]];

OperatorProduct[σ[i_, α_], σ[j_, β_]] := If[i === j,
    δ[α, β, Evaluation -> Identical] + I ϵ[α, β, CoordinateFind[α, β]] σ[i, CoordinateFind[α, β]],
    σ[i, α] ** σ[j, β]
]

OperatorProduct[σ[i_, α_, γ_], σ[j_, β_, ϕ_]] := If[i === j && γ === ϕ,
    δ[α, β, Evaluation -> Identical] + I ϵ[α, β, CoordinateFind[α, β]] σ[i, CoordinateFind[α, β], γ],
    σ[i, α, γ] ** σ[j, β, ϕ]
]

fermitest[σ] := False;

Unprotect[Power];
(σ^Plus)[i_] := σ[i, x] + I σ[i, y];
(σ^Plus)[i_, α_] := σ[i, x, α] + I σ[i, y, α];
(σ^Minus)[i_] := σ[i, x] - I σ[i, y];
(σ^Minus)[i_, α_] := σ[i, x, α] - I σ[i, y, α];
Protect[Power];

Unprotect[NonCommutativeMultiply];
Unprotect[Times];
(σ^Minus)[a__] ** Ket[Vacuum] := 0;
(σ^Minus)[a__] Ket[Vacuum] := 0;
Bra[Vacuum] ** (σ^Plus)[a__] := 0;
Bra[Vacuum] (σ^Plus)[a__] := 0;
Protect[NonCommutativeMultiply];
Protect[Times];
```

(*Spin Operators*)

```
CommutatorDefinition[J[n__], J[m__]] := Module[{u, i, j},
                                        If[Length[{m}[[1]]] > 0,
                                            u = 1;
                                            For[i = 1, i <= Length[{m}[[1]]], i ++,
                                                u = u ** δ[{n}[[1]][[i]], {m}[[1]][[i]]]
                                            ],
                                            u = δ[{n}[[1]], {m}[[1]]]
                                        ];
                                        If[Length[{m}] == 3,
                                            Return[δ[{n}[[3]], {m}[[3]]] ϵ[{n}[[2]], {m}[[2]], CoordinateFind[{n}[[2]], {m}[[2]]]]
       I ℏ u (1 - δ[{n}[[2]], {m}[[2]], Evaluation -> Identical]) J[{n}[[1]], CoordinateFind[{n}[[2]], {m}[[2]]], {n}[[3]]]],
                                            Return[ϵ[{n}[[2]], {m}[[2]], CoordinateFind[{n}[[2]], {m}[[2]]]] I ℏ u
       (1 - δ[{n}[[2]], {m}[[2]], Evaluation -> Identical]) J[{n}[[1]], CoordinateFind[{n}[[2]], {m}[[2]]]]]
                                        ]
                                    ];

AntiCommutatorDefinition[J[i_, α_], J[j_, β_]] := OperatorProduct[J[i, α], J[j, β]] + OperatorProduct[J[j, β], J[i, α]]

AntiCommutatorDefinition[J[i_, α_, γ_], J[j_, β_, ϕ_]] := OperatorProduct[J[i, α, γ], J[j, β, ϕ]] + OperatorProduct[J[j, β, ϕ], J[i, α, γ]]

OperatorProduct[J[i_, α_], J[j_, β_]] := If[i === j,
                                        δ[α, β, Evaluation -> Identical] + I / 2 ℏ ϵ[α, β, CoordinateFind[α, β]] J[i, CoordinateFind[α, β]],
                                        J[i, α] ** J[j, β]
                                    ]

OperatorProduct[J[i_, α_, γ_], J[j_, β_, ϕ_]] := If[i === j && γ === ϕ,
                                            δ[α, β, Evaluation -> Identical] + I / 2 ℏ ϵ[α, β, CoordinateFind[α, β]] J[i, CoordinateFind[α, β], γ],
                                            J[i, α, γ] ** J[j, β, ϕ]
                                        ]

fermitest[J] := False;

Unprotect[Power];
(J^Plus)[i_] := J[i, x] + I J[i, y];
(J^Plus)[i_, α_] := J[i, x, α] + I J[i, y, α];
(J^Minus)[i_] := J[i, x] - I J[i, y];
(J^Minus)[i_, α_] := J[i, x, α] - I J[i, y, α];
Protect[Power];

Unprotect[NonCommutativeMultiply];
Unprotect[Times];
(J^Minus)[a__] ** Ket[Vacuum] := 0;
(J^Minus)[a__] Ket[Vacuum] := 0;
Bra[Vacuum] ** (σ^Plus)[a__] := 0;
Bra[Vacuum] (σ^Plus)[a__] := 0;
Protect[NonCommutativeMultiply];
Protect[Times];
```

■ (*Bra-Ket*)

```
fermitest[Bra] := False;
fermitest[Ket] := False;
```

(*Add/Delete Operators*)

```
AddOperator[a_] := Module[{},
        AppendTo[Operators, a];
        CommutatorDefinition[a[i__], a[j__]] := a[i] ** a[j] - a[j] ** a[i];
        AntiCommutatorDefinition[a[i__], a[j__]] := a[i] ** a[j] + a[j] ** a[i];
        Print["Please enter all neccessary basic commutation and anticommutation relations. For help type ?AddOperator"]
    ]

DeleteOperator[a_] := Module[{},
        operatorlist = Delete[operatorlist, Position[operatorlist, a]];
        Operators = Delete[Operators, Position[Operators, a]]
    ]
```

(*Kronecker Delta definitions*)

```
δ[i_, j_, OptionsPattern[]] := 0 /; (! (i === j)) && (OptionValue[Evaluation] === Identical);
δ[i_, j_, OptionsPattern[]] := 1 /; i === j && OptionValue[Evaluation] === Identical;
δ[i_, j_, OptionsPattern[]] := 1 /; i == j && OptionValue[Evaluation] === default;
δ[i_, j_, OptionsPattern[]] := 0 /; i != j; δ[i_, -i_, OptionsPattern[]] := 0;
δ[-i_, i_, OptionsPattern[]] := 0;
δ[x, y] := 0;
δ[y, z] := 0;
δ[z, x] := 0;
δ[y, x] := 0;
δ[x, z] := 0;
δ[z, y] := 0;
```

(*Levi-Civita symbol definitions*)

```
ϵ[x, y, z] := 1;
ϵ[y, z, x] := 1;
ϵ[z, x, y] := 1;
ϵ[x, z, y] := -1;
ϵ[y, x, z] := -1;
ϵ[z, y, x] := -1;
ϵ[a_, a_, b_] := 0;
ϵ[a_, b_, a_] := 0;
ϵ[a_, b_, b_] := 0;
```

Function Options

```
Options[PositionQ] = {OrganizedExpression -> False};

Options[CommuteParts] = {OrganizedExpression -> False};

Options[StandardOrderQ] = {OrganizedExpression -> False};

Options[SimplifyQ] = {OrganizedExpression -> False};

Options[TakeQPart] = {OrganizedExpression -> False};

Options[TakeCPart] = {OrganizedExpression -> False};
```

```
Options[TakeSummand] = {OrganizedExpression -> False};

Options[QCoefficient] = {OrganizedExpression -> False};

Options[δ] = {Evaluation -> default};

Options[Commutator] = {StandardReordering -> True, OrganizedExpression -> False};

Options[AntiCommutator] = {StandardReordering -> True, OrganizedExpression -> False};
```

■

```
End[];

EndPackage[]
```