$\mathtt{p^\dagger q}$: A tool for prototyping many-body methods for quantum chemistry

Nicholas C. Rubin[1] and A. Eugene DePrince III[2]
[1)]*Google Research, Mountain View, CA, USA*[a)]
[2)]*Department of Chemistry and Biochemistry, Florida State University, Tallahassee, FL 32306-4390*[b)]

$\mathtt{p^\dagger q}$ is a C++ accelerated Python library designed to generate equations for many-body quantum chemistry methods and to realize proof-of-concept implementations of these equations for rapid prototyping. Central to this library is a simple interface to define strings of second-quantized creation and annihilation operators and to bring these strings to normal order with respect to either the true vacuum state or the Fermi vacuum. Tensor contractions over fully-contracted strings can then be evaluated using standard Python functions (*e.g.*, NUMPY's einsum). Given one- and two-electron integrals, these features allow for the rapid implementation and assessment of a wide array of many-body quantum chemistry methods.

I. INTRODUCTION

The development of many-body methods for quantum chemistry applications carries several technical challenges. Deriving working equations by hand can be tedious and error prone, and the realization of even a proof-of-concept implementation that exhibits reasonable performance can be difficult, particularly for recent entrants to the field. Consequently, there is a long history[1] of applying computers for automated generation, manipulation, and implementation of the working equations of quantum chemistry methods. Automation strategies of varying sophistication have been applied in a variety of contexts, such as many-body perturbation theory (MBPT),[2–5] coupled-cluster (CC) theory[6–26] (including response properties[27–29] and analytic gradients[30–32]), multireference (MR) approaches,[29,33–42] and reduced density matrix (RDM) methods.[43] Among the modern examples of automated code generation for quantum chemistry, the tensor contraction engine (TCE)[44], the Symbolic Manipulation Interpreter for Theoretical cHemistry (SMITH3)[45], the General Contraction Code (GeCCo),[46] and the computational science that these tools enable, stand out as prime examples of the utility of this approach to code development.

The primary utility of automation is clear: it eliminates human error. Indeed, documented examples[47] wherein machine-generated algorithms have revealed bugs in mature electronic structure packages highlight the power of such tools. Beyond this obvious benefit, a well-designed interface to a fermion algebra engine also has the potential to accelerate the development and assessment of novel quantum chemistry methods. Consider, for example, the large number of existing many-body approaches that can be realized via selective elimination or modification of diagrams appearing in the CC with single and double excitations (CCSD)[6] equations (*e.g.* quadratic configuration interaction (QCI),[48] approximate second-order CC [CC2],[49] nCC[50] models, parametrized CCSD [pCCSD][51–53], etc.). The development of pCCSD, for example, was motivated by the search for scaling parameters such that a CCSD-like method could mimic the effects of connected triple excitations while remaining exact for two-electron systems. Given a sufficiently flexible code generation platform, more general parametrized CC theories involving larger parameter spaces could be explored in a completely automated fashion. The utility of automated approaches to method development becomes even more apparent when considering modifications to or parametrizations of higher-order CC methods, *e.g.*, CCSD plus triple excitations (CCSDT)[54] models beyond familiar iterative[55] and non-iterative[56] approximations. Our present efforts are partially motivated by the potential utility of a user-friendly framework for just such explorations.

We have developed a C++ accelerated Python library,[57] $\mathtt{p^\dagger q}$, for symbolic fermion algebra that can be used to derive and implement equations for general many-body quantum chemistry methods through a simple and intuitive user interface. At the core of this library is a C++ engine for manipulating strings of fermionic creation and annihilation operators in order to bring them to normal order with respect to either the Fermi vacuum or the true vacuum state. Users can interface with this engine through simple Python calls that support a variety of built-in second-quantized operator types (*e.g.*, cluster amplitudes, the fluctuation potential, etc.) and operations (*e.g.*, commutators, similarity transformations, etc.). Once a list of strings has been defined and brought to normal order, the code collects like terms to deliver compact lists of fully-contracted strings that can either be printed for subsequent use/analysis by the user or translated by $\mathtt{p^\dagger q}$ into tensor contractions that are evaluated using NUMPY's einsum. Given one- and two-electron integrals obtained externally (from OpenFermion[58], for example), these automatically generated contractions can then be easily incorporated by the user into appropriate kernels to realize a functioning quantum chemistry code.

In this work, we provide an overview of the basic capabilities of $\mathtt{p^\dagger q}$ and demonstrate its use as a code generation platform. In Sec. II, we explore a typical use case in which normal-order is defined relative to the Fermi vacuum, providing a walk-through of the generation of spin-orbital-basis equations for determining cluster amplitudes, de-excitation amplitudes, and unrelaxed RDMs at the CCSD level of theory. Additional use cases involving normal-order with respect to the true vacuum are also considered. The procedure by which the equations generated by $\mathtt{p^\dagger q}$ can be converted into executable code via NUMPY einsum contractions is discussed in Sec. III. Some concluding remarks can be found in Sec. IV.

[a)]Electronic mail: nickrubin@google.com
[b)]Electronic mail: adeprince@fsu.edu

II. FERMION ALGEBRA

A. Built-in operator types and operations

p†q supports several built-in operator types, including CC amplitudes, CC de-excitation amplitudes (or left-hand equation of motion [EOM] CC[59–61] amplitudes), right-hand EOM-CC amplitudes, and general one- and two-body operators. A complete list of these operators and their definitions are provided in Table I. Note that we use standard notation for labeling orbital spaces, *i.e.*, i, j, k, l, m, and n refer to spin-orbitals that are occupied in the reference configuration; a, b, c, d, e, and f are unoccupied spin-orbitals; and general spin-orbitals are denoted by labels p, q, r, and s. The library also supports enough basic operator manipulation to compactly define equations for many-body methods such as CCSD (see Table II). Before delving into such a complex case, we first introduce some important features of p†q through a much simpler example.

The following code will evaluate the expectation value of the Hamiltonian with respect to a single reference configuration, which will serve as the vacuum state.

```
import pdaggerq
pq = pdaggerq.pq_helper('fermi')
pq.add_operator_product(1.0,['f'])
pq.add_operator_product(1.0,['v'])
pq.simplify()
terms = pq.fully_contracted_strings()
for my_term in terms:
    print(my_term)
pq.clear()
```

First, the argument passed to `pq_helper` indicates that normal order is defined with respect to the Fermi vacuum. Second, the function `add_operator_product` adds the Fock and fluctuation potential operators the current list of operator strings (see Table I for the definitions of these operators). After each of these calls, the relevant string is automatically brought to normal order. In a similar way, all functions in Table II labeled `add_xxx` automatically bring the argument strings to normal order when they are invoked. The `simplify` function then cleans up the resulting list of normal-ordered strings by removing delta functions (arising from the application of anticommutation relations $\{\hat{a}_p^\dagger, \hat{a}_q\} = \delta_{pq}$) and consolidating or canceling like terms. The user can then request a list of fully contracted strings. Lastly, the `clear` function clears the current list of strings so that the `pq` object can be used to define additional operator products, etc. This simple code produces the output

```
['+1.000000', 'f(i,i)']
['-0.500000', '<i,j||i,j>']
```

where repeated labels imply sums, which, in this case, are carried out over occupied orbitals.

We note here that the string argument that is passed to `add_operator_product` is actually a list of strings that are interpreted as a product of operators by p†q . In this way, equations for essentially any many-body method can be developed via suitable calls to this function. Such a strategy will become cumbersome, though, for theories more complex than low-order perturbation theory, for example. In the following subsection, we introduce additional functions that allow for complex operator manipulations with simple and intuitive calls.

B. Unrelaxed CCSD RDMs

The primary strength of p†q is the ease with which one can generate equations for sophisticated many-body methods. Here, we consider coupled-cluster (CC) theory,[62–65] with a focus on the case of CC with single and double excitations (CCSD).[6] Our goal is to explore enough functionality in p†q to realize a spin-orbital-basis implementation of CCSD unrelaxed one- and two-particle RDMs. Solvers for higher-order approaches (*e.g.* CCSD(T)[56], CC3,[55] CCSDT,[54] and CCSDTQ[66]) can be found on GitHub.[57]

We begin by defining the CCSD Lagrangian[67]

$$L(\boldsymbol{l},\boldsymbol{t}) = \langle\psi_0|(1+\hat{\Lambda})e^{-\hat{T}}\hat{H}e^{\hat{T}}|\psi_0\rangle \tag{1}$$

where the cluster operator, $\hat{T}$, and the de-excitation operator, $\hat{\Lambda}$, are defined as

$$\hat{T} = \sum_{ia} t_i^a \hat{a}_a^\dagger \hat{a}_i + \frac{1}{4}\sum_{ijab} t_{ij}^{ab} \hat{a}_a^\dagger \hat{a}_b^\dagger \hat{a}_j \hat{a}_i \tag{2}$$

and

$$\hat{\Lambda} = \sum_{ia} l_a^i \hat{a}_i^\dagger \hat{a}_a + \frac{1}{4}\sum_{ijab} l_{ab}^{ij} \hat{a}_i^\dagger \hat{a}_j^\dagger \hat{a}_b \hat{a}_a \tag{3}$$

respectively. Here, $\hat{a}^\dagger$ and $\hat{a}$ represent fermionic creation and annihilation operators, respectively, and $|\psi_0\rangle$ represents a reference determinant of orthonormal spin-orbitals. The amplitudes $\boldsymbol{t}$ and $\boldsymbol{l}$ may be be determined by making Eq. 1 stationary with respect to their variations, *i.e.*,

$$\frac{\partial L}{\partial l_e^m} = 0 = \langle\psi_0|\hat{a}_m^\dagger \hat{a}_e e^{-\hat{T}}\hat{H}e^{\hat{T}}|\psi_0\rangle, \tag{4}$$

$$\frac{\partial L}{\partial l_{ef}^{mn}} = 0 = \langle\psi_0|\hat{a}_m^\dagger \hat{a}_n^\dagger \hat{a}_f \hat{a}_e e^{-\hat{T}}\hat{H}e^{\hat{T}}|\psi_0\rangle, \tag{5}$$

$$\frac{\partial L}{\partial t_m^e} = 0 = \langle\psi_0|e^{-\hat{T}}\hat{H}\hat{a}_e^\dagger \hat{a}_m e^{\hat{T}}|\psi_0\rangle + \langle\psi_0|\hat{\Lambda}e^{-\hat{T}}[\hat{H},\hat{a}_e^\dagger \hat{a}_m]e^{\hat{T}}|\psi_0\rangle, \tag{6}$$

and

$$\frac{\partial L}{\partial t_{mn}^{ef}} = 0 = \langle\psi_0|e^{-\hat{T}}\hat{H}\hat{a}_e^\dagger \hat{a}_f^\dagger \hat{a}_n \hat{a}_m e^{\hat{T}}|\psi_0\rangle + \langle\psi_0|\hat{\Lambda}e^{-\hat{T}}[\hat{H},\hat{a}_e^\dagger \hat{a}_f^\dagger \hat{a}_n \hat{a}_m]e^{\hat{T}}|\psi_0\rangle. \tag{7}$$

Equations 4 and 5 are solved for the singles and doubles cluster amplitudes, respectively, while the singles and doubles de-excitation amplitudes can be determined via Eqs. 6 and 7, respectively.

Python scripts for evaluating the singles and doubles cluster and de-excitation amplitudes according to Eqs. 4–7 are provided in Appendices A and B. Here, we highlight the essential

TABLE I. Operator types supported by `p†q` .

operator symbol	operator definition	operator description
`t1`	$t_i^a \hat{a}_a^\dagger \hat{a}_i$	singles cluster amplitudes
`t2`	$\frac{1}{(2!)^2} t_{ij}^{ab} \hat{a}_a^\dagger \hat{a}_b^\dagger \hat{a}_j \hat{a}_i$	doubles cluster amplitudes
`t3`	$\frac{1}{(3!)^2} t_{ijk}^{abc} \hat{a}_a^\dagger \hat{a}_b^\dagger \hat{a}_c^\dagger \hat{a}_k \hat{a}_j \hat{a}_i$	triples cluster amplitudes
`t4`	$\frac{1}{(4!)^2} t_{ijkl}^{abcd} \hat{a}_a^\dagger \hat{a}_b^\dagger \hat{a}_c^\dagger \hat{a}_d^\dagger \hat{a}_l \hat{a}_k \hat{a}_j \hat{a}_i$	quadruples cluster amplitudes
`r1`	$r_i^a \hat{a}_a^\dagger \hat{a}_i$	EOM-CC right-hand singles amplitudes
`r2`	$\frac{1}{(2!)^2} r_{ij}^{ab} \hat{a}_a^\dagger \hat{a}_b^\dagger \hat{a}_j \hat{a}_i$	EOM-CC right-hand doubles amplitudes
`r3`	$\frac{1}{(3!)^2} r_{ijk}^{abc} \hat{a}_a^\dagger \hat{a}_b^\dagger \hat{a}_c^\dagger \hat{a}_k \hat{a}_j \hat{a}_i$	EOM-CC right-hand triples amplitudes
`r4`	$\frac{1}{(4!)^2} r_{ijkl}^{abcd} \hat{a}_a^\dagger \hat{a}_b^\dagger \hat{a}_c^\dagger \hat{a}_d^\dagger \hat{a}_l \hat{a}_k \hat{a}_j \hat{a}_i$	EOM-CC right-hand quadruples amplitudes
`l0`	l_0	EOM-CC left-hand reference amplitude
`l1`	$l_a^i \hat{a}_i^\dagger \hat{a}_a$	EOM-CC left-hand singles amplitudes
`l2`	$\frac{1}{(2!)^2} l_{ab}^{ij} \hat{a}_i^\dagger \hat{a}_j^\dagger \hat{a}_b \hat{a}_a$	EOM-CC left-hand doubles amplitudes
`l3`	$\frac{1}{(3!)^2} l_{abc}^{ijk} \hat{a}_i^\dagger \hat{a}_j^\dagger \hat{a}_k^\dagger \hat{a}_c \hat{a}_b \hat{a}_a$	EOM-CC left-hand triples amplitudes
`l4`	$\frac{1}{(4!)^2} l_{abcd}^{ijkl} \hat{a}_i^\dagger \hat{a}_j^\dagger \hat{a}_k^\dagger \hat{a}_l^\dagger \hat{a}_d \hat{a}_c \hat{a}_b \hat{a}_a$	EOM-CC left-hand quadruples amplitudes
`1`	1	unit operator
`h`	$\sum_{pq} h_{pq} \hat{a}_p^\dagger \hat{a}_q$	general one-body operator
`g`	$\sum_{pqrs} g_{pqrs} \hat{a}_p^\dagger \hat{a}_q^\dagger \hat{a}_s \hat{a}_r$	general two-body operator
`f`	$\sum_{pq} (h_{pq} + \sum_i \langle pi \vert\vert qi \rangle) \hat{a}_p^\dagger \hat{a}_q$	Fock operator
`v`	$\frac{1}{4} \sum_{pqrs} \langle pq \vert\vert rs \rangle \hat{a}_p^\dagger \hat{a}_q^\dagger \hat{a}_s \hat{a}_r - \sum_{pqi} \langle pi \vert\vert qi \rangle \hat{a}_p^\dagger \hat{a}_q$	fluctuation potential operator
`e1(p,q)`	$\hat{a}_p^\dagger \hat{a}_q$	one-body transition operator
`e2(p,q,r,s)`	$\hat{a}_p^\dagger \hat{a}_q^\dagger \hat{a}_r \hat{a}_s$	two-body transition operator
`e3(p,q,r,s,t,u)`	$\hat{a}_p^\dagger \hat{a}_q^\dagger \hat{a}_r^\dagger \hat{a}_s \hat{a}_t \hat{a}_u$	three-body transition operator
`e4(p,q,r,s,t,u,v,w)`	$\hat{a}_p^\dagger \hat{a}_q^\dagger \hat{a}_r^\dagger \hat{a}_s^\dagger \hat{a}_t \hat{a}_u \hat{a}_v \hat{a}_w$	four-body transition operator

details necessary for understanding these examples. Consider first the following code that generates the singles equations defined by Eq. 4

```
# 0 = < 0 | m* e e(-T) H e(T) | 0>
pq.set_left_operators(['e1(m,e)'])
pq.add_st_operator(1.0,['f'],['t1','t2'])
pq.add_st_operator(1.0,['v'],['t1','t2'])
pq.simplify()
t1_terms = pq.fully_contracted_strings()
for my_term in t1_terms:
    print(my_term)
pq.clear()
```

A bra state is defined by the `set_left_operators` function, after which the similarity-transformed Fock and fluctuation potential operators are added via the `add_st_operator` function (see Tables I and II for details). The latter functions evaluate the similarity transformation according to the Baker–Campbell–Hausdorff expansion, including up to four nested commutators. Obviously, many of these commutators will evaluate to zero (*e.g.* quadruple commutators involving `t2`), but the work to evaluate these zeros could be avoided through the use of lower-level commands specifying the non-zero commutators, double commutators, etc. (see Table II). Equations for the doubles cluster amplitudes could be obtained in a similar way; one need only change the bra state via `pq.set_left_operators(['e2(m,n,f,e)'])` (see Appendix A).

Given optimal cluster amplitudes, CCSD de-excitation amplitudes are defined via the solution of Eqs. 6 and 7. For example, the following code will generate the singles de-excitation equations defined by Eq. 6

```
#  <0| e(-T) H e*m e(T)|0>
pq.set_left_operators(['1'])
pq.add_st_operator(1.0,['f','e1(e,m)'],['t1','t2'])
pq.add_st_operator(1.0,['v','e1(e,m)'],['t1','t2'])

# <0| L e(-T) [H,e*m] e(T)|0>
pq.set_left_operators(['l1','l2'])
pq.add_st_operator( 1.0,['f','e1(e,m)'],['t1','t2'])
pq.add_st_operator( 1.0,['v','e1(e,m)'],['t1','t2'])
pq.add_st_operator(-1.0,['e1(e,m)','f'],['t1','t2'])
pq.add_st_operator(-1.0,['e1(e,m)','v'],['t1','t2'])
pq.simplify()
l1_terms = pq.fully_contracted_strings()
for my_term in l1_terms:
    print(my_term)
pq.clear()
```

As compared to the code snippet for the singles cluster amplitudes, additional features to note here are (1) the bra state can be set such that it is defined by a sum of operators (in this case, `l1+l2`), and (2) the similarity transformation function can transform products of operators (*e.g.*, `v` × `e1(e,m)`). As above, equations for the doubles de-excitation amplitudes could be obtained in a similar way; one need only swap all occurrences of `e1(e,m)` for `e2(m,n,f,e)`; a complete script is provided in Appendix B.

Now, given optimal cluster and de-excitation amplitudes, the definition of one- and two-particle RDMs within `p†q` is trivial. The following code will evaluate equations for the occupied-occupied block of the one-particle RDM

TABLE II. p†q function calls and descriptions. The letters a, b, etc. refer to operators tabulated in Table I, and the notations $a+b$ or ab refer to sums or products of operators a and b, respectively. The notation $num\times$ indicates that the operator products/sums are scaled by a numerical factor, num.

function call	description
`pq = pq_helper(vacuum_type) # vacuum_type = 'fermi'/'true'`	initialize operator class with vacuum type
`pq.set_left_operators([a,b,...])`	define bra state as $\langle\psi_0\vert(a+b+...)$
`pq.set_right_operators([a,b,...])`	define ket state as $(a+b+...)\vert\psi_0\rangle$
`pq.add_operator_product(num, [a,b,...])`	$num\times ab...$
`pq.add_commutator(num, [a,b,...],[c,d,...])`	$num\times[ab...,cd...]$
`pq.add_double_commutator(num, [a,b,...],[c,d,...],[e,f,...])`	$num\times[[ab...,cd...],ef...]$
`pq.add_triple_commutator(num, [a,b,...],[c,d,...],[e,f,...],[g,h,...])`	$num\times[[[ab...,cd...],ef...],gh...]$
`pq.add_quadruple_commutator(num, [a,b,...],[c,d,...],[e,f,...],[g,h,...],[i,j...])`	$num\times[[[[ab...,cd...],ef...],gh...],ij...]$
`pq.add_st_operator(num,[a,b,...],[c,d,...])`	$num\times e^{(-c-d-...)}(ab...)e^{(c+d+...)}$
`pq.print()`	print current list of strings
`strings = pq.strings()`	return current list of strings
`pq.simplify()`	simplify the current list of strings by canceling/combining terms
`pq.print_fully_contracted()`	print fully contracted strings
`strings = pq.fully_contracted_strings()`	return list of fully contracted strings
`pq.clear()`	clear current list of strings; reset bra/ket

```
# D(mn) = <0|(1 + l1 + l2) e(-T) m*n e(T) |0>
pq.set_left_operators(['1','l1','l2'])
pq.add_st_operator(1.0,['e1(m,n)'],['t1','t2'])
pq.simplify()
d1_terms = pq.fully_contracted_strings()
for my_term in d1_terms:
    print(my_term)
pq.clear()
```

Additional blocks of the one-particle RDM can be generated through suitable choices of the labels in the `e1` operator, and blocks of the two-particle RDM can be defined via the same code, with `e1(m,n)` swapped in favor of two-body excitation operators, `e2(m,n,f,e)`, etc. Complete scripts fully defining all blocks of the CCSD unrelaxed one- and two-particle RDMs can be found in the Appendices C and D, respectively.

C. Normal order with respect to the true vacuum

To this point, we have considered only normal order with respect to the Fermi vacuum. p†q also supports defining normal order relative to the true vacuum state, which can be useful for theories defined in terms of reduced density matrices (RDMs). One such method is the variational two-electron RDM (2RDM) approach,[68–93] which involves the direct optimization of the elements of the 2RDM, subject to a subset of ensemble N-representability conditions.[94] Many of these conditions involve enforcing correct mappings between RDMs. For example, when enforcing two-particle (PQG) conditions,[68] a positive semidefinite 2RDM ($^2\mathbf{D}$) should map to positive semidefinite two-hole ($^2\mathbf{Q}$) and one-particle-one-hole ($^2\mathbf{G}$) RDMs. The following code generates these mappings

```
pq = pdaggerq.pq_helper('true')

# Q-matrix mapping
pq.set_string(['i', 'j', 'k*', 'l*'])
pq.add_new_string()
pq.simplify()
pq.print()
pq.clear()

# G-matrix mapping
pq.set_string(['i*', 'j', 'k*', 'l'])
pq.add_new_string()
pq.simplify()
pq.print()
pq.clear()
```

Note that we have introduced two new commands here. `set_string` allows the user to set an arbitrary string of creation and annihilation operators that will be brought to normal order upon invokation of `add_new_string`. This code yields

```
// normal-ordered strings:
//     + 1.00000 d(j,k) d(i,l)
//     - 1.00000 l* i d(j,k)
//     - 1.00000 d(i,k) d(j,l)
//     + 1.00000 l* j d(i,k)
//     + 1.00000 k* i d(j,l)
//     - 1.00000 k* j d(i,l)
//     + 1.00000 k* l* i j


// normal-ordered strings:
//     + 1.00000 i* l d(j,k)
//     - 1.00000 i* k* j l
```

where `d(p,q)` represents a Kronecker delta function. Higher order conditions such as the $(2,k)$-conditions[95] where $k \geq 3$ can be non-trivial to generate by hand, but with `p†q` translating these constraints to human readable expressions is easy. For example, the following code generates mappings relevant to the T2 partial three-particle N-representability condition[78,96]

```
pq = pdaggerq.pq_helper('true')

# T2 mapping
ahat = pdaggerq.pq_helper('true')
ahat.set_string(['i*','j*','k','n*','m', 'l'])
ahat.add_new_string()
ahat.set_string(['n*','m','l', 'i*','j*', 'k'])
ahat.add_new_string()
ahat.simplify()
ahat.print()
ahat.clear()
```

and yields

```
// normal-ordered strings:
//     - 1.00000 i* j* l m d(k,n)
//     + 1.00000 n* k d(i,l) d(j,m)
//     - 1.00000 j* n* k m d(i,l)
//     - 1.00000 n* k d(i,m) d(j,l)
//     + 1.00000 j* n* k l d(i,m)
//     + 1.00000 i* n* k m d(j,l)
//     - 1.00000 i* n* k l d(j,m)
```

III. CODE GENERATION

An important part of computer algebra systems is the potential to translate equations into usable code. `p†q` comes with a front-end parser that produces code for residual equations in a NumPy einsum convention. Appropriate summation limits, occupied or virtual indices, are enforced using NumPy array slicing. Using the NumPy einsum format allows a user to implement the generated code with any of a variety of einsum backends. For example, tensor contraction engines in Tensorflow[97], Jax[98], or Dask[99] can be used to implement coupled cluster iterations on a variety of different hardware platforms.

Naive translation of contractions produced by `p†q` leads to correct, though inefficient, code. For example, directly translating the doubles residual of the CCSD amplitude equations yields a tensor contraction that scale as $\mathcal{O}(n^8)$ where n is the number of spin-orbitals. It is well known that the use of intermediate tensors and finding an optimal contraction ordering reduces the doubles residual scaling of CCSD to $\mathcal{O}(n^6)$[100]. The code generated by `p†q` 's parser uses an exhaustive search of tensor contraction orderings provided by the NumPy einsum's `optimize=optimal` flag which was originally devised in the package of opt_einsum[101]. For example, the einsum line for a contraction required to implement the doubles residual is shown below. The code optimally contracts the antisymmeterized two electron integrals g with two t_1 tensors and the t_2 tensor.

```
#   0.5000 <i,j||a,b>*t1(e,i)*t1(f,j)*t2(a,b,m,n)
#   Complete contraction:  ijab,ei,fj,abmn->efmn
#          Naive scaling:  8
#      Optimized scaling:  6
# ---------------------------------------------------
# scaling           current               remaining
# ---------------------------------------------------
#    5        ei,ijab->abej      fj,abmn,abej->efmn
#    5        abej,fj->abef         abmn,abef->efmn
#    6      abef,abmn->efmn              efmn->efmn

doubles_res += 0.5 * einsum('ijab,ei,fj,abmn->efmn',
    g[o, o, v, v], t1, t1, t2, optimize=['einsum_path',
    (0, 1), (0, 2), (0, 1)])
```

The parser also handles the permutation operator notation generated during simplification of contracted strings by introducing intermediate contracted tensors. Examples of this can be seen in the code generation section of Appendix A.

The einsum code construction is in the Python component of `p†q`. Normal ordered strings or tensor contractions are parsed from a string based representation and translated into a Python object representation. Using the function `fully_contracted_stings()` associated with a `pq_helper` object as input to `contracted_strings_to_tensor_terms()`, an internal representation of the contracted string is generated and stored. The stored objects are termed `TensorTerms` and have methods for simply generating the einsum contraction. These routines can be made aware of which labels are not summation indices (the `output_variables` input field) which allows the same routine to be used for energy expressions, amplitude residual equations, etc. The following is an annotated example that produces code corresponding to the singles residual equations for CCSD.

```
import pdaggerq

pq = pdaggerq.pq_helper("fermi")  # initialize vacuum to Fermi vacuum

pq.set_left_operators(['e1(m,e)'])  # single excitation manifold

print('\t < 0 | m* e e(-T) H e(T) | 0> :')

pq.add_st_operator(1.0, ['f'], ['t1', 't2'])  # similarity transform the Fock operator with T1 and T2 generators
pq.add_st_operator(1.0, ['v'], ['t1', 't2'])  # similarity transform the fluctuation operator
pq.simplify()

# grab list of fully-contracted strings, then print
singles_residual_terms = pq.fully_contracted_strings()
singles_residual_terms = contracted_strings_to_tensor_terms(singles_residual_terms) # convert to TensorTerms
for my_term in singles_residual_terms:
    print("#\t", my_term)
    print(my_term.einsum_string(update_val='singles_res', output_variables=('e', 'm'))) # generates einsum code
    print()
pq.clear()  # clear operators
```

which yields

```
    < 0 | m* e e(-T) H e(T) | 0> :

#   1.0000 f(e,m)
singles_res += 1.0 * einsum('em->em', f[v, o])

#  -1.0000 f(i,m)*t1(e,i)
singles_res += -1.0 * einsum('im,ei->em', f[o, o], t1)

#   1.0000 f(e,a)*t1(a,m)
singles_res += 1.0 * einsum('ea,am->em', f[v, v], t1)

#   1.0000 f(i,a)*t2(a,e,i,m)
singles_res += 1.0 * einsum('ia,aeim->em', f[o, v], t2)

#  -1.0000 f(i,a)*t1(a,m)*t1(e,i)
singles_res += -1.0 * einsum('ia,am,ei->em', f[o, v], t1
    , t1, optimize=['einsum_path', (0, 1), (0, 1)])

... elided output
```

This output code and similar code generated for the doubles residual equations can be copied into functions comprising a full CCSD iteration. In Appendix A we provide a full script for producing the energy, singles, and doubles contractions and in Ref. 57 we provide a full implementation of the CCSD cluster amplitude equations, the CCSD de-excitation amplitude equations, and the equations for CCSD unrelaxed one- and two-particle RDMs.

Vacuum normal ordered strings can also be parsed into `TensorTerms` using the function `vacuum_normal_ordered_strings_to_tensor_terms`. This parsing function interprets the collection of normal ordered strings as RDM elements and Kronecker delta functions. The resulting list of `TensorTerm` objects can be further simplified, rearranged, or augmented for any automatic code generation purpose. For example, the matrix elements in extended RPA[102,103] theories can be represented as contractions of the Hamiltonian operator coefficient tensors and RDMs. Consider the extended RPA equations,

$$\sum_{ij} c^n_{ij} E_{ij,kl} = \omega_n \sum_{ij} c^n_{ij} S_{ij,kl}, \tag{8}$$

with the $E_{ij,kl}$ and $S_{ij,kl}$ defined by

$$E_{ij,kl} = \langle \psi | \left[a^\dagger_k a_l \left[H, a^\dagger_j a_i \right] \right] | \psi \rangle \tag{9}$$

$$S_{ij,kl} = \langle \psi | \left[a^\dagger_k a_l, a^\dagger_j a_i \right] | \psi \rangle \tag{10}$$

where $|\psi\rangle$ is the (possibly correlated) ground-state wave function. Note that here, unlike above, all labels refer to general orbitals (*i.e.*, i, j, k, and l are not limited to the occupied space of any reference configuration). These matrix elements are straightforward to generate with $\mathrm{p}^\dagger\mathrm{q}$. For $E_{ij,kl}$, for example, we simply generate an appropriate double commutator (See Tables I and II) and then use the `TensorTerm` einsum infrastructure to generate contraction code over the operator coefficients of the Hamiltonian. In this case we actually consider contractions over the reduced Hamiltonian elements, ${}^2K_{pqrs}$, defined by

$$H = \sum_{pqrs} {}^2K_{pqrs} a^\dagger_p a^\dagger_q a_r a_s \tag{11}$$

$$\begin{aligned} {}^2K_{pqrs} = \frac{1}{4} \Bigg(\frac{1}{N-1} \left(h_{ps}\delta_{qr} - h_{qs}\delta_{pr} - h_{pr}\delta_{qs} \right. \\ \left. + h_{qr}\delta_{ps} \right) + \langle pq || sr \rangle \Bigg) \end{aligned} \tag{12}$$

where N is the number of electrons. The following code

```
import pdaggerq

pq = pdaggerq.pq_helper('true') # normal order with respect to the true vacuum

# [k^l, [H, j^ i]] = - [[H,j^ i],k^l]
pq.add_double_commutator(-1.0,['e2(p,q,r,s)'],['e1(j,i)'],['e1(k,l)'])
pq.simplify()
rpa_tensor_terms = vacuum_normal_ordered_strings_to_tensor_terms(pq.strings())
pq.clear()

k2_idx = [Index('p', 'all'), Index('q', 'all'), Index('r', 'all'), Index('s', 'all')]
for tt in rpa_tensor_terms:
    # add the hamiltonian to contract with
    tt.base_terms += (pdaggerq.BaseTerm(indices=tuple(k2_idx), name='k2'),)
    print("\n# ", tt)
    print(tt.einsum_string(update_val='erpa_val',
                           occupied=['i', 'j', 'k', 'l', 'r', 's', 'p', 'q'],
                           virtual=[],
                           output_variables=['i','j','k','l'],
                           optimize=False))
```

yields

```
#   1.0000 d(j,s)*d(i,k)*d2(p,q,l,r)*k2(p,q,r,s)
erpa_val += 1.0 * einsum('js,ik,pqlr,pqrs->ijkl', kd[:, :], kd[:, :], d2[:, :, :, :], k2[:, :, :, :])

#   1.0000 d(j,s)*d(k,r)*d2(p,q,i,l)*k2(p,q,r,s)
erpa_val += 1.0 * einsum('js,kr,pqil,pqrs->ijkl', kd[:, :], kd[:, :], d2[:, :, :, :], k2[:, :, :, :])

... elided output
```

The code generated above produces a four-tensor which can be reshaped into the matrix $E_{ij,kl}$. The reduced Hamiltonian matrix elements and 2-RDM are stored in OpenFermion order–$\langle 1'2'21\rangle$. Combining this result with the 1-RDM based metric ($S_{ij,kl}$) to define the generalized eigenvalue problem that is the extended RPA equations. Further simplifications can be made to resolve the delta functions but are left here, represented by identity matrices `kd[:, :]`, for clarity. Though further modifications can be applied, such as delta function simplification or taking advantage of antisymmetry of K_2, the resulting code is fully functioning computational kernel ready for deployment in an appropriate solver.

IV. CONCLUSIONS

`p†q` augments the growing number of Python libraries to aid in fermionic many-body simulation and research. By implementing normal ordering and contraction routines along with algebraic simplifications in C++ and providing a high level interface with Python, the library seamlessly integrates with other Python based simulation tools while maintaining a high level of performance. `p†q`'s symbolic fermionic computer algebra system offers a broad set of functionality including bringing arbitrary fermionic algebra elements into normal order with respect to a Fermi or true vacuum, evaluating high order commutators with particular Hamiltonian elements, and similarity transforms generated by cluster operators up to fourth order. To ensure that this framework can aid in a variety of research areas related to fermionic many-body theories we provide functionality to generate tensor contraction code in the NUMPY einsum format that optimizes contraction order whenever possible. To demonstrate some of these features, we provide codes on GitHub[57] that evaluate 1- and 2-RDMs at the CCSD level of theory, including the optimization of the underlying CCSD t- and λ-amplitudes. We also illustrate the ease with which high-order many-body approaches can be realized with working CCSD(T), CC3, CCSDT, and CCSDTQ solvers.

Computer algebra systems have made computational implementation of many-body theories routine and lowered the rate of implementation errors. `p†q` compliments prior works by providing a user friendly C++ accelerated Python variant geared towards rapid prototyping and extensibility. While the first release of the library has focused mainly on code generation for spin-orbital formulations of coupled cluster, more examples and utilities for applying `p†q` outside of this regime (*e.g.*, spin restriction) are planned developments. All code can be found on GitHub[57] and is freely available licensed under Apache 2.0. Bug reports, feature requests, and community contributions are encouraged and can be made through GitHub's issue and pull request features.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. CHE-2100984.

[1] S. Hirata, Theor. Chem. Acc. **116**, 2 (2006).
[2] J. Paldus and H. Wong, Comput. Phys. Commun. **6**, 1 (1973).
[3] H. Wong and J. Paldus, Comput. Phys. Commun. **6**, 9 (1973).
[4] W. D. Laidig, G. Fitzgerald, and R. J. Bartlett, Chem. Phys. Lett. **113**, 151 (1985).
[5] P. Knowles, K. Somasundram, N. Handy, and K. Hirao, Chem. Phys. Lett. **113**, 8 (1985).
[6] G. D. Purvis and R. J. Bartlett, J. Chem. Phys. **76**, 1910 (1982).
[7] J. Čížek, J. Paldus, and F. Vinette, Int. J. Quantum Chem. **38**, 831 (1990).
[8] C. L. Janssen and H. F. Schaefer, Theor. Chim. Acta **79**, 1 (1991).
[9] S. A. Kucharski and R. J. Bartlett, Theor. Chim. Acta **80**, 387 (1991).
[10] P. Piecuch, R. Tobol/a, and J. Paldus, Phys. Rev. A **54**, 1210 (1996).
[11] T. D. Crawford, T. J. Lee, and H. F. Schaefer, J. Chem. Phys. **107**, 7943 (1997).
[12] F. E. Harris, Int. J. Quantum Chem. **75**, 593 (1999).
[13] P. Jankowski and B. Jeziorski, J. Chem. Phys. **111**, 1857 (1999).
[14] M. Nooijen and V. Lotrich, J. Chem. Phys. **113**, 494 (2000).
[15] M. Nooijen and V. Lotrich, J. Chem. Phys. **113**, 4549 (2000).
[16] S. Hirata and R. J. Bartlett, Chem. Phys. Lett. **321**, 216 (2000).
[17] M. Kállay and P. R. Surján, J. Chem. Phys. **113**, 1359 (2000).
[18] J. Olsen, J. Chem. Phys. **113**, 7140 (2000).
[19] M. Kállay and P. R. Surján, J. Chem. Phys. **115**, 2945 (2001).
[20] I. Berente, P. G. Szalay, and J. Gauss, J. Chem. Phys. **117**, 7872 (2002).
[21] A. D. Bochevarov and C. D. Sherrill, J. Chem. Phys. **121**, 3374 (2004).
[22] A. Köhn, J. Chem. Phys. **130**, 131101 (2009).
[23] A. Köhn and D. P. Tew, J. Chem. Phys. **133**, 174117 (2010).
[24] A. Köhn, J. Chem. Phys. **133**, 174118 (2010).
[25] D. Datta and J. Gauss, J. Chem. Theory Comput. **9**, 2639 (2013).
[26] M. Krupička, K. Sivalingam, L. Huntington, A. A. Auer, and F. Neese, J. Comp. Chem. **38**, 1853 (2017).
[27] M. Hanauer and A. Köhn, J. Chem. Phys. **131**, 124118 (2009).
[28] A. Köhn, J. Chem. Phys. **130**, 104104 (2009).
[29] P. K. Samanta and A. Köhn, J. Chem. Phys. **149**, 064101 (2018).
[30] M. Kállay and J. Gauss, J. Chem. Phys. **120**, 6841 (2004).
[31] M. Wladyslawski and M. Nooijen (Academic Press, 2005) pp. 1–101.
[32] D. Datta and J. Gauss, J. Chem. Phys. **141**, 104102 (2014).
[33] X. Li and J. Paldus, J. Chem. Phys. **101**, 8812 (1994).
[34] M. Nooijen and V. Lotrich, J. Mol. Struct. THEOCHEM **547**, 253 (2001).
[35] M. Nooijen, Int. J. Mol. Sci. **3**, 656 (2002).
[36] D. I. Lyakh, V. V. Ivanov, and L. Adamowicz, J. Chem. Phys. **122**, 024108 (2005).
[37] M. Hanauer and A. Köhn, J. Chem. Phys. **134**, 204111 (2011).
[38] M. Hanauer and A. Köhn, J. Chem. Phys. **136**, 204107 (2012).
[39] M. Hanauer and A. Köhn, J. Chem. Phys. **137**, 131103 (2012).
[40] M. K. MacLeod and T. Shiozaki, J. Chem. Phys. **142**, 051103 (2015).
[41] B. Vlaisavljevich and T. Shiozaki, J. Chem. Theory Comput. **12**, 3781 (2016).
[42] D. J. Coughtrie, R. Giereth, D. Kats, H.-J. Werner, and A. Köhn, J. Chem. Theory Comput. **14**, 693 (2018).
[43] M. Nooijen, M. Wladyslawski, and A. Hazra, J. Chem. Phys. **118**, 4832 (2003).
[44] S. Hirata, J. Phys. Chem. A **107**, 9887 (2003).
[45] `SMITH3`, Symbolic Manipulation Interpreter for Theoretical cHemistry, version 3.0. http://www.nubakery.org under the GNU General Public License.
[46] A. Köhn, G. W. Richings, and D. P. Tew, J. Chem. Phys. **129**, 201103 (2008).
[47] T. Yanai, Y. Kurashige, M. Saitow, J. Chalupský, R. Lindh, and P. Åke Malmqvist, Mol. Phys. **115**, 2077 (2017).
[48] J. A. Pople, M. Head-Gordon, and K. Raghavachari, J. Chem. Phys. **87**, 5968 (1987).
[49] O. Christiansen, H. Koch, and P. Jørgensen, Chem. Phys. Lett. **243**, 409 (1995).
[50] R. J. Bartlett and M. Musiał, J. Chem. Phys. **125**, 204105 (2006).
[51] L. M. J. Huntington and M. Nooijen, J. Chem. Phys. **133**, 184109 (2010).
[52] L. M. J. Huntington, A. Hansen, F. Neese, and M. Nooijen, J. Chem. Phys. **136**, 064101 (2012).
[53] V. Rishi, A. Perera, M. Nooijen, and R. J. Bartlett, J. Chem. Phys. **146**, 144104 (2017).
[54] J. Noga and R. J. Bartlett, J. Chem. Phys. **86**, 7041 (1987).

[55] H. Koch, O. Christiansen, P. Jørgensen, A. M. Sanchez de Merás, and T. Helgaker, J. Chem. Phys. **106**, 1808 (1997).
[56] K. Raghavachari, G. W. Trucks, J. A. Pople, and M. Head-Gordon, Chem. Phys. Lett. **157**, 479 (1989).
[57] N. C. Rubin and A. E. D. III, "$p^\dagger q$: A tool for prototyping many-body methods for quantum chemistry: https://github.com/edeprince3/pdaggerq," (2021).
[58] J. R. McClean, N. C. Rubin, K. J. Sung, I. D. Kivlichan, X. Bonet-Monroig, Y. Cao, C. Dai, E. S. Fried, C. Gidney, B. Gimby, P. Gokhale, T. Häner, T. Hardikar, V. Havlíček, O. Higgott, C. Huang, J. Izaac, Z. Jiang, X. Liu, S. McArdle, M. Neeley, T. O'Brien, B. O'Gorman, I. Ozfidan, M. D. Radin, J. Romero, N. P. D. Sawaya, B. Senjean, K. Setia, S. Sim, D. S. Steiger, M. Steudtner, Q. Sun, W. Sun, D. Wang, F. Zhang, and R. Babbush, Quantum Sci. Technol. **5**, 034014 (2020).
[59] J. F. Stanton and R. J. Bartlett, J. Chem. Phys. **98**, 7029 (1993).
[60] R. J. Bartlett, WIRES Comput. Mol. Sci. **2**, 126 (2012).
[61] A. I. Krylov, Annu. Rev. Phys. Chem. **59**, 433 (2008).
[62] J. Čížek, J. Chem. Phys. **45**, 4256 (1966).
[63] J. Čížek and J. Paldus, Int. J. Quantum Chem. **5**, 359 (1971).
[64] I. Shavitt and R. Bartlett, *Many-Body Methods in Chemistry and Physics: MBPT and Coupled-Cluster Theory*, Cambridge Molecular Science (Cambridge University Press, 2009).
[65] R. J. Bartlett and M. Musiał, Rev. Mod. Phys. **79**, 291 (2007).
[66] N. Oliphant and L. Adamowicz, J. Chem. Phys. **95**, 6645 (1991).
[67] H. Koch, H. J. A. Jensen, P. Jørgensen, T. Helgaker, G. E. Scuseria, and H. F. Schaefer, J. Chem. Phys. **92**, 4924 (1990).
[68] C. Garrod and J. K. Percus, J. Math. Phys. **5**, 1756 (1964).
[69] C. Garrod, M. V. Mihailović, and M. Rosina, J. Math. Phys. **16**, 868 (1975).
[70] M. V. Mihailović and M. Rosina, Nucl. Phys. A **237**, 221 (1975).
[71] M. Rosina and C. Garrod, J. Comput. Phys. **18**, 300 (1975).
[72] R. M. Erdahl, C. Garrod, B. Golli, and M. Rosina, J. Math. Phys. **20**, 1366 (1979).
[73] R. M. Erdahl, Rep. Math. Phys. **15**, 147 (1979).
[74] M. Nakata, H. Nakatsuji, M. Ehara, M. Fukuda, K. Nakata, and K. Fujisawa, J. Chem. Phys. **114**, 8282 (2001).
[75] D. A. Mazziotti and R. M. Erdahl, Phys. Rev. A **63**, 042113 (2001).
[76] D. A. Mazziotti, Phys. Rev. A **65**, 062511 (2002).
[77] D. A. Mazziotti, Phys. Rev. A **74**, 032501 (2006).
[78] Z. Zhao, B. J. Braams, M. Fukuda, M. L. Overton, and J. K. Percus, J. Chem. Phys. **120**, 2095 (2004).
[79] M. Fukuda, B. J. Braams, M. Nakata, M. L. Overton, J. K. Percus, M. Yamashita, and Z. Zhao, Math. Program. **109**, 553 (2007).
[80] E. Cancès, G. Stoltz, and M. Lewin, J. Chem. Phys. **125**, 064101 (2006).
[81] B. Verstichel, H. V. Aggelen, D. V. Neck, P. W. Ayers, and P. Bultinck, Phys. Rev. A **80**, 032508 (2009).
[82] J. Fosso-Tande, D. R. Nascimento, and A. E. DePrince III, Mol. Phys. **114**, 423 (2016).
[83] B. Verstichel, H. van Aggelen, D. Van Neck, P. Bultinck, and S. D. Baerdemacker, Comput. Phys. Commun. **182**, 1235 (2011).
[84] W. Poelmans, M. Van Raemdonck, B. Verstichel, S. De Baerdemacker, A. Torre, L. Lain, G. E. Massaccesi, D. R. Alcoba, P. Bultinck, and D. Van Neck, J. Chem. Theory Comput. **11**, 4064 (2015).
[85] D. R. Alcoba, A. Torre, L. Lain, G. E. Massaccesi, O. B. Oña, E. M. Honoré, W. Poelmans, D. Van Neck, P. Bultinck, and S. De Baerdemacker, J. Chem. Phys. **148**, 024105 (2018).
[86] K. Head-Marsden and D. A. Mazziotti, J. Chem. Phys. **147**, 084101 (2017).
[87] H. van Aggelen, P. Bultinck, B. Verstichel, D. Van Neck, and P. W. Ayers, Phys. Chem. Chem. Phys. **11**, 5558 (2009).
[88] B. Verstichel, H. van Aggelen, D. Van Neck, P. W. Ayers, and P. Bultinck, J. Chem. Phys. **132**, 114113 (2010).
[89] H. van Aggelen, B. Verstichel, P. Bultinck, D. Van Neck, P. W. Ayers, and D. L. Cooper, J. Chem. Phys. **134**, 054115 (2011).
[90] R. R. Li and A. E. DePrince, Phys. Rev. A **100**, 032509 (2019).
[91] G. Gidofalvi and D. A. Mazziotti, J. Chem. Phys. **129**, 134108 (2008).
[92] J. Fosso-Tande, T.-S. Nguyen, G. Gidofalvi, and A. E. DePrince III, J. Chem. Theory Comput. **12**, 2260 (2016).
[93] D. A. Mazziotti, Phys. Rev. Lett. **117**, 153001 (2016).
[94] A. J. Coleman, Rev. Mod. Phys. **35**, 668 (1963).
[95] D. A. Mazziotti, Phys. Rev. A **85**, 062507 (2012).
[96] R. M. Erdahl, Int. J. Quantum Chem. **13**, 697 (1978).
[97] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," (2015), software available from tensorflow.org.
[98] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: composable transformations of Python+NumPy programs," (2018).
[99] M. Rocklin, in *Proceedings of the 14th Python in Science Conference*, edited by K. Huff and J. Bergstra (2015) pp. 130 – 136.
[100] J. F. Stanton, J. Gauss, J. D. Watts, and R. J. Bartlett, J. Chem. Phys. **94**, 4334 (1991).
[101] G. Daniel, J. Gray, *et al.*, J. Open Source Softw. **3**, 753 (2018).
[102] D. J. Rowe, Rev. Mod. Phys. **40**, 153 (1968).
[103] K. Chatterjee and K. Pernal, J. Chem. Phys. **137**, 204109 (2012).

Appendix A: Python code generating tensor contractions for the CCSD residual equations

The following code will generate NumPy einsum contractions for the CCSD singles and doubles residual equations. A fully working spin-orbital CCSD code can be found at https://github.com/edeprince3/pdaggerq/.

```
import pdaggerq

from pdaggerq.parser import contracted_strings_to_tensor_terms

pq = pdaggerq.pq_helper("fermi")
pq.set_print_level(0)

# energy equation
pq.set_bra("")
print('\n', '    < 0 | e(-T) H e(T) | 0> :', '\n')
pq.add_st_operator(1.0, ['f'], ['t1', 't2'])
pq.add_st_operator(1.0, ['v'], ['t1', 't2'])

pq.simplify()

# grab list of fully-contracted strings, then print
energy_terms = pq.fully_contracted_strings()
pq.clear()
for my_term in energy_terms:
    print(my_term)
energy_terms = contracted_strings_to_tensor_terms(energy_terms)

for my_term in energy_terms:
    print("#\t", my_term)
    print(my_term.einsum_string(update_val='energy'))
    print()

# singles equations
pq.set_left_operators(['e1(m,e)'])
print('\n', '    < 0 | m* e e(-T) H e(T) | 0> :', '\n')
pq.add_st_operator(1.0, ['f'], ['t1', 't2'])
pq.add_st_operator(1.0, ['v'], ['t1', 't2'])
pq.simplify()

# grab list of fully-contracted strings, then print
singles_residual_terms = pq.fully_contracted_strings()
singles_residual_terms = contracted_strings_to_tensor_terms(singles_residual_terms)
for my_term in singles_residual_terms:
    print("#\t", my_term)
    print(my_term.einsum_string(update_val='singles_res', output_variables=('e', 'm')))
    print()
pq.clear()

# doubles equations
pq.set_left_operators(['e2(m,n,f,e)'])
print('\n', '    < 0 | m* n* f e e(-T) H e(T) | 0> :', '\n')
pq.add_st_operator(1.0, ['f'], ['t1', 't2'])
pq.add_st_operator(1.0, ['v'], ['t1', 't2'])
pq.simplify()

# grab list of fully-contracted strings, then print
doubles_residual_terms = pq.fully_contracted_strings()
doubles_residual_terms = contracted_strings_to_tensor_terms(doubles_residual_terms)
for my_term in doubles_residual_terms:
    print("#\t", my_term)
    print(my_term.einsum_string(update_val='doubles_res', output_variables=('e', 'f', 'm', 'n')))
    print()

pq.clear()
```

Appendix B: Python code for generating tensor contractions for the CCSD Λ-amplitudes equations

The following code will generate NumPy einsum contractions for the CCSD singles and doubles de-excitation residual equations. A fully working spin-orbital Λ-CCSD code can be found at https://github.com/edeprince3/pdaggerq/.

```
#  <0| e(-T) H e*m e(T)|0>
print('\n', '    0 = <0| e(-T) H e*m e(T)|0> + <0| L e(-T) [H, e*m] e(T)|0>', '\n')
pq.set_left_operators(['1'])
pq.set_right_operators(['1'])
pq.add_st_operator(1.0,['f','e1(e,m)'],['t1','t2'])
pq.add_st_operator(1.0,['v','e1(e,m)'],['t1','t2'])

# <0| L e(-T) [H,e*m] e(T)|0>
pq.set_left_operators(['l1','l2'])
pq.add_st_operator( 1.0,['f','e1(e,m)'],['t1','t2'])
pq.add_st_operator( 1.0,['v','e1(e,m)'],['t1','t2'])
pq.add_st_operator(-1.0,['e1(e,m)','f'],['t1','t2'])
pq.add_st_operator(-1.0,['e1(e,m)','v'],['t1','t2'])
pq.simplify()

# grab list of fully-contracted strings, then print
singles_residual_terms = pq.fully_contracted_strings()
singles_residual_terms = contracted_strings_to_tensor_terms(singles_residual_terms)
for my_term in singles_residual_terms:
    print("#\t", my_term)
    print(my_term.einsum_string(update_val='lambda_one', output_variables=('m', 'e')))
pq.clear()


#  <0| e(-T) H e*f*nm e(T)|0>
print('\n','    0 = <0| e(-T) H e*f*nm e(T)|0> + <0| L e(-T) [H, e*f*nm] e(T)|0>', '\n')
pq.set_left_operators(['1'])
pq.set_right_operators(['1'])
pq.add_st_operator(1.0,['f','e2(e,f,n,m)'],['t1','t2'])
pq.add_st_operator(1.0,['v','e2(e,f,n,m)'],['t1','t2'])

# <0| L e(-T) [H,e*f*nm] e(T)|0>
pq.set_left_operators(['l1','l2'])
pq.add_st_operator( 1.0,['f','e2(e,f,n,m)'],['t1','t2'])
pq.add_st_operator( 1.0,['v','e2(e,f,n,m)'],['t1','t2'])
pq.add_st_operator(-1.0,['e2(e,f,n,m)','f'],['t1','t2'])
pq.add_st_operator(-1.0,['e2(e,f,n,m)','v'],['t1','t2'])
pq.simplify()

# grab list of fully-contracted strings, then print
doubles_residual_terms = pq.fully_contracted_strings()
doubles_residual_terms = contracted_strings_to_tensor_terms(doubles_residual_terms)
for my_term in doubles_residual_terms:
    print("#\t", my_term)
    print(my_term.einsum_string(update_val='lambda_two', output_variables=('e', 'f', 'm', 'n')))
    print()
pq.clear()
```

Appendix C: Python code for generating equations for the CCSD one-particle RDM

The following code will generate equations for evaluating the elements of the one-particle RDM at the CCSD level of theory. A fully working code to evaluate the one-particle RDM can be found at https://github.com/edeprince3/pdaggerq/.

```
import pdaggerq

pq = pdaggerq.pq_helper("fermi")

# D1(p,q) = <0|(1 + l1 + l2) e(-T) p*q e(T) |0>
pq.set_left_operators(['1','l1','l2'])

print('\n', '    D1(m,n):', '\n')
pq.add_st_operator(1.0,['e1(m,n)'],['t1','t2'])
pq.simplify()
d1_terms = pq.fully_contracted_strings()
for my_term in d1_terms:
    print(my_term)
pq.clear()

print('\n', '    D1(e,f):', '\n')
pq.add_st_operator(1.0,['e1(e,f)'],['t1','t2'])
pq.simplify()
d1_terms = pq.fully_contracted_strings()
for my_term in d1_terms:
    print(my_term)
pq.clear()

print('\n', '    D1(e,m):', '\n')
pq.add_st_operator(1.0,['e1(e,m)'],['t1','t2'])
pq.simplify()
d1_terms = pq.fully_contracted_strings()
for my_term in d1_terms:
    print(my_term)
pq.clear()

print('\n', '    D1(m,e):', '\n')
pq.add_st_operator(1.0,['e1(m,e)'],['t1','t2'])
pq.simplify()
d1_terms = pq.fully_contracted_strings()
for my_term in d1_terms:
    print(my_term)
pq.clear()
```

Appendix D: Python code for generating equations for the CCSD two-particle RDM

The following code will generate equations for evaluating the elements of the two-particle RDM at the CCSD level of theory. A fully working code to evaluate the two-particle RDM can be found at https://github.com/edeprince3/pdaggerq/.

```
import pdaggerq
pq = pdaggerq.pq_helper("fermi")

# D2(p,q,r,s) = <0|(1 + l1 + l2) e(-T) p*q*sr e(T) |0>
pq.set_left_operators(['1','l1','l2'])

blocks = [['i','j','k','l'],
          ['i','j','k','a'],
          ['i','j','a','l'],
          ['i','a','k','l'],
          ['a','j','k','l'],
          ['a','b','c','d'],
          ['a','b','c','i'],
          ['a','b','i','d'],
          ['i','b','c','d'],
          ['a','i','c','d'],
          ['i','j','a','b'],
          ['a','b','i','j'],
          ['i','a','j','b'],
          ['a','i','j','b'],
          ['i','a','b','j'],
          ['a','i','b','j']]

for my_block in blocks:
    print()
    print('#    D2(%s,%s,%s,%s)' %
         (my_block[0],
          my_block[1],
          my_block[2],
          my_block[3]))
    print()
    my_op = 'e2(' + my_block[0] + ',' \
                  + my_block[1] + ',' \
                  + my_block[3] + ',' \
                  + my_block[2] + ')'
    pq.add_st_operator(1.0,[my_op],['t1','t2'])
    pq.simplify()
    d2_terms = pq.fully_contracted_strings()
    for my_term in d2_terms:
        print("#\t", my_term)
    pq.clear()
```